Neural Sequence Models: A Formal Lens

Gail Weiss

Neural Sequence Models: A Formal Lens

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Gail Weiss

Submitted to the Senate
of the Technion — Israel Institute of Technology
Kislev 5783 Haifa December 2022

This research was carried out under the supervision of Professor Eran Yahav (Computer Science, Technion) and Professor Yoav Goldberg (Computer Science, Bar Ilan), at the faculty of Computer Science.

This thesis is based on the following main publications:

Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting automata from recurrent neural networks using queries and counterexamples. In *ICML 2018*, volume 80 of *PMLR*.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting automata from recurrent neural networks using queries and counterexamples (extended version). *Machine Learning, Springer*, 2022.

Gail Weiss, Yoav Goldberg, and Eran Yahav. On the practical computational power of finite precision rnns for language recognition. In *ACL 2018, Volume 2: Short Papers*.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Learning deterministic weighted automata with queries and counterexamples. In *NeurIPS 2019*.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like transformers. In *ICML 2021*, volume 139 of *PMLR*.

The following publications were part of my PhD research and present results that are supplemental to this work:

William Merrill, Gail Weiss, Yoav Goldberg, Roy Schwartz, Noah A. Smith, and Eran Yahav. A formal hierarchy of RNN architectures. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault, editors, *ACL 2020*, pages 443–459.

Daniel M. Yellin and Gail Weiss. Synthesizing context-free grammars from recurrent neural networks. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *TACAS 2021, as part of ETAPS 2021*, volume 12651 of *Lecture Notes in Computer Science*, pages 351–369.

The author of this thesis states that the research, including the collection, processing and presentation of data, addressing and comparing to previous research, etc., was done entirely in an honest way, as expected from scientific research that is conducted according to the ethical standards of the academic world. Also, reporting the research and its results in this thesis was done in an honest and complete manner, according to the same standards.

Acknowledgements

First, I would like to thank my advisors, Professors Eran Yahav and Yoav Goldberg. No individual project in this thesis would have happened, nor been completed as nicely, without their inspired direction(s), deep knowledge, and occasional pushes and shoves (as necessary) to finish what we'd started. I would like to thank them in particular for not giving up on me and/or RASP throughout the entire process, which mostly consisted of calming my frequent doubts about whether anyone would be interested—a testament to their far clearer vision and understanding of the field than mine.

On that note, I would like to thank my husband, Boris Pismenny, for closing my laptop and calming me down whenever I got too frustrated. Thank you for being solidly by my side ever since we met, for celebrating my victories, comforting me on my losses, supporting me in times of pressure, and dragging me out to wonderful vacations I never would have taken without you: I'm glad I have you along for the ride. I would also like to thank his parents, Alla and Misha, for their support and care in this time.

I would like to thank my parents, Simy and George, for both believing in and fostering our academic abilities throughout our lives, giving me the stable base from which to start this journey, and providing regular advice, encouragement, and perspective throughout. Thank you to my mom for supporting our (tiny) rebellions in England, and to my dad for always pushing us further than we got in school: I would not have gotten so far without either of you. I would also like to thank my siblings Hanna, Ben, and Jakey for their love and laughs, and especially Hanna who has been here for long enough to empathise with and support me in my various quirks and crises.

Back in academia, though not quite in this thesis, I would like to thank my external collaborators William Merrill, Prof. Roy Schwartz, Prof. Noah Smith, and Dr. Daniel Yellin. Thank you to Will and Danny in particular for bringing me along for your individual projects: working with you has brought me a lot of joy and motivation, and I hope the effect has been the same for you!

I would also like to thank all of other the great researchers who have enriched and motivated me along the way, among them: my candidacy and final exam committees for their valuable comments and interest; the wonderful members of the Formal Languages and Neural Networks (FLaNN) community for making such a great home for our field; and Prof. Yonatan Belinkov and his group for welcoming me in to their meetings and events. From a slightly earlier stage, I would like to thank the Weizmann Institute "Young Researchers" program for taking me in for my first research project back in high school, and in particular Prof. Hagai Perets, my mentor there and inspiring friend since.

All of the research in this thesis was improved by valuable discussions with and notes from various colleagues. Here I would like to thank in particular Dr. Uri Alon, Prof. Rémi Eyraud, Daniel Filan, Omri Gilad, Dr. Chris Hammerschmidt, Xiaokun Luan, and every one of our anonymous reviewers for their various comments, suggestions, and help, which have contributed to the works presented in this thesis. Similarly, I would like to thank Prof. Dana Angluin for her thoughtful correspondence, whose contribution here is marked by the merciful absence of a particularly dubious (false) claim from this thesis and my work in general.

I would like to thank my friends from before and since academia for making this whole experience more fun; my immediate and extended family for always believing in me; the faculty 3$^{\text{rd}}$ floor group for making it a great place to come to work while I was there; and my office and lab mates (in particular Dr. Uri Alon, Shaked Brody, and Prof. Hila Peleg) for picking up the baton since my later move to the 7$^{\text{th}}$ floor.

Two friends whose support deserves special mention are Avner Elizarov and Omri Gilad. Avner, thank you for immediately making the dorms a home away from home, for pulling me through the tough courses, taking me to the cool ones, calling me out on bad decisions, and making excuses to hang out whenever I needed it. Omri, thank you for the endless laughs, the unfathomable excitement and interest in my work, staying up late to teach me new concepts, and (generally) not judging my occasional pizza meltdowns.

All of this work was done at the Technion, and enabled by its excellent administrative staff. I would like to thank in particular Yifat Chen Solomon and Limor Gindin, whose combined knowledge has allowed me to enjoy blissful ignorance of the exact administrative rules I've been following throughout this degree. Yifat, thank you for easily handling tasks of every kind, giving good advice, and gracefully smoothing over my various bureaucratic blunders. Limor, thank you for simply being excellent at your job: you always know what to do next, and this degree has been easier for it.

I gratefully acknowledge the generous financial help of the Technion, the Henry and Marilyn Taub Faculty of Computer Science, The Vatat scholarship, and the following granting organisations: the European Union's Seventh Framework Programme (FP7); the Israeli Science Foundation; the Allen Institute for Artificial Intelligence; the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI); and the European Research Council (ERC).

Contents

List of Figures

Abstract

In recent years, there has been significant interest in the use of neural networks for processing sequential data, in particular using recurrent neural networks (RNNs) or transformers. These neural networks—highly parameterised, differentiable functions—are trained on large sets of input-output examples from a target concept, and boast strong results on many sequence processing tasks. They are used for everything from speech recognition and automatic translation to natural language generation, algorithmic trading, and more. But while their results are impressive, their representation is abstruse, making it hard to know what they have actually learned.

The work in this dissertation focuses primarily on the interpretation of such networks, which we refer to as *neural sequence models*. In particular, we present methods for the extraction of interpretable rules from trained models, and provide analyses of different architectures to better understand them.

For RNNs, there is a natural family of rules to focus on: RNNs have a clear analogue in deterministic finite automata (DFAs), and many works attempt to recover DFAs from RNNs for this reason. We propose a new method for extracting DFAs from RNNs, using the active DFA-learning algorithm L^* which had not been applied to RNNs before. To do this, we show how to efficiently approximate a response to L^*'s equivalence queries by modifying an existing algorithm for extracting DFAs from RNNs, ultimately playing the two algorithms off of each other in a successful iterative routine.

Expanding to the weighted case, we then design a weighted L^* variant. Here, the challenge is introducing a tolerance for minor differences between classifications into the L^* algorithm, such that it may be efficiently applied to a (noisy) trained RNN.

Our work on extraction from RNNs leads us to interesting observations on their behaviour. We find that one popular RNN variant—the LSTM—can successfully maintain a counter its state, while another—the GRU—cannot. We elaborate on the mechanisms behind these behaviours in our work.

Finally, in order to continue this line of research for transformers, we must find a natural analogue that will help us intuit about them in the same way that automata helped us understand and extract from RNNs. In this thesis we will propose such a model, presenting the symbolic programming language *RASP* (Restricted Access Sequence Processing).

Notation and Abbreviations

BOS	Beginning of Sequence (A token marking the beginning of a sequence)	114
DFA	Deterministic Finite Automaton (A finite state machine that processes input tokens one at a time, with each token progressing it from one state to the next according to a deterministic transition function)	5, 22
EOS	End of Sequence (A token marking the end of a sequence)	14
LM	Language Model (A model that defines a distribution over all possible finite sequences from an alphabet)	21
NSM	Neural Sequence Model (A neural network adapted to process variable-length sequences of input vectors)	11
PDFA	Probabilistic Deterministic Finite Automaton (A WDFA in which the weights on the transitions are normalised to define a next-token distribution over each state, such that it can be used as an (autoregressive) LM over a given alphabet)	22
RNN	Recurrent Neural Network (A neural sequence model that processes input sequences one vector at a time, maintaining in a single (or more) hidden vector an internal 'state' as it goes)	24
WDFA	Weighted Deterministic Finite Automaton (A DFA with an additional weight defined over each transition, such that the product of transitions a sequence takes in the DFA can be used to assign a weight to that sequence)	22

Chapter 1

The Story

Like the introduction, but more fun.

> In 1895 there were only 2 cars in the entire state of Ohio, yet they still ended up crashing into each other.
>
> Anonymous, Online
> *Also, a lie (on our internet!?)*

In the very beginning, long before there was even a research proposal, Eran and I just wanted to see how a neural net could help with program synthesis. This goal (and group) morphed quickly, but I will tire you with the story anyway.

We began with just DFAs: suppose some expert wants to encode some regular language (e.g. by DFA). They understand their language well enough to label any sample as being in or outside of the language (*'membership query'*), and even know to accept or (with some difficulty, and by way of counterexample) reject any automaton someone proposes to describe this language, though they would really rather not write such an automaton in full themselves. This expert would then, by definition, be a *minimally adequate teacher* for Dana Angluin's L^* algorithm, a DFA learning algorithm that I had just learned about for this exact purpose. L^* learns DFAs by making constant membership and equivalence queries to minimally adequate teachers, and would be a perfect match for our hypothetical expert, if only the latter didn't hate coming up with counterexamples so much.[1]

This is where our neural net was to jump in: before revealing each equivalence query to the expert, our neural net would help to generate extra membership queries, queries whose true labels we hoped—by nebulous reasoning—were likely to contradict those of the current proposed automaton. If the labels did indeed conflict with the given automaton, then their labelling could be used to effectively reject it before it ever

[1] Making up counterexamples sucks, which is why, around the same time, Eran was also working with Hila and Sharon on another method to avoid them—though their work deals with actual code [PSY18].

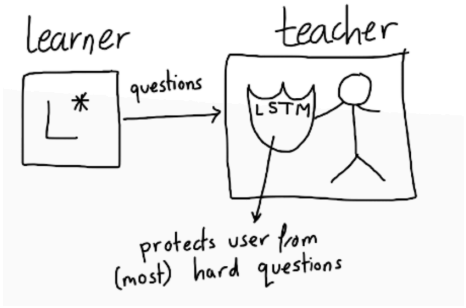


Figure 1.1: A sketch of our initial research project: a trained LSTM network acts as a shield, protecting our expert from L*'s difficult questions.

got to offending our expert's sensibilities (Figure 1.1).

Specifically, our neural network was an RNN, tasked with learning the target language alongside the L* algorithm by using the same samples that our expert was labelling anyway for L* . The idea was simply, at every equivalence query, to randomly sample several sequences and compare their labels from the RNN with those from the L* automaton, floating all disagreements to the expert for their true label. Yoav was quickly enrolled to this effort: someone had to make sure I didn't hurt myself on the RNNs.

We played with this toy for maybe one month (I checked) before Eran had a "crazy idea": what if we tried to figure out what the RNN was learning instead? Yoav soon told us this was actually a rather popular idea. He pointed us to several works on the extraction of automata from RNNs, which begin—as far as I can tell—with the works of Giles and colleagues, and of Watrous and Kuhn, though it is the former with which I became more familiar.

In their 1992 paper, Giles and colleagues propose a method to extract a DFA from a trained RNN, based on the reasonable assumption that "equivalent" state vectors in the RNN are generally arranged into small, easily separable clusters. The algorithm is straightforward: first, quantise the state space of the RNN, and treat each resulting 'block' of state vectors as a single (possibly unreachable) state in the extracted DFA. Then, map the transitions between these cluster-states by transferring all those found by a direct exploration of the RNN's actual states, starting from its initial state and continuing until all abstract cluster-states are either fully mapped or unreachable. If the automaton doesn't seem good enough—for example, it doesn't agree with the RNN on its training set—refine the quantisation, and start again.

We now had two straightforward, easy to implement algorithms for learning DFAs (one from experts and one from RNNs), which is two more than I had known about a month ago. Our first work is the result of smashing these into each other to create one that is neither.

This mix set a general direction for the rest of our work: we were going to pry open the black box of neural networks, and we were going to do it with tools from formal languages. We advanced our next goal to the extraction of *weighted* DFAs, which brought a completely new challenge to the application of L^*: how could we apply it, efficiently, to a model with potentially infinite different classifications? In an approach eventually hailed as "not hacky" in reviewer 4's post-rebuttal comments, we introduced a noise tolerance for considering two classifications as equal, and adapted all of L^* and its guarantees to work around this fuzzy new situation.

Around this point I was also lucky enough to meet Danny Yellin, who wanted to expand in the direction of context free grammars. Danny had read our L^*-based extraction work, and noticed something interesting in the sequence of hypothesis DFAs presented by L^*. Specifically, when L^* was attempting to extract a DFA from an RNN that had actually been trained on a simple context-free language, the differences between its subsequent hypotheses seemed to repeat specific patterns relevant to the language (see for example Figure 4.1). Danny suggested treating these DFAs as being generated by a small set of DFA building rules, which we defined and refer to as *Pattern Rule Sets (PRSs)*. By sufficiently constraining the space of possible PRSs, we were able to devise an algorithm for recovering a PRS from a given sequence of DFAs, and by extension to recover simple context free grammars (which the PRSs can be converted into) from trained RNNs [YW21].[2]

Meanwhile, merely working on extraction had actually directed us to a nice formal observation on RNNs: the LSTM architecture (unlike the GRU) was able to count. For me, this particular realisation was the serendipituous result of staring at debug prints while training LSTMs and GRUs on some simple non-regular languages (that turned out to be counting based), but I was not the first to see it: counting behaviour had actually been observed in LSTMs long before [GS01]. What was interesting was understanding why this was happening, and why it wasn't happening in GRUs, which required a familiarity with the exact inner workings of these RNNs. Luckily I had this familiarity, and for this I would like to credit the wonderful blog posts of Andrej Karpathy on RNNs [And15] and Christopher Olah on LSTMs [Chr15], the pytorch documentation [PGC+17], and my complete inability to use other people's code or packages without worrying too much.

[2]This work is not a part of this thesis.

We quickly published our explanation (along with a demonstration of the effect), opening a parallel line for our research which has led to a somewhat different and very enjoyable line of questioning: how can we use formal tasks to describe the *potential* expressive power of different neural networks?

Within a year, William Merrill (then under the guidance of Robert Frank and Dana Angluin!) formalised our results, introducing the much more rigorous concept of *saturated* neural networks for such analyses [Mer19].[3] We were lucky enough to meet a short time after, which immediately began a collaboration to further analyse the expressive power of different RNN variants, from more angles. Two years after the paper on LSTMs and GRUs, William, our collaborators, and I published a formal hierarchy of different RNN architectures, characterising the expressive power of several RNN variants, differentiating them across two main axes, and even considering the effects of pairing them with different final classifiers [MWG+20].[4]

All this time however, transformers had been growing in popularity, and it was getting impossible to ignore them (or Eran and Yoav's comments) any longer. This time I used Alexander Rush's excellent *Annotated Transformer* [Rus18] to get into the details, after which I had to accept an unfortunate observation: *these networks were nothing like any of the formal models with which we were familiar.*[5] This meant that, if we wanted to continue our work for transformers at all, we would have to begin with finding a relevant symbolic model to work with.

This unsurprisingly turned out to be a massive task. It took a while to advance from strange mathematical description to the general idea of a (loop-free) programming language, which would allow both symbolic representations and reflect the bounded depth of the transformer's computation. Then there was the question of how much control to give the user: the possible attention patterns advanced from a fixed finite set to something user definable. We argued over whether to use numerical or binary scores in the attention, agonised over how (if even possible) to make the attention-making operation readable, let alone the aggregation, and quibbled over what different things should be called at all (Figure 1.2).[6] For a brief time, map-reduce terminology was involved, but multimap-reduce-aggregate-average or whatever it all turns into when considering the attention operation soon killed that dream.

[3] In the original work, William refers to this as the *asymptotic* assumption, but we have since moved to *saturation* as the term (`https://lambdaviking.com/post/saturated-networks`).

[4] This work is also not a part of this thesis.

[5] I should note: several works have obtained results by comparisons with different circuit families! But they do not show that transformers are *like* circuits so much as that they can be *expressed* by them.

[6] When the transformer itself has no awareness of input order, does it really make sense to call its abstraction a *sequence* operator?

Figure 1.2: A key moment in designing RASP (Restricted Access Sequence Processing).

On one more exciting (and defining) evening, we figured out how to use the base transformer operations to count (!) conditionally (!!) (i.e., count different things according to the request of each position). From there it was just a quick ∼1 year journey to finish everything up and publish our results.

At ICML 2021 we presented RASP, a programming language that defines, more or less, the possible sequence operations computable with a transformer. RASP is hard to absorb and was described as "Matlab, but made by Satan" by my ever-supportive friend Omri Gilad,[7] to which my only response is that the transformer is worse.

[7] See `https://twitter.com/gail_w/status/1336080805454548994` for more such supportive comments.

Chapter 2

Introduction

The deep learning revolution! Since the advent of powerful, affordable GPUs,[1] and following a few notable successes on existing machine learning tasks,[2] deep neural networks have become the go-to model for almost every kind of task.

They can predict protein structures from amino acid sequences [JEP+21]. They can accurately recognise objects in images (the leading models on the Imagenet [DDS+09] leaderboard are neural network based), and generate beautiful new images from text [RDN+22; RBL+22]. They can transcribe your speech [CZH+21], translate written text (see [DCK20; YWC20] for surveys), answer questions (the SQuAD [RZLL16] leaderboard is dominated by deep neural nets), and even mimick your written speech in a convincing manner [BMR+20]). They can play chess, go, and others better than you [SHS+17]. They handle cars on the road, or at least aspire to [YLCT20]. And they have even been accused of being "slightly conscious" [Sut22]—though never without an ensuing debate.

But what fun is success on tasks without insight and understanding? And what trust can we give a machine we don't understand?

If we are going to learn anything from these fantastic models, and especially if we are going to use them in safety-critical applications, we have to understand how they are working. Obtaining this understanding—recognising their internal mechanisms and strengths, and recovering interpretable rules from trained networks—will be the primary focus of this work.

To keep the task manageable, we will restrict ourselves specifically to Neural Sequence Models (NSMs): neural networks that operate on variable-length sequences of input vectors, and within these we will focus further on *symbolic* input sequences: sequences in which the individual input vectors are chosen from a finite set, representing some finite input alphabet.

[1] Maybe space, time, and data was all deep neural networks ever really needed? Certainly [CMGS10] revel in the power afforded them by the latest technology.

[2] AlexNet famously beat all of its (non-neural) competitors on the ImageNet task [DDS+09] in 2012 [KSH12], and further research on speech and image processing in the same year [HDY+12; LRM+12] soon convinced huge industry players of the promise in the field.

The majority of the work will focus on the Recurrent Neural Network [Elm90], an NSM that processes input sequences one token at a time, maintaining a state as it goes. In this focus we will consider its existing parallel to the well researched model of Deterministic Finite Automata (DFAs) [CSM89], and exploit it both to recover weighted and unweighted DFAs from RNNs, and to fully understand the effect of one mechanism which we encounter in a popular RNN variant, the LSTM [HS97]. Finally however, we will turn to the popular transformer architecture [VSP+17], which has become the base of so many successful NSMs today. Unlike RNNs, transformers do not process their input sequences one token at a time but rather all at once, and so we need an entirely different kind of model to understand them. We will propose such a model in the form of a programming language, which we dub *RASP (Restricted Access Sequence Processing)*, and present empirical results supporting their similarity.

2.1 Background

The main concepts explored in this work—automata, RNNs, and transformers—are presented fully in Chapter 3. Still, we give a brief introduction to these concepts here.

2.1.1 DFAs, variants, and the L* Algorithm

DFAs

A plain *deterministic finite automaton* (DFA) is a finite state machine used to process and classify sequences from a given finite alphabet Σ in a deterministic manner. The machine classifies input sequences as follows: first, it starts in its single initial state q_0. Then, it reads the input sequence one token at a time, each time updating its state according to its deterministic transition function δ, which defines the next state from each current state and input token (effectively a lookup table). Once the sequence has been read completely, it checks if its current state is accepting or rejecting, and this is the classification of the sequence.

The set of sequences accepted by a DFA is the *language* defined by that DFA, and the set of all languages that can be defined by a DFA is called the set of *regular languages*.

DFA Variants

A *weighted* DFA (WDFA) is a DFA which also has a weight associated with each of its transitions and states. In this case, the output of the WDFA on each input sequence is not binary, but rather the product of the weights all the transitions taken in reading the sequence (including repetitions) and the weight of the final state it reaches. If we further require that at each state, the set of weights over its transitions and own weight form a distribution, and that there are no cycles in the DFA with 0 weight assigned to the states themselves, then the output of the WDFA forms a distribution over all

possible finite sequences $w \in \Sigma$, and we refer to it as a *probabilistic* DFA (PDFA). For a PDFA, we refer to the state- and outgoing transition- weights of each state as that state's *next-token distribution.*

Outside of weighting, DFAs can also be augmented by adding various memory structures to them, and allowing them to react to and control these structures alongside each transition. At the extreme, we have Turing machines: DFAs augmented with an infinite read/write 'tape' of memory cells and a head that can move back and forth along this tape.[3] More weakly, we have k-counter machines: DFAs augmented with k *counters*, which can only be controlled by resets, no-ops, or increases/decreases by one, and 'read' by comparison to zero [FMR68].

Note. A $k \geq 2$-counter machine may be described as Turing complete—i.e., able to compute any function that a Turing machine can—but this is only under a careful interpretation of Turing completeness that allows the input and output sequences to be appropriately encoded. When the inputs and outputs are not carefully encoded, there are Turing computable functions which a k-counter machine cannot compute. For example, for any input alphabet $|\Sigma| \geq 2$, no k-counter machine can recognise the set of palindromes of arbitrary length over Σ.

$\mathbf{L}^*$

The L^* algorithm [Ang87] is an algorithm for learning regular langauges from a *minimally adequate teacher*, that is, an oracle that can answer any *membership* or *equivalence* query about the language being learned. 1. *membership queries* are requests to classify a sequence: is it in the target language? 2. *equivalence queries* are requests to (informatively) classify a DFA: does it define the target language? And if not, what is a sequence it misclassifies (a counterexample)? L^* runs in polynomial time in the size of its final DFA, the alphabet, and the longest counterexample it receives. It keeps the responses to all of its membership queries in an *observation table* and generates an equivalence query every time it can find a DFA consistent with all of the observations it has made thus far.

2.1.2 Neural Sequence Models

A *neural network* is a highly parameterised differentiable function $g : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$, and a *neural sequence model* is a neural network that has been adapted to (meaningfully) handle a variable-length sequence of input vectors, $x \in (\mathbb{R}^{d_{in}})^*$. This adaptation can be either by special application of its immediate function (as in the case of RNNs), or direct adaptation of its internal implementation (as in the case of Transformers).

[3] Turing machines use and control this head and tape at each state transition by reading the current cell's value (which is treated as another component of their current state), deciding how (if at all) the head should move next, and what new value should be written in the current cell before moving.

RNNs

Recurrent Neural Networks (RNNs) are NSMs built on a simple neural network, specifically, a neural network consisting of an 'initial state' $h_0 \in \mathbb{R}^{d_{state}}$ and update function $g : \mathbb{R}^{d_{state}} \times \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{state}}$ as follows: g takes two vectors h_t and x_t, interpreted as current state and current input token, and returns a single new vector which is interpreted as the next state: $g(h_t, x_t) = h_{t+1}$. These networks (g, h_0) are applied to input sequences similarly to DFAs, with the main difference between the models being that the states and input tokens of RNNs are presented as real-valued vectors as opposed to symbols from finite sets [CSM89].

By pairing an RNN with a state classification function $f : \mathbb{R}^{d_{state}} \to C$ we can treat it as a classifier. If $C = \{Acc, Rej\}$ then g, h_0, f define a language over the input alphabet Σ similarly to a DFA, and we call the trio an RNN-*acceptor*. If C contains tuples $(w_1, ..., w_{|\Sigma|}, w_{EOS})$ of weights for each input token and one additional End of Sequence (EOS) token, this can be interpreted as a weight for each input token transition and each state (by the EOS token), defining a weight for each input sequence similarly to a weighted DFA. As in WDFAs, if we further require that C contain only distributions rather than sets of weights—and make sure that the trio (g, h_0, f) avoids loops with zero EOS weight—then (g, h_0, f) defines a distribution over all finite sequences over Σ, and we call it a *language model*-RNN (LM-RNN). In LM-RNNs as in PDFAs, we refer to the tuple $f(h) = (w_1, ..., w_{|\Sigma|}, w_{EOS})$ of each state h as its *next-token distribution*.

The actual functions g and f can take on many forms, including among them the popular LSTM [HS97] and GRU [CvMBB14; CGCB14b] architectures. Our DFA- and WDFA- extraction algorithms are agnostic to these differences (Chapters 4 and 6), but the actual expressive power of the networks is not, and we remark on this in a separate work Chapter 5 and a later collaboration [MWG$^+$20].

Transformers

Note. A transformer is a combination of two very similar networks, a *transformer-encoder* and a *transformer-decoder*, where the encoder is used to process input sequences and the decoder to autoregressively generate new ones, possibly with the help of processed information it got from an encoder. In both cases, the networks themselves are *length-preserving* functions $f : (\mathbb{R}^d)^* \to (\mathbb{R}^d)^*$, meaning that $|f(x)| = |x|$ for every input sequence $x \in (\mathbb{R}^d)^*$. In our work we consider only the transformer-encoder, and refer to it simply from here on as a *transformer* for convenience.

Like RNNs, transformers [VSP$^+$17] are neural networks adapted to process sequences, but their adaptation is internal, and their overall behaviour is much more complicated and parallelised. Specifically, while an RNN processes input sequences one token (input vector) at a time, a transformer processes all of its input tokens (vectors) in parallel, recombining them differently at each position according to the values in that position. While the full description is given in Section 3.4, we try to give a brief

overview of their behaviour here.

Each transformer (encoder) is composed of a finite number L of layers (with the input going into the bottom layer, and then the output of each layer feeding into the next layer), each of which can each be divided roughly into two main sublayers: a length-preserving *feed-forward* sublayer $\mathcal{F} : (\mathbb{R}^d)^* \to (\mathbb{R}^d)^*$ and a length-preserving *self-attention* sublayer a : $(\mathbb{R}^d)^* \to (\mathbb{R}^d)^*$. (In a full transformer, the decoder will also have an encoder-attention sublayer, which it uses to gather information from the encoder.)

The *feed-forward sublayers* are straightforward: each effectively consists of a single function $f : \mathbb{R}^d \to \mathbb{R}^d$ that is applied independently to each input vector $x_i \in \mathbb{R}^d$ passed to the sublayer. The implementation of this function (a 2-layer feed-forward network) is powerful for local manipulations [HSW89a], but the independent applications mean that the sublayer itself cannot meaningfully process the overall input sequnce, e.g., it cannot even check if there were two appearances of the same token.

The *attention sublayers* are the only part of the transformer that process the input sequence $x \in (\mathbb{R}^d)^*$ as a whole. They consist of *attention heads*, which define functions by which the information from all the input vectors will be mixed and shared to each output vector. Specifically, each attention head computes an *attention matrix* over the positions—a weight $w_{i,j}$ for every ordered pair of positions $i, j \in [|x|]$—-and then computes weighted averages of the information from its input vectors, for each output position i according to its weights $w_{i,j}$ over the input positions j. The exact details are presented in Section 3.4.

Embeddings Both the feed-forward and attention sublayers of the transformer are *permutation equivariant*, meaning that the only effect of permuting the vectors in a transformer's input sequence is to identically permute the vectors in its output sequence. Hence to differentiate between input sequences such as "apples are fruits" and "fruits are apples", each input vector passed to a transformer actually encodes a (token,position) pair,[4] obtained by summing a token and a positional embedding for that pair.

2.2 Overview

This thesis covers four papers: two on extracting DFAs from RNNs (one for plain DFAs from RNN-acceptors [WGY18a; WGY22]), and one for PDFAs from LM-RNNs [WGY19]), and two on analysing NSM architectures, specifically: one exploring a tangible difference between two popular RNN architectures [WGY18b], and one proposing a framework for intuiting about the more complicated transformer architecture [WGY21]. These works inspired some collaborations which are not included in this thesis: in [YW21] we build on our DFA extraction work to recover some simple context

[4] For transformer *decoders*, due to a subtle difference in their attention sublayer, this may not actually be necessary [HRP+22].

free grammars from RNNs, and in [MWG+20] we consider further differences between a wider array of RNN architectures, exploring what kind of DFA variations each RNN architecture is most like, and even how the strength of the classification functions f paired with them can affect their expressive power.

2.2.1 Extracting DFAs from RNNs

In Chapter 4, we present a method for recovering DFAs from trained RNN-acceptors. Given the close analogue of DFAs to RNNs [CSM89], this task has been extensively explored: see e.g. [GMC+92; OG96; CSS03; MS17] for examples, or [Jac05; WZO+17] for surveys. We differentiate from existing works by approaching the task with *exact learning* as opposed to state space clustering, employing the L* DFA learning algorithm (described briefly above and more fully in Subsection 3.2.1) to recover a DFA from the RNN.

Specifically, we take the RNN-acceptor R from which we wish to extract a DFA, and designate it as teacher for the L* algorithm. R can be used trivially answer membership queries, but equivalence queries—in which we must determine whether a given DFA is equivalent to R—are more challenging. As this problem is likely to be intractable, we use an approximation. One approach would be random sampling; however, should R and $\mathcal{A}$ be similar, this may take time.[5]

Instead, we answer equivalence queries with the help of abstractions of the RNNs, based on the general partition-mapping method used in previous works (described fully in Subsection 3.3.1). For our case, we define a custom, gradually refined partitioning p. We avoid this partitioning becoming unmanageably large—a risk in some previous methods—by only creating a single new cluster at every refinement, and only refining it when proven insufficient to describe the RNN.

The partitioning-based abstraction A_p and the L* DFA $\mathcal{A}$ act as two hypotheses for the RNN's ground truth, and the extraction does not terminate until the two are at least equivalent to each other. Whenever the two disagree on a sample w, we find its true classification in R, obtaining through this either a counterexample to $\mathcal{A}$ or a refinement to the partitioning p.

We show our method to efficiently and correctly recover many DFAs from RNNs trained to classify regular languages, expand easily to languages with minimal DFAs that are larger than any recovered in previous work, and even experiment a little with non-regular languages, obtaining interesting regular approximations of our true targets.

This work was presented initially in ICML 2018 [WGY18a], and later in an extended format in the Springer Machine Learning Journal, in the 2022 Special Edition on Grammatical Inference [WGY22].

[5] A later work has since considered this approach, presenting the algorithm alongside a relevant PAC analysis [MY18]

2.2.2 LSTMs can count, GRUs can't

In our DFA extraction work, we work with the popular RNN architectures LSTM [HS97] and GRU [CvMBB14; CGCB14b], and as a by-product of this become familiar with their specific implementations. Meanwhile, through experiments on a non-regular synthetic language which we refer to as *balanced parentheses* (BP, a minor variant of the dyck-1 language), we realise we are challenging the neural networks to count: recognising BP language requires following how many open vs closed parentheses there are at every point in the input sequence, and verifying that 1. the balance never drops below zero, and 2. it reaches exactly zero at the end.

Meditating on these pieces of information—the internal architecture implementations, and the tasks we are challenging them with—leads us to a main observation as follows: the LSTM architecture is suited to count, while the GRU architecture is not.

While we leave the exact details Chapter 5, the overall explanation is as follows: part of the LSTM state is composed of an unbounded vector $c \in \mathbb{R}^{d_{state}}$, which is updated at each input token by an independently gated *addition* between its current values and new candidate values—allowing straightforward implementation of increase/decrease and reset mechanisms on each value in c. Further, the activations used in creating these gates and candidate values are such that the values $\{0, 1\}$ (for gates) and $\{-1, 1\}$ (for candidate values) are easy to approximate—meaning, they do not require carefully balanced weights—which further accomodates the implementation of the keep/reset and increase/decrease/leave mechanisms needed to implement counter machines.

In contrast, the GRU state is composed entirely of a bounded vector $h \in [-1, 1]^{d_{state}}$, which is updated at each input token by an *interpolation* between its current values and similarly bounded new candidate values. Bounded in this way, the GRU has no opportunity to directly implement any counter dimension in h, and ultimately struggles to implement any counting mechanism at all.

While we are not the first to observe the LSTMs using counting to successfully perform counting-based tasks (see for example [GS01]), we are the first to categorise their power as that of counter machines, and to explain exactly how the counting mechanism is achieved. In our paper, which was presented in ACL 2018 [WGY18b], we expand this analysis slightly to cover a couple more architectures, and present a small set of experiments on counting tasks which clearly display learned counting dimensions in the LSTM, but none in the GRU.

Our results have since both been replicated on more complex tasks [SGBS19] and inspired further works analysing the practical power of different neural architectures (see eg [Mer19; MWG$^+$20]).

2.2.3 Extracting PDFAs (and WDFAs) from RNNs

Having extracted DFAs from RNN-acceptors, we turn to extracting interpretable rules from LM-RNNs, which define a *distribution* over all possible input sequences.

At this point, much work has been done on learning *weighted finite automata* (WFAs)—a non-deterministic variant of WDFAs, in which each (state,token) pair may transition to multiple different new states—from observations. In particular, the *spectral learning* method has been introduced to construct WFAs consistent with large tables ("Hankel Matrices") of observations from a target classifier [BCLQ14], and applied to LM-RNNs by sampling their relevant Hankel Matrices [AEG18; OWSH20].

As WFAs are not analogous to RNNs in the same way as DFAs (RNNs are deterministic machines), we prefer to recover DFAs from RNNs as before. This time, we will need a probabilistic DFA to reflect the LM-RNN behaviour. We return to the L^* algorithm, which we can modify easily to the multi-class setting and apply to PDFAs by treating each state's next-token distribution as its classification. Unfortunately, this approach creates an extremely large number of potential state classifications C, and applying the simple modification to LM-RNNs—which are unlikely to return identical next-token distributions for *any* two different states $h_1 \neq h_2$—simply cannot return anything much more meaningful than a full mapping of the RNN's states.

Modifying L^* more carefully, we introduce a tolerance for the difference between two different next-token distributions which we will still consider identical. This has ramifications for the guarantees of the algorithm and even the definitions of closedness and consistency which L^* follows in its computations (described in Subsection 3.2.1). We follow through the effects of these changes, prove the relevant new guarantees on the resulting modified algorithm, and evaluate it on RNNs trained on several language modelling tasks. We also compare our extracted PDFAs to n-grams and WFAs which we recover from the same RNNs.

While our motivation was the recovery of PDFAs, our algorithm does not rely on the classifications of the states being *distributions*, and is ultimately suitable for learning any weighted DFAs (WDFAs).

This work was presented in NeurIPS 2019 [WGY19].

2.2.4 Finding an Analogue for Transformers

Having successfully extracted from and analysed differences in RNNs, we wish to achieve the same for the more modern and extremely powerful transformer architecture. Immediately we hit a fundamental obstacle: transformers do not have a well researched analogue in the same way that RNNs have automata. In fact, it would seem they have no analogue in existing models at all![6]

[6]Some recent works have shown how, under certain simplifying assumptions, transformers can be represented using different types of circuit families [HAF22; MSS22]. These results provide an interesting upper bound on transformer expressive powers! But they do not suggest a *parallel* between transformers and circuits, and in particular do not provide a framework for reasoning about a trained transformer's behaviour, such as the attention patterns and positionwise computations it might use in solving a task. In fact, while many works analyse the expressive powers of transformers (see e.g. [YBR+20; PBM21; Hah20] for a small sample), they are all are forced to do so without the framework

In this work, we propose such a computational model for transformers, with the help of the programming language RASP (*Restricted Access Sequence Processing.*[7]) The main intuition in using a programming language is as follows: first, we assume that the information input to each transformer layer ℓ_l can be kept available for each deeper layer ℓ_j, $j < l$, similar to how variables in code persist after creation. And second, we recognise that the computation *depth* of a transformers output—that is, the longest path of dependencies from an output value to an input value anywhere in the sequence—is bounded,[8] much like the depth of a computer program described by a short sequence of basic operations.

If so, all that remains is to define the set of legal operations that such a computer program may contain, and the inputs it receives.

1. The inputs are straightforward: they are the sequences of individual tokens and positions encoded in the transformers input embeddings, which we maintain separately for clarity: one sequence of individual tokens, and one sequence of individual positions.

2. The feed-forward sublayers are also straightforward: these are operations which map a single atomic operation to each value in a sequence, for example adding 1 to the sequence $[0, 1, 2]$ to yield the sequence $[1, 2, 3]$.

3. The attention sublayers are more complicated, and abstracted into two steps: first an attention pattern is created, and then that pattern is used to average (or *aggregate*) the values of a given sequence (one of the existing variables) into a new sequence, using a different weighting for each output position in accordance with the attention pattern. To keep things simple, we only create binary attention patterns, which we refer to instead as *selection* patterns. To avoid code repetition, selection patterns in RASP are given the same standing as sequences, meaning they can be reused in multiple aggregation operations.

Finally, to avoid having to write each new transformer abstraction as a function explicitly receiving the input and position sequences, we bake this behaviour directly into RASP:

1. Instead of manipulating *sequences* and *selection patterns*, RASP actually manipulates *sequence-to-sequence* and *sequence-to-selection-pattern* functions, which we refer to as *sequence operators* (*s-ops*) and *selectors* respectively, and

2. The input and position sequences (embeddings) are provided in the form of two base s-ops: `tokens` and `indices`, which describe the initial conversion of each

of an intuitively related computational model!

[7]So named to reflect the fact that it is designed to manipulate input sequences, but cannot do so arbitrarily

[8]Specifically in $O(L)$ where L is the number of transformer layers

input sequence to its individual tokens and token positions, respectively. For example: `tokens("hi")=["h","i"]`, and `indices("hi")=[0,1]`.

Manipulating functions as opposed to sequences and selection patterns directly means that the actual operations in RASP—`select`, `aggregate`, and all position-wise manipulations—do not apply directly to input sequences, but rather to s-ops. In particular, each RASP operation actually creates a composition of its input s-op (or s-ops) with the underlying operation being described: the line `a=v+1` in RASP creates a new s-op `a` whose output is defined simply: for each input sequence x and output position $i \in [|x|]$, $\mathtt{a}(x)_i \triangleq \mathtt{v}(x)_i + 1$.

To clarify: *the RASP s-ops are to transformers what DFAs are to RNNs, and the RASP language itself is actually how we build them: the set of rules which defines the set of all possible RASP s-ops.* In particular, while RASP could be Turing complete, the same is not true for the set of all possible s-ops which it can generate.

The RASP s-ops are not necessarily a perfect analogue to transformers: it is possible that they both under- and over- approximate their abilities in different aspects. Still, they provide an intuitive framework for reasoning about how a transformer might make use of its intermediate embeddings and attention heads to solve a task, and small experiments even suggest that for simple tasks, these intuitions can already be quite accurate.

This work was presented in ICML 2021 [WGY21].

2.3 Summary

In this thesis, we extensively explore the connection between RNNs and DFA variants, using it both to successfully extract small models from RNNs and to meaningfully analyse their practical expressive powers. Our extraction work takes on a completely different direction from previous methods, approaching the problem with the help of exact learning and achieving strong results in the process. Our analysis approaches the networks from a more practical viewpoint than previous works, observing and explaining tangible differences between different RNN architectures that had previously been considered equivalent.[9] We are motivated to continue such research on transformers, but here encounter a much more basic problem: there is no familiar model to connect to transformers to begin with. In this case, our work begins with inventing and exploring such a model, paving the way for future work.

[9] Due to previous interesting analyses, whose assumptions (infinite precision and computation time) unfortunately do not hold for how RNNs are used in practice [SS92]

Chapter 3

Preliminaries

The works presented in this thesis will focus heavily on formal languages and deterministic finite automata (DFAs), recurrent neural networks (RNNs), transformers, and the learning algorithm L*. These were presented briefly in the introduction, and are presented in full detail here.

3.1 Sequences, Notations, and Language Models

For a finite alphabet Σ, the set of finite sequences over Σ is denoted by Σ^*, and the empty sequence by ε. For any Σ and stopping symbol $\$ \notin \Sigma$, we denote $\Sigma_\$ \triangleq \Sigma \cup \{\$\}$, and $\Sigma^{+\$} \triangleq \Sigma^* \cdot \Sigma_\$$ – the set of $s \in \Sigma_\$ \setminus \{\varepsilon\}$ where the stopping symbol may only appear at the end.

For a sequence $w \in \Sigma^*$, its length is denoted $|w|$, its concatenation after another sequence u is denoted $u \cdot w$, its i-th element is denoted w_i, and its prefix of length $k \leq |w|$ is denoted $w_{:k} = w_1 \cdot ... \cdot w_k$. We use the shorthand $w_{-1} \triangleq w_{|w|}$ and $w_{:-1} \triangleq w_{:|w|-1}$. A set of sequences $S \subseteq \Sigma^*$ is said to be *prefix closed* if for every $w \in S$ and $k \leq |w|$, $w^k \in S$. *Suffix closedness* is defined analogously.

For any finite alphabet Σ and set of sequences $S \subseteq \Sigma^*$, we assume some internal ordering of the set's elements $s_1, s_2, ...$ to allow discussion of vectors of observations over those elements.

Language Models (LMs) Given a finite alphabet Σ, a *language model* M over Σ is a model defining a distribution P_M over Σ^*, i.e., a function $P_M : \Sigma^* \to [0, 1]$ such that $\sum_{w \in \Sigma^*} P_M(w) = 1$. For any $w \in \Sigma^*, S \subset \Sigma^{+\$}$, and $\sigma \in \Sigma$, $P = P_M$ induces the following:

- *Prefix Probability:* $P^p(w) \triangleq \sum_{v \in \Sigma^*} P(w \cdot v)$.
- *Last Token Probability:* if $P^p(w) > 0$, then $P^l(w \cdot \sigma) \triangleq \frac{P^p(w \cdot \sigma)}{P^p(w)}$ and $P^l(w \cdot \$) \triangleq \frac{P(w)}{P^p(w)}$.
- *Last Token Probabilities Vector:* if $P^p(w) > 0$, $P^l_S(w) \triangleq (P^l(w \cdot s_1), ..., P^l(w \cdot s_{|S|}))$.
- *Next Token Distribution:* $P^n(w) : \Sigma_\$ \to [0, 1]$, defined: $P^n(w)(\sigma) = P^l(w \cdot \sigma)$.

A language model that can directly compute its next-token distribution for each prefix is known as an *autoregressive* language model. If we assume that P≠NP, polynomial-time autoregressive language models are strictly weaker than polynomial-time language models in general[1].

3.2 Automata

Deteministic Finite Automata A Deterministic Finite Automata (DFAs) A is a tuple $\langle \Sigma, Q, i, F, \delta \rangle$, in which Σ is the alphabet, Q the set of states, $F \subseteq Q$ the set of accepting states, $i \in Q$ the initial state, and $\delta : Q \times \Sigma \to Q$ the transition function. For a given automaton we add the notation $f : Q \to \{Acc, Rej\}$ as the function giving the classification of each state, i.e. $f(q) = Acc \iff q \in F$, and the notation $\hat{\delta} : Q \times \Sigma^* \to Q$ as the recursive application of δ to a sequence, i.e.: for every $q \in Q$, $\hat{\delta}(q, \epsilon) = q$, and for every $w \in \Sigma^*$ and $\sigma \in \Sigma$, $\hat{\delta}(q, w{\cdot}\sigma) = \delta(\hat{\delta}(q, w), \sigma)$. As an abuse of notation, we use $\hat{\delta}(w)$ to denote $\hat{\delta}(i, w)$.

(Regular) Languages The classification of a word $w \in \Sigma^*$ by a DFA A is defined $A(w) = f(\hat{\delta}(w))$, and set of words it accepts, $L_A = \{w \in \Sigma^* \mid A(w) = Acc\}$, is said to be the *language recognised by A*. Any language which can be recognised by some automaton A is said to be a *regular language*.

Equivalence and Minimality Two automata A and B are *equivalent* if $L_A = L_B$, and an automaton $A = \langle \Sigma, Q, i, F, \delta \rangle$ is *minimal* if for every automaton $A' = \langle \Sigma, Q', i', F', \delta' \rangle$ equivalent to A, $|Q| \leq |Q'|$. Two states $q_1, q_2 \in Q$ of an automaton $A = \langle \Sigma, Q, i, F, \delta \rangle$ are *equivalent* if for every $w \in \Sigma^*$, $f(\hat{\delta}(q_1, w)) = f(\hat{\delta}(q_2, w))$, and an automaton is minimal iff it has no two equivalent states.

Presentation Note For visual clarity, 'sink reject states'—states $q \notin F$ for which $\delta(q, \sigma) = q$ for every σ—are not drawn in images of DFAs in this work. Thus for example the second DFA in Figure 4.1 actually has 3 states, and rejects the sequence "`)`".

Probabilistic and Weighted DFAs (PDFAs and WDFAs) Probabilistic Deterministic Finite Automata (PDFAs) are similar to DFAs, except there is additionally a distribution defined over the next-state transitions for each current state, such that this (now weighted) DFA additionally consists of a language model over all possible input sequences. Specifically, a PDFA is a tuple $A = \langle Q, \Sigma, \delta_Q, q^i, \delta_W \rangle$ such that Q is a finite set of states, $q^i \in Q$ is the initial state, Σ is the finite input alphabet, $\delta_Q : Q \times \Sigma \to Q$ is the transition function and $\delta_W : Q \times \Sigma_\$ \to [0, 1]$ is the transition weight function, satisfying $\sum_{\sigma \in \Sigma_\$} \delta_W(q, \sigma) = 1$ for every $q \in Q$. (A PDFA not satisfying this last condition is referred to simply as a *weighted* DFA (WDFA).)

[1] By reduction from an NP complete language, through the concatenations of samples from the language with their relevant proofs [LJL+21]

The recurrent application of δ_Q to a sequence is denoted by $\hat{\delta} : Q \times \Sigma^* \to Q$, and defined: $\hat{\delta}(q, \varepsilon) \triangleq q$ and $\hat{\delta}(q, w{\cdot}a) \triangleq \delta_Q(\hat{\delta}(q, w), a)$ for every $q \in Q, a \in \Sigma$, $w \in \Sigma^*$. We abuse notation to denote: $\hat{\delta}(w) \triangleq \hat{\delta}(q^i, w)$ for every $w \in \Sigma^*$. If for every $q \in Q$ there exists a series of non-zero transitions reaching a state q with $\delta_W(q, \$) > 0$, then A defines a distribution P_A over Σ^* as follows: for every $w \in \Sigma^*$, $P_A(w) = \delta_W(\hat{\delta}(w), \$) \cdot \prod_{i \leq |w|} \delta_W(\hat{\delta}(w_{:i-1}), w_i)$.

3.2.1 The L* Algorithm

Angluin's L* algorithm is an exact learning algorithm for regular languages [Ang87]. The algorithm learns an unknown regular language L over an alphabet Σ from a teacher T, generating as output a DFA $\mathcal{A}$ that accepts L.

L* interacts with an oracle (also referred to as a teacher) that must answer two types of queries: *membership queries*, in which the oracle must classify words presented by L*, and *equivalence queries*, in which the oracle must accept or reject automata proposed by L* based on whether or not they correctly represent the target language. If the oracle rejects an automaton $\mathcal{A}$, it must also provide a counterexample—a word that $\mathcal{A}$ misclassifies with respect to the target language. L* continues to present queries to the oracle until the oracle accepts a hypothesis $\mathcal{A}$, at which point it terminates and returns $\mathcal{A}$.

The L* algorithm is guaranteed to always present a minimal DFA consistent with all membership queries given so far. Additionally, provided the target language T is regular, L* is guaranteed to return a minimal DFA for T in polynomial time in $(|Q|+|w|+|\Sigma|)$, where $|Q|$ is the number of states in that DFA, Σ is the input alphabet, and $|w|$ is the length of the longest counterexample given by the oracle [Ang87; BJLS05].

In our work on extracting DFAs from RNNs (Chapter 4), we will use L* as a black box, without getting into the exact details of how it works—meaning the interaction above is all we need to know. But in our work on extracting *weighted* DFAs from RNNs (Chapter 6), we will get deeper into L*, adapting its internal behaviour to work in the weighted and moreover slightly noisy setting. In particular, we will modify the behaviour and definitions around the internal *observation table* which L* maintains.

3.2.2 The L* Internals

Observation Tables Given an oracle $\mathcal{O}$, an observation table for $\mathcal{O}$ is a sequence indexed matrix $O_{P,S}$ of observations taken from it, with the rows indexed by prefixes P and the columns by suffixes S. The observations are $O_{P,S}(p, s) = \mathcal{O}(p{\cdot}s)$ for every $p \in P$, $s \in S$. For any $p \in \Sigma^*$ we denote $\mathcal{O}_S(p) \triangleq (\mathcal{O}(p{\cdot}s_1), ..., \mathcal{O}(p{\cdot}s_2))$, and for every $p \in P$ the p-th row in $O_{P,S}$ is denoted $O_{P,S}(p) \triangleq \mathcal{O}_S(p)$. Intuitively, the rows (prefixes) of the observation table represent the different states in the DFA that L* is learning, with the columns (suffixes) serving to prove their differences (two prefixes p_1, p_2 with different observations on some suffix s, i.e. for which $\mathcal{O}(p_1{\cdot}s) = \mathcal{O}(p_2{\cdot}s)$,

must necessarily reach different states in the DFA).

Before every equivalence query, the L* algorithm queries its oracle until its observation table is *closed* and *consistent*, at which point it creates a DFA consistent with all of the observations in the table and presents it as an equivalence query to the oracle.

Closedness and Consistency An observation table is called *closed* if for every $p \in P$ and $\sigma \in \Sigma$, there exists a $p' \in P$ such that $\mathcal{O}_S(p{\cdot}\sigma) = \mathcal{O}_S(p')$. It is called *consistent* if for every $p_1, p_2 \in P$ such that $\mathcal{O}_S(p_1) = \mathcal{O}_S(p_2)$, for every $\sigma \in \Sigma$, $\mathcal{O}_S(p_1{\cdot}\sigma) = \mathcal{O}_S(p_2{\cdot}\sigma)$.

Intuitively, *closedness* means that as far as can be seen by the observations, for every current state in the table, a transition with any one of the input tokens will lead to a state that is already recorded in the table. Meanwhile, *consistency* means that if any two prefixes in the table seem to represent the same state, then their transitions on each of the input tokens will also lead to identical states.

An unclosed table is expanded by adding the new prefix $p{\cdot}\sigma \in P \times \Sigma$ for which a matching prefix $p' \in P$ did not already exist in the table. An inconsistent table is expanded by adding a new suffix $\sigma{\cdot}s \in \Sigma \times S$ on which two the seemingly matching prefixes $p_1, p_2 \in P$ turned out to disagree. Whenever the oracle returns a counterexample to L* , all of the prefixes of that example are added to P, and L* begins querying the oracle to fill these new rows in the table and then expand it until it is again closed and consistent.

Classically, L* recovers regular languages, meaning its oracle provides binary classifications for each input sequence. In our weighted DFA extraction work (Chapter 6), we will modify L* to the weighted case. For this we will use an oracle for the *last-token probabilities* of the target, $\mathcal{O}(w) = P^l(w)$ for every $w \in \Sigma^{+\$}$, and maintain $S \subseteq \Sigma^{+\$}$.

3.3 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) An RNN R is a parameterised function $g_R(h, x)$ that takes as input a state-vector $h_t \in \mathbb{R}^{d_s}$ and an input vector $x_{t+1} \in \mathbb{R}^{d_i}$ and returns a state-vector $h_{t+1} \in \mathbb{R}^{d_s}$. An RNN can be applied to a sequence $x_1, ..., x_n$ by recursive application of the function g_R to the vectors x_i, beginning from a given initial state $h_{0,R}$ associated with the network. When applying an RNN to a sequence over a finite alphabet, each symbol is deterministically mapped to an input vector using either a one-hot encoding[2] or an embedding matrix, the discussions in this work are agnostic to this choice. For convenience, we refer to input symbols and their corresponding input vectors interchangeably.

We denote the state space of a network R by $S_R \subseteq \mathbb{R}^{d_s}$, and by $\hat{g_R} : S_R \times \Sigma^* \to S_R$ the recursive application of g_R to a sequence, i.e. for every $h \in S_R$, $\hat{g_R}(h, \epsilon) = h$, and

[2] A one-hot encoding assigns each symbol in an alphabet of size v to an integer i in $1, ..., v$, and maps the symbol to an indicator vector in $\mathbb{R}^v$ where the ith entry is 1 and the others are 0.

for every $w \in \Sigma^*$ and $\sigma \in \Sigma$, $\hat{g_R}(h, w{\cdot}\sigma) = g_R(\hat{g_R}(h, w), \sigma)$. As an abuse of notation, we also use $\hat{g_R}(w)$ to denote $\hat{g_R}(h_{0,R}, w)$.

RNN-acceptors A binary *RNN-acceptor* is an RNN with an additional function $f_R : S_R \to \{Acc, Rej\}$ that receives a state vector h_t and returns an accept or reject decision. The RNN-acceptor R is the pair of functions g_R, f_R with associated initial state $h_{0,R}$. Typically, f_R is a log-linear classifier or a multi-layer perceptron. The classification of a word $w \in \Sigma^*$ by an RNN-acceptor R is defined $R(w) = f_R(\hat{g_R}(w))$, and the language defined (or *recognised*) by R is the set of words it accepts, $L_R = \{w \in \Sigma^* \mid R(w) = Acc\}$.

A given RNN-acceptor can be interpreted as a deterministic, though possibly infinite, automaton, which we do note is a more powerful model than that of deterministic *finite* automata.

LM-RNNs A language model RNN (LM-RNN) over an alphabet Σ is an RNN coupled with a prediction function $f_R : h \mapsto d$, where $d \in [0,1]^{|\Sigma_\$|}$ is a vector representation of a next-token distribution. LM-RNNs differ from PDFAs only in that their number of reachable states (and so number of different next-token distributions for sequences) may be unbounded. Assuming finite precision (including for integers) they become technically equivalent, but the number of possible next-token distributions is still unmanageably large, and an approximation will be required to convert an LM-RNN to a W- or P-DFA.

We drop the subscript R when it is clear from context.

Multi-layer RNNs RNNs are often arranged in layers ("deep RNNs"). In a k-layers layered configuration, there are k RNN functions $g_1, ..., g_k$, which are applied to an input sequence $x = x_1, ..., x_m$ as follows: x is mapped by g_1 to a sequence of state vectors $h_{1,1}, ..., h_{1,m}$, and then each sequence $h_{i,1}, ..., h_{i,m}$ is mapped by g_{i+1} to the sequence $h_{i+1,1}, ..., h_{i+1,m}$. For such multi-layer configurations, we take the entire state-vector at time t to be the concatenation of the individual layers' state vectors: $h_t = h_{1,t}{\cdot}\ h_{2,t}{\cdot}...{\cdot}h_{k,t}$. Generally, the classification component of a multi-layered RNN-acceptor or LM-RNN is applied only to the final state of the top layer: $f_R(h_t) = f'_R(h_{t,x})$ for some f'_R.

RNN Architectures The parameterised functions g_R and f_R can take many forms. The function f_R can take the form of a linear transformation or a more elaborate classifier. RNNs were first introduced by Elman [Elm90], with a simple form: in *Elman-RNNs* (also known as *simple-RNNs*), g_R is an affine transform followed by a non-linearity, $g_R(h, x) = \tanh(W^x x + W^h h + b)$. Here W^x, W^h and b are the parameters of the function that need to be trained, and have dimensions $d_s \times d_i$, $d_s \times d_s$, and $d_s \times 1$ respectively. Other popular forms are the Long Short-Term Memory (LSTM) [HS97] and the Gated Recurrent Unit (GRU) [CvMBB14; CGCB14b]. These more elaborate functions are based on a differentiable gating mechanism, and have been repeatedly demonstrated to be easier to train than the Elman RNN, and to robustly handle long-range sequential dependencies. We refer the interested readers to textbooks such as

[GBC16; Gol17] or to the documentation of the PyTorch framework [PGC+17] for their exact forms.

While the extraction algorithms we present in these works (Chapters 4 and 6) will be agnostic to the exact forms of g_R and f_R, applying equally to any RNN architecture, we will show in another section that their formulations do matter, and can even lead to substantial differences in the expressive power of different RNN architectures (Chapter 5).

3.3.1 RNN Abstraction

Given a neural network R with state space S and alphabet Σ, and a partitioning function $p\colon S \to \mathbb{N}$, Giles et al presented a method for extracting a DFA for which every state is a partition from p, and the state transitions and classifications are defined by a single sample from each partition [GMC+92]. Their method can be seen as a simple sheared exploration of the partitions defined by p. The exploration begins from the partition containing the initial state $p(h_{0,R})$, explores according to the network's transition function g_R, and shears wherever it reaches an abstract state (partition) that has already been visited. We present it as pseudocode in Algorithm 3.1.

We denote by $A_{R,p}$ the DFA extracted by this method from a network R and partitioning p, and denote all its related states and functions by subscript R, p.[3] Note that the algorithm is guaranteed to extract a deterministic finite automaton (DFA) from any network and finite partitioning.

Algorithm 3.1 Pseudo-code of RNN R exploration with state space partitioning $p : S \to \mathbb{N}$. The functions of the network are marked R subscript.

Method *map_transitions(R, p)*:
```
Q, F, δ ← ∅
New ← {h_{0,R}}
while New ≠ ∅ do
    h ← pop from New
    q ← p(h)
    if q ∉ Q then
        Q ← Q ∪ {q}
        if f_R(h) = Acc then F ← F ∪ {q}
        for σ ∈ Σ do
            h' ← g_R(h, σ)
            δ ← δ ∪ {((q, σ), p(h'))}
            New ← New ∪ {h'}
        end
    end
end
```

[3] The exact order of the exploration (i.e., selection of states from New) is not important, but if we want to be well defined we can assume that New is FIFO and that Σ has an order which the for loop over it follows. This would make the exploration a (sheared) BFS.

3.4 Transformers

Transformers–both encoders and decoders–are highly parallelisable neural networks, which implement a length preserving sequence-to-sequence function using several components. They are composed of multiple layers of *attention* and *feed forward* computations, connected with layer-norms, residual connections (also known skip connections), and linear transformations.

In this thesis, we will present an abstraction (RASP) of the transformer encoder (Chapter 7), and so for completeness we present the architecture here in full. We will begin with the high-level description, and then get into the details of the separate components.

Transformer-Encoders A transformer-encoder [VSP+17] with L layers, H heads, and input and internal dimensions d, m is a length-preserving [4] function $\mathcal{T} : (\mathbb{R}^d)^* \to (\mathbb{R}^d)^*$ parameterised by the weights of L *transformer-encoder layers* $\ell_1, ..., \ell_L$, each with H heads and input and internal dimensions d, m, and defined for every $X \in (\mathbb{R}^d)^*$ as follows:

$$\mathcal{T}(X) = \ell_L(...\ell_2(\ell_1(X)))$$

We interpret each $X \in (\mathbb{R}^d)^*$ as a sequence of $n = |X|$ input vectors, and occasionally also refer to it in matrix form $X \in \mathbb{R}^{n \times d}$, where each row is one of the input vectors. We abuse notation and use these representations interchangeably.

Transformer-Encoder Layer A *transformer-encoder layer* with input dimension d, internal dimension m, and H heads (such that $d/H \in \mathbb{N}$) is a length-preserving function $\ell : (\mathbb{R}^d)^* \to (\mathbb{R}^d)^*$ composed of one multi-headed attention $\mathcal{A}$ with input dimension d and H heads, one feed forward function $\mathcal{F}$ with input dimension d and internal dimension m, two layer-norm functions $\mathbf{n}_1, \mathbf{n}_2$ over d, and one linear transformation $l_A : (\mathbb{R}^d)^* \to (\mathbb{R}^d)^*$, as follows: for every $X \in (\mathbb{R}^d)^*$,

$$X_1 = X + l_A(\mathcal{A}(\mathbf{n}_1(X))) \tag{3.1}$$

$$\ell(X) = X_1 + \mathcal{F}(\mathbf{n}_2(X_1)) \tag{3.2}$$

The additions in both equations are referred to as *skip* or *residual* connections, a simple but helpful mechanism in neural networks [HZRS16]. The layer-norm, feed-forward, and skip connection components of the layer are all positionwise; were it not for the attention, the entire layer would be positionwise.

Permutation Equivariance of Transformers An interesting trait of the transformer architecture is that it has no inherent positional awareness, i.e., every transformer $\mathcal{T} : (\mathbb{R}^d)^* \to (\mathbb{R}^d)^*$ is permutation equivariant . Specifically: for any transformer

[4] i.e. for every $X \in (\mathbb{R}^d)^*$, $|f(X)| = |X|$

$\mathcal{T} : (\mathbb{R}^d)^* \to (\mathbb{R}^d)^*$, input sequence $X = x_1, x_2, ..., x_n \in \mathbb{R}^d$, and permutation function π over $[n]$, then $\mathcal{T}(\pi(X)) = \pi(\mathcal{T}(X))$ [5]. This is overcome using a *positional embedding*, which is added alongside the *token embedding* when using a transformer in practice.

Embedding Functions Transformers $\mathcal{T} : (\mathbb{R}^d)^* \to (\mathbb{R}^d)^*$ can be used to process non-empty sequences over a finite alphabet Σ by composing them with a simple length-preserving *embedding* function, $y_0 : \Sigma^+ \to (\mathbb{R}^d)^*$: $\mathcal{T}_{y_0}(x) \triangleq \mathcal{T}(y_0(x))$. This y_0 is in turn composed from a *token embedding* $w : \Sigma \to \mathbb{R}^d$ and *position embedding* $p : \mathbb{N} \to \mathbb{R}^d$, which are normally combined using addition: for every $x = x_1...x_n \in \Sigma^*$, $y_0(x_1, x_2, ..., x_n)_i = w(x_i) + p(i)$[6]. w and p are essentially lookup tables, whose values may be taken from existing or designed tables or learned alongside the transformer.

From here, whenever we refer to a transformer, we mean a transformer-encoder over some finite alphabet Σ and paired with an initial embedding y_0 as described above.

In theory transformers are designed over the reals, of course in practice they are implemented over floating point approximations.

We now present the main components of the transformer in detail.

3.4.1 Basic Functions

Linear Transformations A *linear transformation* is a function $l : \mathbb{R}^d \to \mathbb{R}^m$ parameterised by a matrix $M_l \in \mathbb{R}^{m \times d}$ and bias $b_l \in \mathbb{R}^m$, as follows: for every $x \in \mathbb{R}d$, $l(x) = M_l x^T + b$. Linear transformations are applied positionwise to input sequences, i.e. for $X \in (\mathbb{R}^d)^*$, $l(X) = M_l X^T + b$.

Softmax *Softmax* is a length-preserving function $\mathcal{S} : \mathbb{R}^* \to (0, 1]^*$ that creates distributions from any vector x of scalars as follows: denoting $n = |x|$, then for each $i \in [n]$:

$$\mathcal{S}(x)_i = \frac{e^{x_i}}{\sum_{i \in [n]} e^{x_i}}$$

ReLU The *Rectified Linear Unit ReLU* : $\mathbb{R} \to \mathbb{R}_{\geq 0}$ is the non-linear function $ReLU(x) = \max(x, 0)$.

3.4.2 Feed-Forward

A *feed-forward* function with input dimension d and internal dimension m is a positionwise function $\mathcal{F} : (\mathbb{R}^d)^* \to (\mathbb{R}^d)^*$ obtained by composing two linear transformations $L_1 : (\mathbb{R}^d)^* \to (\mathbb{R}^m)^*$,$L_2 : (\mathbb{R}^m)^* \to (\mathbb{R}^d)^*$ and ReLU, as follows: $\mathcal{F}(X) \triangleq L_2(ReLU(L_1(X)))$.

[5] As all components of transformers other than attention are positionwise, we need only consider attention in order to be convinced of this. We will see that at each output location i, attention is a function only of $l_K(X), l_V(X)$, and $l_Q(X)_i$, where the order of the rows of $l_K(X)$ and $l_V(X)$ does not matter as long as they remain aligned with each other.

[6] Note that without the position embedding, y_0 would be permutation equivariant, and so the combination $\mathcal{T}_{y_0}$ would be too (as $\mathcal{T}$ is permutation equivariant)—an undesirable trait for sequence processing.

The feed-forward component is positionwise (it manipulates each row of the input matrix (each position in the input sequence) independently), but the combination of two linear transformations with nonlinear activation between them provides strong (positionwise) expressive capacity [HSW89b]. In transformers, the feed-forward component is often referred to as the *feed-forward sublayer*.

In RASP, our abstraction of the transformer model (Section 7.3), we will represent the feed-forward sublayers by allowing the application of most basic mathematical, boolean, and character-manipulating functions globally to all positions in an input sequence.

3.4.3 Attention

Attention is a function originally devised to enable 'recollection' of previously processed data from a history of arbitrary length [BCB15; LPM15]. It can also be used to collect data from positions in past and future, which is useful in settings where full sequences are given at once–e.g. in masked language modelling, where a missing word must be recovered. Transformers use a variant called *scaled dot-product attention*.

Scaled Dot-Product Attention with input dimension m_i, inner dimension d, and output dimension m_o is a function $\mathrm{a} : (\mathbb{R}^{m_i})^* \to (\mathbb{R}^{m_o})^*$ parameterised by 4 linear transformations, $l_Q, l_K, l_V : \mathbb{R}^{m_i} \to \mathbb{R}^d$ and $l_O : \mathbb{R}^d \to \mathbb{R}^{m_o}$, defined for every $X \in (\mathbb{R}^{m_i})^*$ as follows:

$$\mathrm{a}(X) = l_O(\mathcal{S}\Big(\frac{l_Q(X)l_K(X)^T}{\sqrt{m}}\Big)l_V(X))$$

Note. Often attention is presented as $\mathcal{S}(\frac{QK^T}{\sqrt{m}})V$, placing l_O separately in the transformer definition, and omitting–only for the presentation–the biases in the linear transformations.

For convenience, from here on we refer to scaled dot-product attention simply as *attention*.

Transformers often use multiple attention heads in parallel, as follows:

Multi-Headed Attention Given input dimension m_i, output dimension m_o, and some H such that $d = \frac{m_i}{H} \in \mathbb{N}$, a *multi-headed attention* from m_i to m_o with H heads is simply the sum of H attention heads from m_i to m_o, each with inner dimension d.

$$\mathcal{A}(X) = \sum_{h \in [H]} \mathrm{a}_h(X)$$

In transformers, the attention (or more often, multi-headed attention) component is often referred to as the *attention sublayer*.

In RASP, we will represent the attention sublayers by way of two manipulations: `select`, which can create a binary attention matrix between all positions according to

symbolic comparisons between their current values, and `aggregate`, which can take a binary attention matrix and some representation v of the current values in the positions, and average into each output position the current v values of the positions it has attended to.

3.4.4 Layer-norm

Layer-norm A *single-row layer-norm* [BKH16] over dimension d is a function $\mathbf{n} : \mathbb{R}^d \to \mathbb{R}^d$ parameterised by vectors $a, b \in \mathbb{R}^d$, and defined for every $x \in \mathbb{R}^d$ and $i \leq d$ as follows:

$$g(x)_i = \frac{a_i(x_i - \bar{x})}{\sqrt{\text{var}(x)}} + b_i$$

where $\bar{x} = \frac{\sum_{j \in [d]} x_j}{d}$ is the mean of x and $\text{var}(x) = \frac{1}{d-1} \sum_{i \leq d} (x_i - \bar{x})^2$ is its variance.

A *layer-norm* over dimension d, $\mathbf{n} : (\mathbb{R}^d)^* \to (\mathbb{R}^d)^*$, is a positionwise application of a single-row layer-norm of dimension d.

Note. Layer-norm as stated above cannot be used in practice. In particular, to avoid division by zero, an additional parameter $\varepsilon \in \mathbb{R}$ is used and added to $\text{var}(x)$ before applying the square root. We denote this variant $\mathbf{n}_\varepsilon$, according to the parameter ε (note that $\mathbf{n}_0 = \mathbf{n}$).

Layer-norm makes transformers train better in practice, but its expressive effect on the transformer architecture—if any—is unclear. We will not represent it in our RASP abstraction.

Chapter 4

Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples

4.1 Introduction

In recent years, there has been significant interest in the use of neural models, and in particular recurrent neural networks (RNNs), for learning languages. Like other supervised machine learning techniques, RNNs are trained based on a large set of examples of the target concept.

RNNs can reasonably approximate a variety of languages, and even precisely represent a regular language [Cas98]. However, they are in practice unlikely to generalise exactly to the concept being trained, and what they eventually learn in actuality is unclear [OG00]. Indeed, several lines of work attempt to glimpse into the RNN black-box [GMC+92; ZGS93; OG96; CSS03; Jac05; KJL15; LCHJ15; LDG16; SGH+16; LBJ16; KCA16; SPK16; AKB+16; MS17; WZO+17; AMMS17].

In contrast to the supervised ML paradigm, the *exact learning* paradigm considers setups that allow learning a target language without approximation. For example, Angluin's L^* algorithm enables the learning of any regular language, provided a *teacher* capable of answering *membership* (request to label example) and *equivalence* (comparison of proposed language with target language) queries is available [Ang87].

In this work we use exact learning to elicit the true concept class of a trained recurrent neural network. This is done by treating the trained RNN as the teacher of the L^* algorithm. To the best of our knowledge, this is the first attempt to use exact learning with queries and counterexamples to extract an automaton from a given RNN.

Recurrent Neural Networks Recurrent neural networks (RNNs) are a class of neural

networks which are used to process sequences of arbitrary lengths. When operating over sequences of discrete alphabets, the input sequence is fed into the RNN on a symbol-by-symbol basis. For each input symbol the RNN outputs a *state vector* representing the sequence up to that point, combining the current state vector and input symbol at every step to produce the next one. An RNN is essentially a parameterised mathematical function that takes as input a state vector and an input vector, and produces a new state vector. The RNN is trainable, and, when trained together with a *classification component*, the training procedure drives the state vectors to provide a representation of the prefix which is informative for the classification task being trained.

Classification An RNN can be paired with a *classification component*, a classifier function that takes as input a state vector and returns a binary or multi-class classification decision. The RNN and the classifier are combined by applying the RNN to the sequence, and then the classifier to the final resulting state vector. When the classification component gives a binary classification for each state vector, the combination defines a binary classifier over sequences, which we call an *RNN-acceptor*. When the component gives a distribution over the possible next tokens, the combination defines a next-token distribution for each input sequence, which we call a *Language-Model RNN (LM-RNN)*.

A trained RNN-acceptor can be seen as a state machine in which the states are high-dimensional vectors: it has an initial state, a well defined transition function between internal states, and a well defined classification for each internal state. A trained LM-RNN is not immediately analogous to a binary state machine, but we will see in this work how it may be interpreted as a one, and under this interpretation also extracted from using our method.

RNNs play a central role in deep learning, and in particular in natural language processing. For more in-depth overview, see [GBC16; Gol16; Gol17].

We now turn to the question of understanding what an RNN has actually learned. We formulate the question around RNN-acceptors, but later (in Section 4.7) show how the solution relates to LM-RNNs.

Motivation Given an RNN-acceptor R trained over a finite alphabet Σ, our goal is to extract a deterministic finite-state automaton (DFA) A that classifies sequences in a manner observably equivalent to R. (Ideally, we would like to obtain a DFA that accepts *exactly* the same language as the network, but this is a much more difficult task.[1])

Note: In this work, when understood from context, we use the term RNN to mean RNN-acceptor. Additionally, we use "automata" to refer specifically to deterministic finite automata (DFAs) (as opposed to other automata variants, such as pushdown automata or weighted automata).

[1] In fact, given the results showing that some RNN architectures can count [GS01; WGY18b], a DFA may not be sufficient for representing the language learned by an RNN at all.

Previously existing techniques for DFA extraction from recurrent neural networks are based on creating an a-priori partitioning of the RNN's state space, and mapping the transitions between the resulting clusters (e.g., [GMC+92; ZGS93]). In this work however, we approach the question using exact learning.

Exact Learning In the field of exact learning, *concepts* (sets of instances) can be learned precisely from a *minimally adequate teacher*—an oracle capable of answering two query types [GK95]:

- *membership queries*: state whether a given instance is in the concept or not
- *equivalence queries*: state whether a given hypothesis (set of instances) is equal to the concept held by the teacher. If not, return an instance on which the hypothesis and the concept disagree (a *counterexample*).

The L* algorithm [Ang87] is an exact learning algorithm for learning a DFA from a minimally adequate teacher with knowledge of some regular language L. In this context, the concept is L, the instances are finite sequences ('words') over its alphabet, and the hypotheses are presented as automata $\mathcal{A}$ defining a regular language $L_{\mathcal{A}}$. L* completes when the oracle accepts its latest equivalence query, i.e. when $L_{\mathcal{A}} = L$.

Our Approach We treat DFA extraction from RNNs as an exact learning problem. We use Angluin's L* algorithm to elicit a DFA from *any type* of trained RNN, using the RNN as a *teacher*. In doing so, we maintain only a coarse partitioning of the RNN's state space, refining it only as much as necessary to answer L*'s queries.

RNNs as Teachers A trained RNN-acceptor can trivially answer membership queries, by feeding input sequences to the network for classification. Answering equivalence queries, however, is not so easy. The main challenge is that no finite interpretation of the network's states and transitions is given upfront: the states of an RNN are high-dimensional real-valued vectors, resulting in an infinite state space which cannot be exhaustively enumerated and compared to the hypothesis.

To address this challenge, we use a *finite abstraction* of the RNN R to answer equivalence queries: we define a finite partitioning of the state space, and create from it an automaton which can be compared to the hypothesis $\mathcal{A}$. A unique aspect of this setting compared to previous L* works is that we only observe *an abstraction* of the teacher. This means that when there is a disagreement between the teacher and the learner, it may be not that the learner is incorrect and needs to refine its representation, but rather (or also) that our abstraction of the teacher is not precise enough and must be refined. Indeed, at every equivalence query, the current finite abstraction and current proposed automaton $\mathcal{A}$ act as two hypotheses for the RNN R's ground truth, which must at least be equivalent to each other in order to both be equivalent to R. Thus, whenever the two disagree on a sample, we find its true classification in R, obtaining through this either a counterexample to $\mathcal{A}$ or a refinement to the abstraction.

Main Contributions The main contributions of this work are:

- We present a novel and general framework for extracting automata from trained RNNs, using the RNNs as teachers in an exact learning setting.
- We implement[2] the technique and show its ability to extract descriptive automata in settings where previous approaches fail. We demonstrate its effectiveness on modern RNN architectures—multi-layer LSTMs and GRUs.
- We describe how the technique can be used to learn DFAs from only positive examples, and demonstrate its effectiveness in this setting. To do so we show how to create RNN-acceptors from positive examples only, using a language modelling objective.
- We apply our technique to RNNs trained to 100% train and test accuracy on simple languages, and discover in doing so that some RNNs have not generalised to the intended concept. Our method easily reveals and produces *adversarial inputs*—words misclassified by the trained RNN and not present in the train or test set.

A shorter version of this work has been presented in ICML 2018 [WGY18a].

Notes The technique we will present in this work applies to all RNNs, and in particular is agnostic to possible differences in the update and classification functions g_R and f_R as described in Section 3.3. In our experiments, we use linear transformation for f_R, and the popular LSTM and GRU architectures for g_R. For the LSTM, whose transition function is often described as converting a triplet of input-vector, state-vector and memory-vector to a next state-vector and memory-vector, we treat the concatenation of the state-vector and memory-vector as a single state-vector with dimension $d_s = 2h_s$, where h_s is the hidden size of the cell.

4.2 Existing Approaches and Related Work

Soon after the introduction of the RNN [Elm90], it was shown that, when learning a regular language, a simple ("Elman-") RNN is able to cluster its reachable states in a manner that resembles a (not necessarily minimal) DFA for that language [CSM89][3]). Researchers soon began seeking ways to recover small DFAs from trained RNNs [WK91; GMC+92; ZGS93], and there has since been a lot of research on extracting rules, and in particular DFAs, from RNNs: see [WZO+17] and [Jac05] for partial surveys.

Transition Mapping Continuing from the above observation of Cleeremans, Servan-Schreiber, and McClelland, Giles, Miller, Chen, Chen, Sun, and Lee proposed a method for extracting DFAs from second-order RNNS, by traversing the clusters of states ac-

[2] `www.github.com/tech-srl/lstar_extraction`

[3] This work references a slightly older version of [Elm90].

cording to the RNN's behaviour [GMC+92][4]. In particular, given a neural network R with state space S and alphabet Σ, and a partitioning function $p\colon S \to \mathbb{N}$, [GMC+92] presents a method (Algorithm 3.1) for extracting a DFA abstraction of the network in which every abstracted state is an entire partition from p, and the transitions between abstracted states and their classifications are obtained by a single sample of the continuous values in each such partition.

In both their own work and more recent research by others (e.g. [WZO+17]), this extraction method has been shown to produce DFAs that are reasonably representative of given second-order RNNs—provided the given partitioning captures the differences between the network states well enough.

Quantisation For networks with bounded output values, [GMC+92] suggests dividing each dimension of the network state space into $q \in \mathbb{N}$ (referred to as the *quantisation level*) equal intervals, yielding q^{d_s} subsets of the output space with d_s being the length of the state vectors.

However, because this technique applies a uniform quantisation over the entire output space, it suffers from inherent state explosion and does not scale to the networks used in practice today: the original paper demonstrates the technique on networks with 8 hidden values, whereas today's can have hundreds to thousands.

Clustering Other state-partitioning approaches use *clustering* [CSS03; WZO+17; CCRB17]. In these approaches, an unsupervised classifier such as k-means is applied to a large sample set of reachable network states, creating a finite number of clusters. The sample states can be found by various methods, such as a BFS exploration of the network state space to a certain depth, or by recording all state vectors reached by the network when applied to its train set (if available). The partitioning of the state space defined by the clusters is then explored in a similar way to that described by [GMC+92]. Clustering approaches yield automata that are much smaller than those given by the partitioning method originally proposed in [GMC+92], making them more applicable to networks of today's standards.

Weaknesses In both of these approaches the partitioning is set before the extraction begins, with no mechanism for recognising and overcoming overly coarse behaviour. Both approaches thus face the challenge of choosing the best parameter value for extraction, and are generally applied several times with different parameter values, after which the 'best' DFA is chosen according to a heuristic (e.g., accuracy against RNN on the test set). Additionally, both approaches can still have rather large state space, and—as the exploration of the extracted DFA is performed blindly—these states cannot be merged until the extraction is complete and the DFA can be minimised.

Note on Architectures Many of these works use *second order RNNs* [GSC+90], which are shown to better map DFAs than simple RNNs [GGCC94; WZO+18]. In this work

[4] In later work, Omlin and Giles futher evaluated this method and the clustering of a trained RNN's states, confirming that the states are indeed organised in tight clusters [OG96].

however, we experiment on the popular GRU [CvMBB14; CGCB14b] and LSTM [HS97] architectures, as they are more widely used in practice.

4.2.1 Recent Works and Future Directions

Since the initial publication of this method, several other approaches for extracting DFAs have been suggested, and still other works have begun grappling with more complicated targets such as weighted automata or context free languages.

DFAs [MY18] released an L*-based approach for learning DFAs from *any* neural network architecture, answering equivalence queries by drawing random samples over the input alphabet and checking if they are counterexamples to the proposed automaton. Their work analyses this approach from a PAC learning perspective and applies also to completely black box models, in contrast to our own work and other extraction works listed above (which rely on access to the RNN's hidden state from different prefixes). In Section 4.6.7, we compare our method to this approach, highlighting the advantage of the abstraction based approach to equivalence queries when the hidden state is available.

[WN19] propose *state-regularised RNNs*, a variant of RNNs that is regularised towards transitioning between a finite number of learned internal states. Their work discusses both training these new RNNs and the recovery of DFAs from them once trained, presenting an extraction method tailored to their proposed architecture.

WFAs [AEG18] use spectral learning ([BCLQ14]) to extract weighted, non-deterministic finite automata (WFAs) from any black box language model, evaluating on RNNs. [OWSH20] also apply spectral learning for WFA extraction, but this time to whitebox RNNs, using an adaptation of the equivalence query presented in this work to refine the WFA beyond the initial spectral extraction. In a later work, we adapt L* to a weighted setting, extracting weighted deterministic finite automata (WDFAs) from any black box language model [WGY19]. Finally, more recently, [ZDX+21] expand on the partitioning and then transition-mapping approach of the classical DFA extraction papers to recover WFAs from RNNs without using exact or spectral learning.

CFGs With the understanding that some RNN architectures behave more like counter machines [GS01; WGY18b; SGBS19], which are more expressive than DFAs, and indeed that an RNN in general might be trained on something more complicated than a regular language, it becomes interesting to consider extraction of context free languages (CFGs) from RNNs.

Recently, [YW21] use the DFA-extraction method presented in this work as the initial step in an algorithm for extracting a subclass of CFGs from trained RNNs,[5] and [BBF+21] apply results on *visibly pushdown languages* and *tree automata* to extract a different subclass of CFGs, also from trained RNNs. Independently, there exist several

[5] By creating an algorithm for generalising CFGs from a sequence of DFAs, and using the hypotheses provided by L*as that sequence.

works on learning (subclasses of) CFGs from queries, or from examples only, that have not yet been applied for extraction from RNNs [Sak92; Yok03; Tel06; CE07; Cla10; DFG10; SY16; CY16; Yos19].

4.3 Learning Automata from RNNs using L*

In the following sections we show how to build a teacher for the L* algorithm around a given RNN-acceptor R. The teacher must be able to answer membership and equivalence queries as required by L* .

To implement **membership queries** we rely on the RNN classifier itself. To determine whether a given word w is in the unknown language L_R, we simply run the RNN on this word, and check whether it accepts or rejects w.

To implement **equivalence queries** we check the equivalence of the L* hypothesised automaton $\mathcal{A}$ against an abstraction $A_{R,p}$ of the network, where p is a partitioning over the network's state space. If we find a disagreement $w \in \Sigma^*$ between $\mathcal{A}$ and the current abstraction $A_{R,p}$, we use R to determine whether this is because the L* hypothesis is incorrect (i.e., $L_R(w) \neq \mathcal{A}(w)$), or a result of a poor abstraction (i.e., $L_R(w) \neq A_{R,p}(w)$). In the former case ($L_R(w) \neq \mathcal{A}(w)$), we end the equivalence query and return w as a counterexample to L* . Otherwise, we refine p and restart the comparison of $\mathcal{A}$ and $A_{R,p}$. If no such disagreement w is found (i.e., $\mathcal{A}$ and $A_{R,p}$ are equivalent), we accept L*'s hypothesis and the extraction ends.

p is maintained between equivalence queries, i.e., the partitioning p at the start of the $j{+}1^{\text{th}}$ equivalence query is the same partitioning p from the end of the j^{th} equivalence query.

In theory, the extraction continues until the automaton proposed by L* is accepted, i.e., $\mathcal{A}$ and $A_{R,p}$ converge. In practice, for some RNNs this may take a long time and yield a large DFA (>30,000 states). To counter this, we place time or size limits on the interaction, after which the last L* hypothesis is returned.[6] We see that these DFAs still generalise well to their respective networks.

The partitioning p has to be coarse enough to facilitate feasible computation of $A_{R,p}$, but fine enough to capture the interesting observations made by the network. As we have an iterative setting, we can satisfy this by starting with a very coarse initial abstraction and refining it only sparingly, whenever it is proven incorrect.

The equivalence queries are described in Section 4.4, and the partitioning and its refinements in Section 4.5.

[6] We could also return the last abstraction, $A_{R,p}$, and focus on refining p over returning counterexamples. But we find that the abstractions are often less accurate (see Section 4.6.8). We suspect this is due to the lack of 'foresight' $A_{R,p}$ has, as opposed to L*'s many separating suffix strings (loosely, L* works by maintaining two growing lists of 'interesting' prefixes and suffixes, generating an equivalence query only when all the prefixes going into the each hypothesis state have the same classification on all of the suffixes).

Note Convergence of $A_{R,p}$ and $\mathcal{A}$ does not guarantee that R and $\mathcal{A}$ are equivalent. Providing such a guarantee would be an interesting direction for future work.

4.4 Answering Equivalence Queries

Given a network R, a partitioning function p over its state space S, and a proposed minimal automaton $\mathcal{A}$, we wish to check whether the abstraction of the network $A_{R,p}$ is equivalent to $\mathcal{A}$, preferably while exploring as little of $A_{R,p}$ as necessary. If the two are not equivalent—meaning, necessarily, that at least one is not an accurate representation of the network R—we wish to find and resolve the cause of the inequivalence, either by returning a counterexample to L* (and so refining $\mathcal{A}$), or refining the partitioning function p (and so the abstraction $A_{R,p}$) in the necessary area. Hence our equivalence query must be able not only to return counterexamples when necessary, but also to specifically identify overly-coarse partitions in the partitioning p.

For clarity, from here onwards we refer to the continuous network states $h \in S$ as R-states, the abstracted states in $A_{R,p}$ as A-states, and the states of the L* DFAs $\mathcal{A}$ as L-states.

In this section we describe the details of an equivalence query assuming a given partitioning p and refinement operation `refine`. We present our initial partitioning p_0 and `refine` operation in Section 4.5.

4.4.1 Parallel Exploration

The key intuition to our approach is the fact that $\mathcal{A}$ is minimal, and so each state in the DFA $A_{R,p}$ should—if the two automata are equivalent—be equivalent to exactly one state in the DFA $\mathcal{A}$. This is based on the fact that for automata $A = \langle \Sigma, Q, i, F, \delta \rangle$ and $A' = \langle \Sigma, Q', i', F', \delta' \rangle$ in which A' is minimal, A and A' are equivalent if and only if there exists a mapping $m : Q \to Q'$ satisfying that $m(i) = i'$, $f(q) = f'(m(q))$, and $m(\delta(q, \sigma)) = \delta'(m(q), \sigma)$ for every $q, \sigma \in Q \times \Sigma$.

To check the equivalence of $A_{R,p}$ and $\mathcal{A}$ without necessarily having to fully explore $A_{R,p}$, we build such a mapping between their states on-the-fly: we associate between states of the two automata during the extraction of $A_{R,p}$, by traversing $\mathcal{A}$ in parallel to the extraction of $A_{R,p}$ (which is extracted according to algorithm 3.1). We update this association for all R-states visited during this extraction, i.e., including those at which the traversal is sheared.[7] Any inconsistencies (*conflicts*) in this association are definite indicators of inequivalence between $A_{R,p}$ and $\mathcal{A}$.

Conflict types We refer to associations in which an accepting A-state is associated with a rejecting L-state or vice versa as *abstract classification conflicts*. We refer to multiple but disagreeing associations for a single A-state, i.e. situations in which one

[7] These are important: they are the repeat visits to an A-state, from which a *partitioning conflict* may occur.

A-state is associated with two different (minimal) L-states, as *partitioning conflicts*. (The inverse, in which one minimal L-state is associated with several A-states, is not a problem: $A_{R,p}$ is not necessarily minimal and so these states may be equivalent.)

Recalling that the ulterior motive is to find inconsistencies between the proposed automaton $\mathcal{A}$ and the given network R, and that the exploration of $A_{R,p}$ runs atop an exploration of the actual R-states, we also check at each point during the exploration whether the current R-state $h \in S_R$ has identical classification to that of the current L-state reached in the parallel traversal of $\mathcal{A}$. As the classification of a newly discovered A-state is determined by the R-state with which it was first mapped, this also covers all abstract classification conflicts. We refer to failures of this test generally as *classification conflicts*, and check only for them and for partitioning conflicts.

4.4.2 Conflict Resolution and Counterexample Generation

Classification conflicts are a sign that a path $w \in \Sigma^*$ satisfying $R(w) \neq \mathcal{A}(w)$ has been traversed in the exploration of $A_{R,p}$, and so necessarily that w is a counterexample to the equivalence of $\mathcal{A}$ and R. They are resolved by returning the path w as a counterexample to L* , so that it may refine its observations and provide a new automaton. All that is necessary for this is to maintain the current path w throughout the exploration.

Partitioning conflicts are a sign that an A-state $q \in Q_{R,p}$, that has already been reached with a path w_1 during the exploration of $A_{R,p}$, has been reached again with a new path w_2 for which the L-state is different from that of w_1. In other words, partitioning conflicts give us two sequences $w_1, w_2 \in \Sigma^*$ for which $\hat{\delta_{R,p}}(w_1) = \hat{\delta_{R,p}}(w_2)$ but $\hat{\delta_{\mathcal{A}}}(w_1) \neq \hat{\delta_{\mathcal{A}}}(w_2)$. We denote by $q_1, q_2 \in Q_{\mathcal{A}}$ the L-states reached in $\mathcal{A}$ by these sequences, $q_i = \hat{\delta_{\mathcal{A}}}(w_i)$. As $\mathcal{A}$ is a minimal automaton, q_1 and q_2 are necessarily inequivalent, meaning there exists a differentiating suffix $s \in \Sigma^*$ for which $f_{\mathcal{A}}(\hat{\delta_{\mathcal{A}}}(q_1, s)) \neq f_{\mathcal{A}}(\hat{\delta_{\mathcal{A}}}(q_2, s))$, i.e. for which $f_{\mathcal{A}}(w_1{\cdot}s) \neq f_{\mathcal{A}}(w_2{\cdot}s)$. Conversely, as $\hat{\delta_{R,p}}(w_1) = \hat{\delta_{R,p}}(w_2)$ then $\hat{\delta_{R,p}}(w_1{\cdot}s) = \hat{\delta_{R,p}}(w_2{\cdot}s)$, and so $f_{R,p}(w_1{\cdot}s) = f_{R,p}(w_2{\cdot}s)$.

Clearly in this case $\mathcal{A}$ and $A_{R,p}$ must disagree on the classification of either $w_1{\cdot}s$ or $w_2{\cdot}s$, and so at least one of them must be inconsistent with the network R. In order to determine the 'offending' automaton, we pass both $w_1{\cdot}s$ and $w_2{\cdot}s$ to R for their true classifications. If $\mathcal{A}$ is found to be inconsistent with the network, the word on which $\mathcal{A}$ and R disagree is returned to L* as a counterexample.

Else, $w_1{\cdot}s$ and $w_2{\cdot}s$ are necessarily classified differently by the network, and $A_{R,p}$ should not lead w_1 and w_2 to the same A-state. The R-states $h_1 = \hat{g}(w_1)$ and $h_2 = \hat{g}(w_2)$ are passed, along with the current partitioning p, to a refinement operation, which refines p such that the two are no longer mapped to the same A-state—preventing a reoccurrence of that particular conflict.

The previous reasoning can be applied to w_2 with *all* paths w_1 that have reached the conflicted A-state $q \in Q_{R,p}$ without conflict before w_2 was traversed. As such, the

classifications of *all* the words $w_1 \cdot s$ are tested against the network, prioritising returning a counterexample over refining the partitioning.[8] If eventually it is the partitioning that is refined, then the R-state that triggered the conflict, $h = \hat{g}(w_2)$, is split from all R-states $h_1 = \hat{g}(w_1)$ for w_1 that have already reached q in the exploration, in one single refinement.[9]

Every time the partitioning is refined, the guided exploration starts over, and the process repeats until either a counterexample is returned to L*, equivalence is reached (exploration completes without a counterexample), or some predetermined limit (such as time or partitioning size) is exceeded. We note that in practice—and very often so with the decision-tree based refinement operation that we present—there are cases in which starting over is equivalent to merely updating the associated A-state $p(h)$ of the R-state h that triggered the refinement and continuing the exploration from there, and we implement our equivalence query to take advantage of this.

In our implementation, whenever we find several potential counterexamples to the proposed DFA, we check them in order of increasing length and return the shortest counterexample we have found.

4.4.3 Algorithm

Pseudocode for this entire equivalence checking procedure (ignoring preference for shortest counterexamples) is presented in Algorithms 4.1 and 4.2, with the main loop in algorithm 4.1.[10] The description here assumes the existence of a refinement operation `refine` separating in the partitioning an R-state h from a set of other R-states H, we present such a method in Section 4.5.

The overall iterative process, including the refinements to p, is desribed in `check_equivalence`, and the equivalence checking for a specific partitioning p is given in `parallel_explore`.

`parallel_explore` attempts to build $A_{R,p}$ in variables Q, F, q_0, δ, while also maintaining the associations of these states to R and $\mathcal{A}$ as follows:

- `Visitors` holds for every A-state q the set of all R-states h satisfying $p(h) = q$ that have been visited during the exploration. This is used for refinements triggered by partitioning conflicts.

- `Path` holds for every R-state h the sequence $w \in \Sigma^*$ with which h has been visited during the exploration.[11] This is used for generating potential counterexamples

[8] As we will ultimately return the last L* hypothesis and not the abstraction if time runs out (see Subsection 4.6.8).

[9] At least, this is the ideal case. In practice, we allow a relaxed setting where it might only be split from some (non-empty) subset of them. In the worst case, this will trigger a further refinement when the query is attempted again.

[10] And full code is available online at `www.github.com/tech-srl/lstar_extraction`.

[11] Technically this should be maintained as a *set* of sequences reaching h, but in practice, the probability of there being more than one such sequence per h is too low to consider.

Algorithm 4.1 Pseudo-code for equivalence checking of an RNN R and minimal DFA $\mathcal{A}$, with initial partitioning p_0. The main loop is in *check_equivalence*. The supporting functions *update_records* and *handle_partition_conf* are presented in algorithm 4.2.

Method *parallel_explore*$(R, \mathcal{A}, p)$**:**
empty all of: Q, F, δ, Unexplored, Visitors, Path, Association
$q_0 \leftarrow p(h_{0,R})$
update_records$(q_0, h_{0,R}, q_{\mathcal{A},0}, \varepsilon)$
while *Unexplored* $\neq \emptyset$ **do**
  $h \leftarrow$ Pop(Unexplored)
  $q \leftarrow p(h)$
  $q_{\mathcal{A}} \leftarrow$ Association(q)
  if $f_R(h) \neq f_{\mathcal{A}}(q_{\mathcal{A}})$ **then**
    return *Reject, Path(h)*
  end
  if $q \in Q$ **then**
    continue
  end
  $Q \leftarrow Q \cup \{q\}$
  if $f_R(h) = Acc$ **then**
    $F \leftarrow F \cup \{q\}$
  end
  for $\sigma \in \Sigma$ **do**
    $h' \leftarrow g_R(h, \sigma)$
    $q' \leftarrow p(h')$
    $\delta(q, \sigma) \leftarrow q'$
    $q'_{\mathcal{A}} \leftarrow \delta_{\mathcal{A}}(q_{\mathcal{A}}, \sigma)$
    if $q' \in Q$ *and* Association$(q') \neq q'_{\mathcal{A}}$ **then**
      return *handle_partition_conf*$(q', h',$ Association$(q'), q'_{\mathcal{A}})$
    end
    update_records$(q', h', q'_{\mathcal{A}},$Path$(h)\cdot\sigma)$
  end
end
return *Accept,* ε

Method *check_equivalence*$(R, \mathcal{A}, p_0)$**:**
$p \leftarrow p_0$
verdict $\leftarrow$ Restart_Exploration
while *verdict = Restart_Exploration* **do**
  verdict, $w \leftarrow$ parallel_explore$(R, \mathcal{A}, p)$
end
return *verdict,w*

Algorithm 4.2 Supporting functions (in pseudo-code) for equivalence checking of an RNN R and minimal DFA $\mathcal{A}$, with initial partitioning p_0. The main loop is in *check_equivalence*, presented in algorithm 4.1.

Method *update_records*$(q, h, q_{\mathcal{A}}, w)$**:**
  Visitors$(q) \leftarrow$ Visitors$(q) \cup \{h\}$
  Path$(h) \leftarrow w$
  Association$(q) \leftarrow (q_{\mathcal{A}})$
  Push(Unexplored,h)

Method *handle_partition_conf*$(q, h, q_{\mathcal{A}}, q'_{\mathcal{A}})$**:**
  find $s \in \Sigma^*$ **s.t.** $f_{\mathcal{A}}(q_{\mathcal{A}}, s) \neq f_{\mathcal{A}}(q'_{\mathcal{A}}, s)$
  for $h' \in$*Visitors(q)* **do**
    $w \leftarrow Path(h')\cdot s$
    if $f_R(w) \neq f_{\mathcal{A}}(w)$ **then**
      return *Reject, w*
    end
  end
  $p \leftarrow$ *refine*$(p, h,$Visitors$(q)\backslash\{h\})$
  return *Restart_Exploration,* ε

when handling a partitioning conflict.

- `Association` holds for every A-state q the L-state $q' \in Q_{\mathcal{A}}$ visited in the parallel exploration of $\mathcal{A}$ the first time that q was visited. If at any point q is visited while the parallel exploration is on a different state $q'' \neq q'$, a partitioning conflict is triggered.

Note that finding the separating suffix for two inequivalent states q_1, q_2 of a given automaton A can be done by a simple parallel BFS exploration of the states reachable from q_1 and q_2 in A, continuing until two states with opposite classifications are found.

4.5 Abstraction and Refinement

Given a partitioning p, an R-state h, and a set of R-states $H \subseteq S \setminus \{h\}$, we must refine p to obtain a new partitioning p' satisfying:

1. for every $h_1 \in H$, $p'(h) \neq p'(h_1)$, and
2. for every $h_1, h_2 \in S$, if $p(h_1) \neq p(h_2)$ then $p'(h_1) \neq p'(h_2)$.

The first condition separates (in the partitioning) the R-states that caused the partitioning conflict leading to the refinement. The second condition maintains separations made by earlier refinements, i.e., it prevents previously created abstract states from being merged.

We want to generalise the information given by h and H well, so as not to invoke excessive refinements as new R-states are explored. Additionally, we would like to keep the partitioning as small as possible, so that $A_{R,p}$ can be explored and compared to $\mathcal{A}$ in reasonable time at every equivalence query.

To keep the partitioning small, we settle on a decision tree structure, in which each refinement only splits the partition in which the conflict was recognised. Additionally, seeing that in practice our equivalence checking method can overcome imperfect splits between H and h by generating further splits if necessary, we relax the first condition. Specifically, we allow the classifiers splitting between H and h in the conflicated partition to not do so perfectly, provided they separate at least some of H from h.

Our method is unaffected by the length of the R-states, and very conservative: each refinement increases the number of A-states by exactly one. Our experiments show that it is fast enough to quickly find counterexamples to proposed DFAs.

4.5.1 Initial Partitioning

In addition to a refinement method, our algorithm needs an initial partitioning p_0 from which to start the first equivalence query. As we wish to keep the abstraction as small as possible, we begin with no state separation at all: $p_0 : h \mapsto 0$.

4.5.2 Decision-Tree based Partitioning, with Support Vector Refinement

Let $h \in S, H \subset S$ be the R-states with which a refinement was invoked. We know the refinement is only applied to h, H satisfying $p(h) = p(h')$ for every $h' \in H$. To keep the partitioning small, we define a gentle refinement operation, in which for every call we only split the single partition $p(h)$. This approach avoids state explosion by adding only one A-state per refinement.

Decision Tree It is natural to maintain a partitioning p refined over time in this way as a decision tree, where each internal node tracks some single refinement made to p, and its leaves are the current A-states of the abstraction.

SVM classifiers At every refinement, for the split of $p(h)$, we would like to allocate a region around the R-state h that is large enough to contain other R-states that behave similarly, but separate from neighbouring R-states that do not. We achieve this by fitting an SVM [BGV92] classifier with an RBF kernel[12] to separate h from H (splitting the partition $p(h)$ in exactly two). The max-margin property of the SVM ensures a large space around h, while the Gaussian RBF kernel allows for a non-linear partitioning of the space. We use this classifier to split the A-state $p(h)$, yielding a new partitioning p' with exactly one more A-state than p.

[12]While we see this as a natural choice, other kernels or classifiers may yield similar results. We do not explore such variations in this work.

Whenever the SVM successfully separates h from H entirely, this approach satisfies the requirements of refinement operations. Otherwise, the method fails to satisfy condition 1 of the refinement operation. Nevertheless, the SVM classifier will always separate at least one of the R-states $h' \in H$ from h, and later explorations can invoke further refinements if necessary. In practice we see that this does not hinder the main goal of the abstraction, which is finding counterexamples to equivalence queries.

Unlike mathematically defined partitionings such as the quantisation proposed by [GMC+92], our abstraction's storage is linear in the number of A-states it can map to; and computing an R-state's associated A-state may be linear in this number as well (e.g. if the decision tree is a chain). Luckily, as this number of A-states also grows very slowly (linearly in the number of refinements), this does not become a problem.

4.5.3 Practical Considerations

As the initial partitioning and the refinement operation are very coarse, our method runs the risk of accepting very small but wrong DFAs early in the extraction.

To counter this, two measures are taken:

1. At the beginning of extraction, one accepting and one rejecting sequence are provided to the teacher, and then checked as potential counterexamples at the beginning of every equivalence query.[13] Conversely, if these are not available, equivalence queries are extended with n random samples for some small n (e.g. $n = 100$) and range of lengths (e.g. 0-100): whenever $\mathcal{A}$ and $A_{R,p}$ are equivalent, n random samples are generated and checked as potential counterexamples ($\mathcal{A}(w) \neq R(w)$) before $\mathcal{A}$ can be accepted.
2. The first refinement is aggressive, generating a greater (but still manageable) number of A-states than made with the main single-partition split approach used for the rest of the extraction.

The first measure is taken specifically to prevent erronous termination of the extraction on a single state automaton, and requires only two samples (if provided) or short additional time before accepting an equivalence query.

The second measure prevents the extraction from too readily terminating on small DFAs, by creating a (manageably) large $A_{R,p}$ that will hopefully capture a relatively rich representation of the RNN. Our method for it is presented in Section 4.5.3.

Aggressive Difference-based Refinement

At the first refinement, instead of splitting $p_0(h)$ to separate h from all or most of H using a single SVM, we split S in its entirety across multiple dimensions chosen according to h and H. Specifically, we calculate the mean h_m of H, find the d dimensions

[13] When using these in our experiments, we used the shortest possible examples, e.g., the empty sequence and) for the balanced parentheses language.

with the largest gap between h and h_m, and then split S along the middle of that gap for each of the d dimensions.

The resulting partitioning can be comfortably stored in a decision tree of depth d. It is intuitively similar to that of the quantisation suggested in [GMC+92], except that it focuses only on the dimensions with the greatest deviation of values between the states being split, and splits the 'active' range of values.

The value d may be set by the user, and increased if the extraction is suspected to have converged too soon. We found that $d = 10$ generally provides a strong enough initial partitioning of S, without making the abstraction too large for feasible exploration.

4.6 Experimental Results

We first demonstrate the effectiveness of our method on LSTM- and GRU-acceptors[14] trained on the Tomita grammars (1982), which have been used as benchmarks in previous automata-extraction work [WZO+17], and then on substantially more complicated languages. We show the effectiveness of our refinement-based equivalence query approach over that of plain random sampling and present cases in which our method extracts informative DFAs where other approaches fail. In addition, for some seemingly perfect networks, we find that our method quickly returns counterexamples representing deviations from the target language.

We clarify that when we refer to extraction time for any method, we consider the *entire* process: from the moment the extraction begins, to the moment a DFA is returned.[15]

Prototype Implementation and Settings We implemented all methods in Python, using PyTorch [PGC+17] and scikit-learn [PVG+11]. For the SVM classifiers, we used the SVC variant, with regularisation factor $C = 10^4$ to encourage perfect splits and otherwise default parameters—in particular, the RBF kernel with gamma value 1/(num features).

All training and extraction was done on amazon instances of type `p3.2xlarge`, except for the BP and email classifier RNNs which were run on `p2.xlarge`.

[14] While many previous automata-extraction works evaluate on second-order RNNs [GSC+90], we evaluate on the more popular LSTM and GRU architectures. We note that with the exception of quantisation-based partitioning [OG96], which requires minor adaptation for unbounded RNN state space, all of these methods—including our own—can be applied to any RNN architecture.

[15] Covering among others: abstraction exploration, abstraction refinements (including training SVM classifiers), and L* refinements (for our method), and total time for all created DFAs (for k-means clustering). Unless otherwise stated, this time is measured using the `process_time` method in python's time module.

4.6.1 Languages

We consider the Tomita Grammars (4.6.4), and more complicated regular languages defined by small, randomly sampled DFAs (4.6.4). We also consider the language of legal email addresses (defined precisely in 4.6.9), and the language of *balanced parentheses* (BP): the set of sequences over `()a-z` in which the parentheses are balanced, e.g. `a(a)ba` and `()(())`.

4.6.2 Sample Sets and Training

Tomita and Random Regular Languages We use train, validation, and test sets of sizes 5000, 1000 and 1000 containing samples of lengths 1-100 (uniformly distributed). To get 'representative' sample sets, we define a distribution over each DFA's state transitions favouring transitions which do not reduce the number of reachable states,[16] sample from that distribution, and train the RNN to provide correct output for all prefixes of every sample (as opposed to only the full samples).[17] We train these RNNs with the Adam optimiser, using initial learning rate 0.0003, an exponential learning rate scheduler with gamma 0.9, and dropout 0.1. Each RNN was trained for up to 100 epochs on its train set, or until the validation set had 100% accuracy for 3 epochs in a row, whichever came sooner.

Balanced Parentheses and Email Addresses We generated positive samples using tailored functions,[18] and negative samples as a mix of both random sequences and mutations of the positive samples.[19] Here we train the RNN only on the full samples (as opposed to classifying every prefix). We trained all networks to 100% accuracy on their train sets, and considered only those that reached 99.9+% accuracy on a test set consisting of up to 1000 uniformly sampled words of each of the lengths $n \in 1, 4, 7, ..., 28$. The positive to negative sample ratios in the test sets were not controlled. The BP and email train sets were randomly generated during training. The BP train set created ≈44600 samples, of which ≈60% were positive for each RNN, and reached balanced parentheses up to depth 11. The email addresses train set created 40000 samples.

[16] (E.g., a transition into a sink reject state—unless it also comes from the sink reject state—reduces the number of reachable states.)

[17] The intuition behind this choice is that every 'irreversible' transition in the DFA (e.g., the first `0` in a sample for Tomita 1, the language of sequences containing only `1`) is delated, increasing the time spent in the states before them, which might otherwise be underrepresented in the samples.

[18] For instance, a function that creates emails by uniformly sampling 2 sequences of length 2−8, choosing uniformly from the options `.com`, `.net`, and all `.co.XY` for `X,Y` lowercase characters, and then concatenating the three with an additional `@`.

[19] With mutations obtained by adding, removing, changing, or moving up to 9 characters.

4.6.3 Details on Our Extraction (Practical considerations)

We apply the measures discussed in Section 4.5.3 as follows: First, for all networks, we apply our method with aggressive initial refinement depth $d = 10$ (Section 4.5.3). Second, we use additional counterexamples:

Additional Counterexamples For the Tomita and random DFA languages, during extraction, we used random samples as additional potential counterexamples. Specifically, whenever an equivalence query was going to accept, we considered an additional 100 potential counterexamples, each generated as follows: first, we choose a length from $0-10$ (uniformly), and then uniformly sample a sequence of that length over the RNN input alphabet.

For BP and email addresses, during extraction, we presented each RNN along with one positive and one negative sample to check for counterexamples at each equivalence query. These were chosen as the shortest positive and shortest negative word in the train set of the RNN, in particular: for BP, the initial samples were the empty sequence (positive) and `)` (negative), and for emails, the initial samples were `0@m.com` (positive) and the empty sequence (negative). For BP, these samples are covered anyway by L*'s initial membership queries, but for email addresses the positive sample helps 'kick off' the extraction, preventing the method from accepting an automaton with a single (rejecting) state.

No further parameter tuning was required to achieve our results.

4.6.4 Small Regular Languages

The Tomita Grammars

The Tomita grammars (1982) are the following 7 languages over $\Sigma = \{0, 1\}$:

1. `1*`
2. `(10)*`
3. The complement of `((0|1)*0)*1(11)*(0(0|1)*1)*0(00)*(1(0|1)*)*`, i.e.: all sequences w which do not contain an odd series of 1s followed later by an odd series of 0s
4. All words w not containing `000`,
5. All w for which $\#_0(w)$ and $\#_1(w)$ are even (where $\#_a(w)$ is the number of a's in w),
6. All w for which $(\#_0(w) - \#_1(w)) \equiv_3 0$, and
7. `0*1*0*1*`.

They are the languages classically used to evaluate DFA extraction from RNNs.

We trained one 1-layer GRU network with hidden size 50 for each Tomita grammar (7 GRUs in total), in the manner described in Section 4.6.2. In training, all but one of the RNNs reached 3 consecutive epochs with 100% validation set accuracy within 10 epochs, and reached 100% test set accuracy. The 6th Tomita grammar was harder

to train, with the RNN reaching only 78% validation accuracy after 100 epochs. As our focus is on extraction rather than training, we repeated training on this language, eventually obtaining an RNN with perfect train and validation accuracy for this language as well (this time with initial learning rate 0.0004 and gamma 0.95). We then applied our method to extract from the perfectly trained RNNs.

For each one, our method correctly extracted and accepted the target grammar in under 1 second.

Random Small Regular Languages

Though the Tomita grammars are a popular language set for evaluating DFA extraction from RNNs, they are quite simple: the largest Tomita grammars are still only 5-state DFAs over a 2-letter alphabet. As our method performed so well on these grammars, we expand to more challenging languages.

We considered randomly-generated minimal DFAs of varying complexity, specifically, DFAs with alphabet size and number of states $(|\Sigma|, |Q|) = (3, 5), (5, 5)$ and $(3, 10)$. For each combination we randomly generated 10 minimal DFAs, making 30 DFAs overall. For each DFA we trained 6 2-layer RNNs: 3 GRUs and 3 LSTMs, each with hidden state sizes $d_s = 50, 100$ and 500, this makes 180 RNNs overall. The training method is described in Section 4.6.2. We applied our extraction method to each of these RNNs, with a time limit of 30 seconds (after which the last L* hypothesis is returned) and initial split depth and counterexamples as described in Section 4.6.3. The results of these experiments are shown in Table 4.1. Each row in the table represents the average of 10 extractions.

Most extractions completed before the time limit, having reached equivalence.[20] We compared the extracted automata against the networks on their training sets and on 1000 randomly generated word samples for each of the word-lengths 10,50,100 and 1000. In all settings (hidden size, alphabet size, and DFA size) where the RNNs achieved 100% test set accuracy, our extraction obtained DFAs with perfect accuracy against their RNNs. For two RNNs which reached 99% accuracy, our extraction achieved 99% accuracy against the RNNs, and for the two RNNs with less than 99% accuracy our extraction achieved on average $\geq 88\%$ accuracy for all evaluation sets.

4.6.5 Comparison with a-priori Quantisation

In [GMC+92], Giles and colleagues suggested partitioning the network state space by dividing each state dimension into q equal intervals, with q being the *quantisation level*. We tested this method on each of our small regular language RNNs (Section 4.6.4), with

[20] Though this is not necessarily a guarantee of true equivalence, it does generally indicate strong similarity.

Extraction from LSTM Networks — Our Method

						max	RNN	Average Extracted DFA Accuracy				
d_s	$\|\Sigma\|$	$\|Q_T\|$	Time (s)	$\|Q_\mathcal{A}\|$	#c-exs	|c-ex|	Acc.	l=10	l=50	l=100	l=1000	Train
50	3	5	6.82	10.8	2.5	12	1.0	1.0	1.0	1.0	1.0	1.0
100	3	5	4.09	5.0	2.0	10	1.0	1.0	1.0	1.0	1.0	1.0
500	3	5	10.03	5.0	1.8	10	1.0	1.0	1.0	1.0	1.0	1.0
50	5	5	16.66	19.9	3.0	8	0.99	0.99	0.99	0.99	0.99	0.99
100	5	5	12.53	6.6	2.4	12	1.0	1.0	1.0	1.0	1.0	1.0
500	5	5	18.34	5.0	2.3	8	1.0	1.0	1.0	1.0	1.0	1.0
50	3	10	30.97	67.8	5.2	9	0.91	0.92	0.88	0.88	0.88	0.89
100	3	10	21.15	23.4	4.6	18	0.99	1.0	0.99	0.99	0.99	0.99
500	3	10	16.27	10.0	4.0	9	1.0	1.0	1.0	1.0	1.0	1.0

Extraction from GRU Networks — Our Method

						max	RNN	Average Extracted DFA Accuracy				
d_s	$\|\Sigma\|$	$\|Q_T\|$	Time (s)	$\|Q_\mathcal{A}\|$	#c-exs	|c-ex|	Acc.	l=10	l=50	l=100	l=1000	Train
50	3	5	4.85	5.0	2.0	13	1.0	1.0	1.0	1.0	1.0	1.0
100	3	5	3.22	5.0	2.0	10	1.0	1.0	1.0	1.0	1.0	1.0
500	3	5	6.29	5.0	1.8	10	1.0	1.0	1.0	1.0	1.0	1.0
50	5	5	15.98	11.8	2.8	16	1.0	1.0	1.0	1.0	1.0	1.0
100	5	5	7.22	5.0	2.4	18	1.0	1.0	1.0	1.0	1.0	1.0
500	5	5	12.3	4.9	2.1	8	1.0	1.0	1.0	1.0	1.0	1.0
50	3	10	29.09	76.2	5.6	11	0.94	0.97	0.92	0.92	0.92	0.93
100	3	10	13.88	23.3	4.7	20	1.0	1.0	1.0	1.0	1.0	1.0
500	3	10	12.01	10.0	3.8	9	1.0	1.0	1.0	1.0	1.0	1.0

Table 4.1: Results for DFA extracted using our method from 2-layer GRU and LSTM networks with various state sizes, trained on random regular languages of varying sizes and alphabets. Each row in each table represents 10 experiments with the same parameters (network hidden-state size d_s, alphabet size $|\Sigma|$, and minimal target DFA size $|Q_T|$). In each experiment, a random DFA is generated and an RNN is trained on it, after which a DFA is extracted from and compared to the RNN. The column $|Q_\mathcal{A}|$ represents the size of the final returned DFA, *#c-exs* describes how many counterexamples were used during extraction, *max |c-ex|* describes their maximum length, and *RNN Acc.* is the accuracy of the trained RNN on its test set. Each column represents the average of the 10 experiments, except for *max |c-ex|* which gives the overall maximum counterexample used across all RNNs in that row. Each extraction was run with a time limit of 30 seconds, and whenever an extraction timed out the last automaton proposed by L* was taken as the extracted automaton. For the accuracies on the different lengths, 1000 random words of each length were sampled and evaluated, and for the accuracy on the training set all of the RNN's training set was evaluated (i.e., comparing DFA against RNN).

Extraction from LSTM Networks — Quantisation

d_s	$\|Q_T\|$	$\|\Sigma\|$	Time (s)	$\|Q_\mathcal{A}\|$	RNN Acc.	Extracted DFA Accuracy × Coverage l=10	l=50	l=100	l=1000	Train
50	5	3	100.17	16868	1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0
100	5	3	237.06	40088	1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0
500	5	3	469.64	60295	1.0	1.0×1.0	1.0×0.53	1.0×0.42	1.0×0.21	1.0×0.59
50	5	5	283.83	29472	0.99	0.99×1.0	0.99×1.0	0.99×1.0	0.99×1.0	0.99×1.0
100	5	5	469.88	47873	1.0	1.0×1.0	1.0×0.94	1.0×0.91	1.0×0.79	1.0×0.95
500	5	5	503.68	39508	1.0	1.0×0.03	-1×0.0	-1×0.0	-1×0.0	1.0×0.08
50	10	3	434.75	77538	0.91	0.98×1.0	0.94×0.58	0.97×0.44	0.94×0.31	0.97×0.65
100	10	3	500.62	83402	0.99	1.0×1.0	1.0×0.46	1.0×0.32	1.0×0.02	1.0×0.55
500	10	3	502.84	64720	1.0	1.0×1.0	-1×0.0	-1×0.0	-1×0.0	1.0×0.12

Extraction from GRU Networks — Quantisation

d_s	$\|Q_T\|$	$\|\Sigma\|$	Time (s)	$\|Q_\mathcal{A}\|$	RNN Acc.	Extracted DFA Accuracy × Coverage l=10	l=50	l=100	l=1000	Train
50	5	3	102.48	21359	1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0
100	5	3	239.93	49203	1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0
500	5	3	501.37	82933	1.0	1.0×1.0	1.0×0.22	1.0×0.13	1.0×0.0	1.0×0.35
50	5	5	335.37	42008	1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0	1.0×1.0
100	5	5	500.34	60089	1.0	1.0×0.98	1.0×0.77	1.0×0.67	1.0×0.41	1.0×0.8
500	5	5	502.31	49206	1.0	1.0×0.02	-1×0.0	-1×0.0	-1×0.0	1.0×0.08
50	10	3	500.42	100417	0.94	1.0×1.0	0.99×0.4	0.99×0.27	0.98×0.14	0.99×0.51
100	10	3	500.42	103488	1.0	1.0×1.0	1.0×0.51	1.0×0.34	1.0×0.06	1.0×0.58
500	10	3	501.93	82378	1.0	1.0×1.0	-1×0.0	-1×0.0	-1×0.0	1.0×0.12

Table 4.2: Results for DFA extracted using a simple partitioning of the RNN state space, in which each state dimension is split into $q = 2$ equal segments (positive and negative). The extractions were applied to the same RNNs as in Table 4.1, with each row representing 10 experiments as before. $|Q_\mathcal{A}|$ again reports the (average) number of states in the extracted DFAs, though this time it is rounded for clearer presentation. The extractions were run with a time limit of 500 seconds. This time, instead of reporting only the accuracy of the extracted DFAs against their RNNs on different samples sets, we also report their coverage: the fraction of samples for which the DFAs have a classification at all (i.e., do not have missing transitions). The accuracy is computed only on covered sequences, and we write report the accuracy as -1 when all extractions in the row have 0 coverage for that set. For example: 1.0×0.12 tells us that only 12% of samples have full transitions in the extracted DFA, but that for those 12%, the DFA accuracy against the RNN is perfect.

$q = 2$ and a time limit of 500 seconds to avoid excessive memory consumption.[21]

In many cases, we found that 500 seconds was not enough time for this method to extract a complete DFA from our RNNs.[22] To enable some comparison, we allow the method to return incomplete DFAs, i.e. DFAs in which some transitions are missing, and we move from evaluating just the accuracy of a DFA to evaluating both its accuracy and its *coverage*, with *coverage* being the fraction of samples for which it has a full transition path.

We provide the results of extracting with this method in Table 4.2, which uses the exact same RNNs as in Table 4.1.

The extracted DFAs are very large—with some even having 100,000 states–and yet their coverage of sequences of length 1,000 and even 100 tends to zero as the RNN complexity (state size d_s, or RNN target language complexity) increases. For the covered sequences, the extracted DFA's accuracy was often very high (99+%), suggesting that quantisation—while impractical—is sufficiently expressive to describe a network's state space. However, it is also possible that the sheer size of the quantisation (2^{50} for our smallest RNNs, and more for others) simply allowed each explored R-state its own A-state, giving high accuracy just by observation bias (only covered sequences could have their accuracy checked).

This is in contrast to our method, which always returns complete DFAs,[23] and which consistently extracted accurate DFA from the same networks in a fraction of the time and memory used by the plain quantisation approach. This is because our method maintains from a very early point in extraction a complete DFA $\mathcal{A}$ that constitutes a constantly improving approximation of the considered RNN.

4.6.6 Comparison with k-Means Clustering

Next, we implemented a simple k-means clustering and extraction approach and applied it to the same networks from Section 4.6.4 with varying k.

Specifically, for each RNN, we sampled $N = 5000$ unique prefixes from its train set, computed the states reached from them in the RNN, and used k-means clustering to partition the state space according to those states for each of $k = 1, 6, 11, ..., 31$.[24] We then mapped the transitions of each partitioning to create 7 potential DFAs, and evaluated each one against the RNN on its 1000-sample test set to choose the best.

[21] LSTMs have unbounded state space, which makes quantisation challenging. Specifically for $q = 2$ however, we just split each dimension along 0.

[22] This is because the quantisation method, even for the smallest possible q ($q = 2$), generates far more partitions than can be traversed within the time limit (q^{d_s} where d_s is the RNN state size, and $d_s \geq 50$ in our case). Note that this is in contrast to our method: our method only applies this quantisation method on d initial dimensions for user-defined d (typically ≤ 10), before continuing with only very gentle refinements as needed.

[23] Provided L* manages to generate at least one equivalence query before the time limit, which we observe to always happen in practice (usually taking $\leq$1 second).

[24] Using `sklearn.cluster.KMeans`.

Extraction from LSTM Networks — k-means Clustering

						RNN	Average Extracted DFA Accuracy				
d_s	$\|\Sigma\|$	$\|Q_T\|$	Time (s)	$\|Q_\mathcal{A}\|$	k	Acc.	l=10	l=50	l=100	l=1000	Train
50	3	5	48.93	4.5	25.5	1.0	0.85	0.85	0.85	0.85	0.85
100	3	5	69.85	3.9	20.0	1.0	0.82	0.81	0.81	0.81	0.81
500	3	5	274.96	5.0	18.5	1.0	0.87	0.85	0.86	0.85	0.86
50	5	5	53.13	3.3	18.5	0.99	0.8	0.8	0.8	0.79	0.8
100	5	5	84.48	4.3	18.0	1.0	0.83	0.83	0.83	0.82	0.83
500	5	5	289.63	5.6	24.0	1.0	0.98	0.97	0.98	0.97	0.98
50	3	10	52.74	6.4	19.5	0.91	0.67	0.66	0.65	0.67	0.67
100	3	10	63.06	12.0	27.5	0.99	0.8	0.75	0.75	0.74	0.76
500	3	10	250.99	11.6	28.0	1.0	0.93	0.89	0.89	0.89	0.9

Extraction from GRU Networks — k-means Clustering

						RNN	Average Extracted DFA Accuracy				
d_s	$\|\Sigma\|$	$\|Q_T\|$	Time (s)	$\|Q_\mathcal{A}\|$	k	Acc.	l=10	l=50	l=100	l=1000	Train
50	3	5	21.95	5.0	14.5	1.0	0.99	1.0	1.0	1.0	1.0
100	3	5	24.89	4.9	12.0	1.0	1.0	1.0	1.0	1.0	1.0
500	3	5	85.46	5.0	13.5	1.0	0.99	1.0	1.0	1.0	1.0
50	5	5	22.74	5.4	20.0	1.0	1.0	1.0	1.0	1.0	1.0
100	5	5	29.13	5.0	18.5	1.0	1.0	1.0	1.0	1.0	1.0
500	5	5	91.68	5.1	19.0	1.0	1.0	1.0	1.0	1.0	1.0
50	3	10	27.98	12.4	28.5	0.94	0.87	0.84	0.84	0.83	0.85
100	3	10	27.15	10.5	31.0	1.0	0.96	0.94	0.94	0.94	0.94
500	3	10	92.63	10.0	28.5	1.0	0.99	0.98	0.99	0.98	0.99

Table 4.3: Results for DFA extracted using k-means clustering from the same 2-layer GRU and LSTM networks considered in Table 4.1, i.e., each row represents the average results of 10 experiments as before, and considers the exact same trained RNNs. The extractions did not have a time limit, instead, the number of states sampled was set to 5000 and the k values considered were $k = 1, 6, 11, ..., 31$. The accuracies were evaluated on the same sample sets as in Table 4.1.

k-means has a well defined and 'reasonably quick' stopping condition: the number of RNN states visited, and the number of clusters to be created and traversed from them, is given as input to the extraction.[25] Hence for this extraction we do not use a time limit, allowing the method to extract all of its potential DFAs in full, evaluate them, and return the best DFA. As done for the other methods, we measure for k-means the total time from beginning the extraction until a single final DFA is returned. In particular, this covers sampling once all 5000 RNN states (generally <10 seconds), making a k-state DFA from these RNN states by applying k-means clustering to them (taking from <1 to ∼50 seconds for each k, depending on the states and on k), and finally choosing the best DFA by evaluating on the test set (generally <10 seconds). We note that the bulk of the extraction time is spent in clustering the sampled states into different numbers of clusters k.

In Table 4.3 we report the results of these extractions. In particular, we report the

[25]This is in contrast to our method, which may continue to refine its hypothesis indefinitely without ever reaching equivalence (consider for example an RNN that has learned a non-regular language), or quantisation, which creates so many partitions in modern-sized architectures (2^{50} even for our smallest networks and quantisation level) that it cannot be used without adding some time or size limit.

time (in seconds) spent on each full extraction, the number of clusters k used for each best DFA, each DFA's size $|Q_{\mathcal{A}}|$ after minimisation, and of course each extracted DFA's accuracy against the same sample sets as before (i.e., as in 4.1).

For the GRU networks trained on smaller DFAs (which reached 100% test-set accuracy), k-means clustering is as successful as our method, often returning a DFA with perfect or near-perfect accuracy against the target RNN. For the LSTMs and the larger DFAs however, our method obtains far higher accuracy, and often in less time. The difference in success on the LSTMs and GRUs is curious, we leave this question open in this work.

4.6.7 Comparison with Random Sampling For Counterexample Generation

For 3 of the Tomita grammars (specifically, Tomitas 3,4, and 7), the first counterexample returned in our extraction (Section 4.6.4) was actually created by the initial random sampling. Moreover, for all of the Tomita grammars, answering all equivalence queries using a random sampler alone (with up to $1,000$ samples per query) was successful at extracting the grammars from the RNNs, and this was also true for many of the languages considered in Section 4.6.4. The termination is slightly slower than our own, to allow for sampling many potential counterexamples before accepting the L* hypothesis, but still fast enough to make random sampling seem appealing (the method spent ≈ 10 seconds on each Tomita grammar). Indeed, [MY18] even suggest such a method in their recent work, analysing it from a PAC perspective.

Given this, the question may arise whether there is at all merit to the exploration and refinement of abstractions of the network, as opposed to a simple random sampling approach to counterexample generation for L* equivalence queries.

In this section we show the advantage of our method for counterexample generation, through the example of balanced parentheses (BP): the language of sequences with correctly balanced parentheses over the alphabet `()a-z`. BP is not a regular language, but the attempt to approximate it with DFAs, and in particular the search for counterexamples to proposed DFAs, proves informative. In particular, when sampling the tokens with uniform distribution, the probability of randomly generating a sequence with nested and correctly balanced parentheses over the BP alphabet is very low. This prevents the random sampler from finding counterexamples to L*'s proposed automata, each of which accept balanced parentheses to a bounded depth (see Examples in Figure 4.1), highlighting the advantage of our approach.

We train one GRU and one LSTM network on BP, each with 2 layers and hidden dimension 50. We extract DFAs from these networks using L*, generating counterexamples once with our method and once with a random counterexample generator. The random counterexample generator works as follows: for each equivalence query, it randomly samples sequences over the input alphabet Σ until a counterexample (sample on

	Accuracy on Train Set		Max Nesting Depth			
Network	Our Method	Random	Our Method	Random	d_s	#Layers
GRU	99.98	87.12	8	2	50	2
LSTM	99.98	94.19	8	3	50	2

Table 4.4: Accuracy of extracted automata against their networks, which were trained to 100% training accuracy on the balanced parentheses (BP) language. The comparisons were done on the training sets of the networks. The maximum nesting depth the extracted automata reached while still behaving as BP is recorded (the GRU network ultimately returned a more complex automaton than the one extracted from the LSTM network, but this automaton no longer behaved as BP and so we have no reasonable measure for its 'depth'). The hidden size d_s and the number of layers in each network is also noted. (For the LSTM network, this is the size of both the memory and the cell vectors, meaning the total hidden size of a single cell in this network is twice as big as the value listed.)

which $\mathcal{A}$ and the RNN disagree) is found. In particular, for each length $l = 1, 2, 3, ...$ and increasing until a counterexample is found, it generates and compares up to 1000 random samples of length l, with uniform distribution.

We allowed each method 400 seconds[26] to extract an automaton from networks trained to 100% train set accuracy. The accuracy of these extracted automata against the original networks on their training sets is recorded in Table 4.4, as well as the maximum parentheses nesting depth the L* proposed automata reached during extraction.

We list the counterexamples and counterexample generation times for each of the BP network extractions in Table 4.5. Note the succinctness and the generation speed of the counterexamples generated by our method: excluding two samples at the end of the GRU extraction, they are clear of the 'neutral' tokens `a-z` and of repeating parentheses (e.g., `()()`), as these were not necessary to advance the automata learned by L* (Figure 4.1). In contrast, the random sampling method has difficulty finding legally balanced sequences, taking a long time to find counterexamples at all, and including many 'uninformative' neutral tokens in its results.

The extracted DFAs themselves were also pleasing: each subsequent DFA proposed by L* for this language was capable of accepting all words with balanced parentheses of increasing nesting depth, as pushed by the counterexamples provided by our method (Figure 4.1). In addition, for the GRU network trained on BP, our extraction method managed to push past the limits of the network's 'understanding'—finding the point at which the network begins to overfit to the particularly deeply-nested examples in its training set, and extracting the slightly more complicated automaton seen in Figure 4.2.

[26]Timed using the `clock()` method from python's `time` module.

Refinement-based vs. Brute-Force Counterexample Generation on the Balanced Parentheses Language

GRU

Refinement Based		Brute Force	
Counterexample	Time (seconds)	Counterexample	Time (seconds)
))	1.1	))	0.4
(())	1.2	(()i)ma	32.6
((()))	2.1		
(((())))	3.1		
((((()))))	3.8		
((((()))))	4.4		
((((((()))))))	6.6		
(((((((())))))))	9.2		
(((((((v)))))))	10.7		
(((((((a()z)))))))	8.3		

LSTM

Refinement Based		Brute Force	
Counterexample	Time (seconds)	Counterexample	Time (seconds)
))	1.4	))	1.5
(())	1.6	tg(gu()uh)	57.5
((()))	3.1	((wviw(iac)r)mrsnqqb)iew	231.5
(((())))	3.1		
((((()))))	3.4		
((((()))))	4.7		
((((((()))))))	6.3		
(((((((())))))))	9.2		
((((((((())))))))	14.0		

Table 4.5: Extraction of automata from GRU and LSTM networks trained to 100% accuracy on the training set for the language of balanced parentheses over the 28-letter alphabet `a-z,(,)`. Each table shows the counterexamples and the counterexample generation times for each of the successive equivalence queries posed by L^* during extraction, for both our method and a brute force approach. Generally, each successive equivalence query from L^* for either network was an automaton classifying the language of all words with balanced parentheses up to nesting depth n, with increasing n. The exception to this comes after the penultimate counterexample in the extraction from the GRU network, in which a word with unbalanced parentheses was returned as a counterexample to L^* (whose automaton currently rejects it).

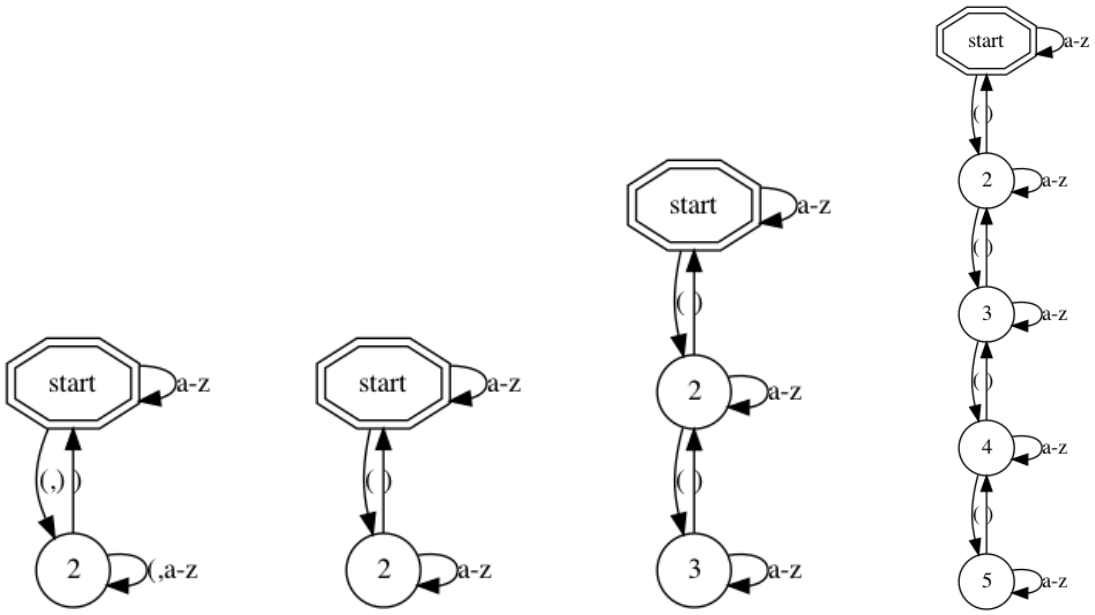


Figure 4.1: Select automata of increasing size for recognising balanced parentheses over the 28 letter alphabet `a-z,(,)`, up to nesting depths 1 (flawed), 1 (correct), 2, and 4, respectively. In this and in all following automata figures, the initial state is an octogon, accepting states have a double border, and sink reject states (rejecting states whose transitions all lead back to themselves) are not drawn.

4.6.8 Additional variations on our method

We show the necessity of the initial split and counterexamples for our method, the effect of running extraction for a longer time (if it has not completed), and support the decision to return the final L* hypothesis $\mathcal{A}$ as opposed to the final abstraction $A_{R,p}$ whenever the extraction has not reached equivalence in time.

Removing the Initial Split Heuristics We run the extraction again on the same RNNs as in Table 4.1, but this time setting the initial split depth to 1 and the number of random samples before accepting a hypothesis to 0. We report the results in Table 4.6. The average number of counterexamples ("#c-exs") per extraction drops to almost 0 for most settings, meaning the majority L* initial hypotheses are accepted immediately by the method (without counterexamples). The number of states in the returned automata is often smaller than in the target, and their accuracy drops significantly.

This shows that indeed our method must be coupled with some heuristics to prevent acceptance in the early stages, during which both the abstraction and the L* hypothesis only reflect the RNN's classification on very short sequences, and have not yet diverged.

Timing out: Using the Abstraction, and Increasing the Time Limit When we increase $|\Sigma|$ and $|Q|$ of our randomly generated target DFAs to 10, the training routine used in this work is not sufficient for the RNNs with dimensions $d_s = 50$ and $d_s = 100$ to train perfectly, and they reach on average $< 0.8\%$ test set accuracy on

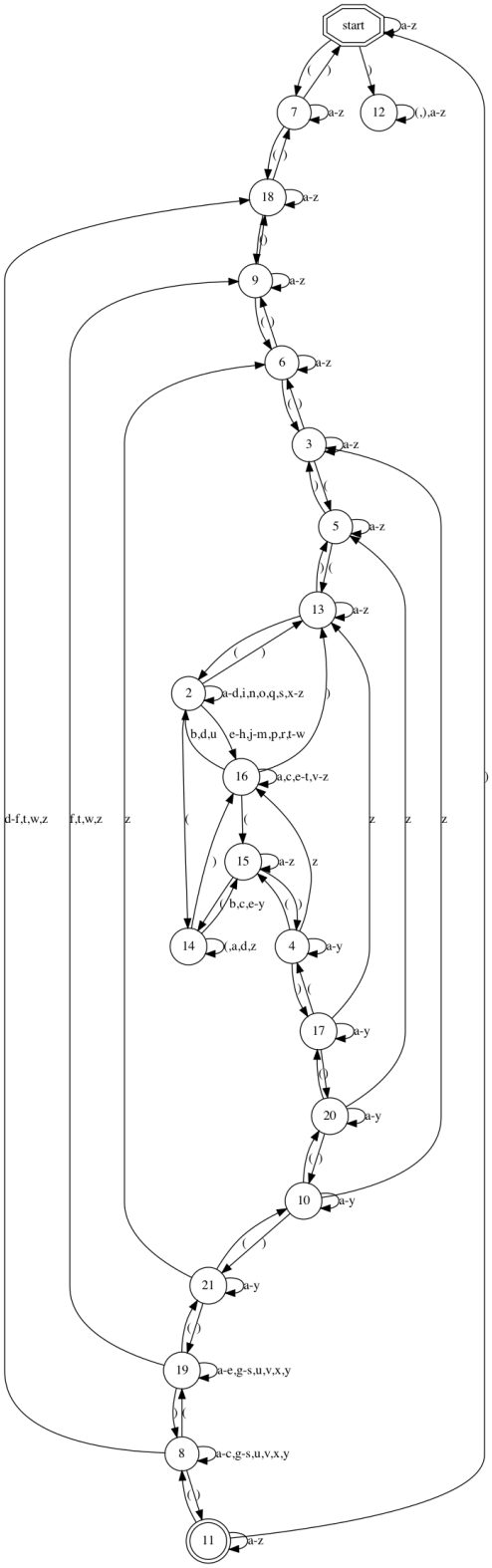


Figure 4.2: Automaton with vague resemblance to the BP automata of Figure 4.1, but no longer representing a language of balanced parentheses up to a certain depth. (Showing how a trained network may be overfitted past a certain sample complexity.)

Extraction from LSTM Networks — Our Method (No Heuristics)

						max	RNN	Average Extracted DFA Accuracy				
d_s	$\|\Sigma\|$	$\|Q_T\|$	Time (s)	$\|Q_\mathcal{A}\|$	#c-exs	|c-ex|	Acc.	l=10	l=50	l=100	l=1000	Train
50	3	5	0.26	2.2	0.2	3	1.0	0.63	0.64	0.63	0.63	0.64
100	3	5	0.21	2.1	0.1	3	1.0	0.64	0.65	0.63	0.64	0.64
500	3	5	0.23	2.1	0.1	3	1.0	0.64	0.65	0.63	0.64	0.64
50	5	5	0.25	1.8	0.0	-	0.99	0.74	0.74	0.74	0.74	0.75
100	5	5	0.2	1.8	0.0	-	1.0	0.74	0.74	0.74	0.74	0.75
500	5	5	0.29	1.8	0.0	-	1.0	0.74	0.74	0.74	0.74	0.75
50	3	10	9.52	15.8	1.6	8	0.91	0.66	0.65	0.65	0.65	0.66
100	3	10	0.65	2.4	0.3	5	0.99	0.58	0.56	0.57	0.57	0.58
500	3	10	0.16	1.7	0.0	-	1.0	0.55	0.54	0.54	0.55	0.55

Extraction from GRU Networks — Our Method (No Heuristics)

						max	RNN	Average Extracted DFA Accuracy				
d_s	$\|\Sigma\|$	$\|Q_T\|$	Time (s)	$\|Q_\mathcal{A}\|$	#c-exs	|c-ex|	Acc.	l=10	l=50	l=100	l=1000	Train
50	3	5	0.16	2.1	0.1	3	1.0	0.64	0.65	0.63	0.64	0.64
100	3	5	0.16	2.0	0.0	-	1.0	0.66	0.67	0.66	0.66	0.67
500	3	5	0.2	2.0	0.0	-	1.0	0.66	0.67	0.66	0.66	0.67
50	5	5	0.19	1.8	0.0	-	1.0	0.74	0.74	0.74	0.74	0.75
100	5	5	0.18	1.8	0.0	-	1.0	0.74	0.74	0.74	0.74	0.75
500	5	5	0.21	1.8	0.0	-	1.0	0.74	0.74	0.74	0.74	0.75
50	3	10	0.13	1.7	0.0	-	0.94	0.55	0.55	0.54	0.55	0.56
100	3	10	0.16	1.7	0.0	-	1.0	0.55	0.54	0.54	0.55	0.55
500	3	10	0.31	2.5	0.3	7	1.0	0.59	0.59	0.59	0.59	0.6

Table 4.6: Extracting with our method from the same RNNs as in Table 4.1, but this time without the initial heuristics as described in Section 4.6.3. The extraction time is reduced significantly, along with the accuracy: L*'s first hypotheses are frequently very small, and without the aggressive initial state-splitting and random samples, the abstraction is too coarse to find counterexamples.

their target languages. For these RNNs, we observe that our extraction method does not reach equivalence in the provided time. In particular, the L* hypotheses grow very large, and the extraction often times out while increasing the *observation table*: the internal table of sequence labels maintained by L* between equivalence queries (i.e., the majority time is spent on refining $\mathcal{A}$ after each new counterexample).

In all of our experiments, whenever we run out of time, we return the last L* hypothesis $\mathcal{A}$ as the extracted automaton. In this section, we check how much this hypothesis improves as we increase the time limit, and evaluate the option of returning the last abstraction $A_{R,p}$ used by our method instead.

Table 4.7 shows a set of extractions from imperfectly trained RNNs, trained with the same training routine and number of repetitions as before. We make 10 DFAs all with $|Q| = |\Sigma| = 10$ and on each DFA train 4 2-layer RNNs: 2 GRUs and 2 LSTMs, each with hidden state sizes $d_s = 50$ and $d_s = 100$. We then extract from each RNN with 5 different time limits ranging from 50 to 1000 seconds. This means that overall Table 4.7 shows results for 10 DFAs, 40 RNNs, and 200 extractions (each row represents 10 extractions).[27]

Alongside the details of the last L* hypothesis $\mathcal{A}$, we also report the size of our final

[27]We ran each extraction in itself, for example each RNN's 1000 second extraction was not merely a continuation of its 50 second extraction but a full extraction in its own.

Extraction from LSTM Networks — Our Method (Time Limits)

d_s	Time Limit	$\lvert Q_{\mathcal{A}}\rvert$	$\lvert Q_{R,p}\rvert$	$\lvert p\rvert$	#c-exs	RNN Acc.	$\mathcal{A}$ Accuracy			$A_{R,p}$ Accuracy		
							l=10	l=1000	Train	l=10	l=1000	Train
50	50	116	157	1029	5.9	0.74	0.84	0.85	0.85	0.66	0.66	0.66
	100	178	163	1031	6.8	0.74	0.86	0.86	0.87	0.66	0.67	0.66
	200	300	160	1031	7.5	0.74	0.86	0.86	0.87	0.66	0.67	0.67
	500	466	162	1031	8.0	0.74	0.88	0.88	0.88	0.65	0.66	0.66
	1000	810	162	1032	9.1	0.74	0.89	0.9	0.9	0.65	0.66	0.66
100	50	113	304	1029	5.0	0.78	0.77	0.77	0.77	0.62	0.62	0.62
	100	200	307	1031	6.0	0.78	0.79	0.79	0.79	0.61	0.62	0.62
	200	313	305	1029	6.6	0.78	0.8	0.8	0.81	0.61	0.62	0.62
	500	532	310	1032	7.4	0.78	0.81	0.81	0.81	0.61	0.62	0.62
	1000	728	309	1032	7.6	0.78	0.81	0.82	0.83	0.61	0.62	0.62

Extraction from GRU Networks — Our Method (Time Limits)

d_s	Time Limit	$\lvert Q_{\mathcal{A}}\rvert$	$\lvert Q_{R,p}\rvert$	$\lvert p\rvert$	#c-exs	RNN Acc.	$\mathcal{A}$ Accuracy			$A_{R,p}$ Accuracy		
							l=10	l=1000	Train	l=10	l=1000	Train
50	50	132	349	1031	6.3	0.75	0.86	0.86	0.86	0.68	0.69	0.7
	100	210	348	1031	6.9	0.75	0.86	0.86	0.86	0.68	0.69	0.69
	200	352	348	1030	7.6	0.75	0.87	0.87	0.87	0.68	0.69	0.69
	500	540	349	1032	8.8	0.75	0.89	0.89	0.89	0.68	0.7	0.69
	1000	830	353	1034	9.7	0.75	0.89	0.89	0.9	0.68	0.7	0.7
100	50	141	508	1031	5.0	0.79	0.76	0.76	0.77	0.61	0.63	0.62
	100	174	506	1031	5.6	0.79	0.77	0.78	0.78	0.62	0.63	0.63
	200	306	508	1030	6.5	0.79	0.78	0.78	0.79	0.61	0.62	0.63
	500	567	522	1035	7.4	0.79	0.81	0.81	0.82	0.62	0.63	0.63
	1000	780	517	1033	7.5	0.79	0.81	0.81	0.82	0.61	0.62	0.62

Table 4.7: Extracting with our method from 2-layer GRUs and LSTMs trained imperfectly on DFAs with size $|\Sigma| = |Q| = 10$, varying RNN hidden size (d_s) and extraction time limit. Each row represents the average of 10 experiments, with average DFA ($|Q_{\mathcal{A}}|$, $|Q_{A_{R,p}}|$) and final partitioning ($|p|$) sizes rounded for space. We report both the accuracy (against the RNN) of the final L^* hypothesis, $\mathcal{A}$, and the abstraction $A_{R,p}$ used by the method to find counterexamples to each $\mathcal{A}$. We see that the final L^* hypothesis is clearly the superior option when extraction has not terminated. Unfortunately, we also see that the accuracy does not increase well with more time, this is because the hypothesis generation (time from counterexample to new hypothesis) grows slower with each iteration.

partitioning p (i.e., number of partitions it divides the state space into), the size (after minimisation) of the abstraction $A_{R,p}$ it defines, and the accuracy of $A_{R,p}$ against its target RNN.

The results show clearly that the L* hypothesis is the preferable choice when the extraction does not complete. Effectively, the partitioning p and abstraction $A_{R,p}$ it defines act as a tool for refining the L* hypotheses, and not so much the other way around.[28]

The results also show that, for these non-terminating extractions, it is 'difficult' to improve beyond the automata reached in the early stages: increasing the extraction time to 100, 200, and even 1000 seconds gives only a small increase in accuracy each time. We also see that the number of counterexamples used per extraction grows very slowly with the increase in time, i.e., more time does not significantly increase the number of hypotheses presented by L* .

Analysing the time spent by the extraction reveals that L* gets 'stuck' refining the large hypotheses it creates, generating many membership queries without reaching new equivalence queries. The average equivalence query time across all experiments is <1.5s, whereas the maximum hypothesis refinement time in each experiment grew to over 10, 48, 60, 170 and 314 seconds for each of the time limits respectively.[29] A more efficient implementation of L* , or possibly an approximation of it, would be an important step towards scaling this method.

4.6.9 Discussion

Adversarial Inputs

Balanced Parentheses Excitingly, the penultimate counterexample returned by our method during the extraction of balanced parentheses (BP) in Section 4.6.7 is an *adversarial input*: a sequence with unbalanced parentheses that the network accepts (despite its target language accepting only sequences with balanced parentheses). This input is found in spite of the network's seemingly perfect behaviour on its set of 44000+ training samples. Note that the random sampler did not manage to find such samples.

Inspecting the extracted automata indeed reveals an almost-but-not-quite correct DFA for the BP language (Figure 4.2). The RNN overfit to random peculiarities in

[28]This may be because, whenever the equivalence checking finds a disagreement, it first checks for the possibility of a counterexample to L* before checking whether the abstraction needs to be refined. However, the difference may also come through the more long-sighted nature of L* : internally, L* maintains a growing list of prefixes and suffixes, all combinations of which its hypotheses have to classify correctly. In contrast, traversing a partitioning of the state space only looks as far as the immediate classification and transitions of each visited partition. L*'s advantage here is also its curse: learning a DFA with L* has polynomial time complexity in the size of the DFA, whereas traversing a partitioning is linear in the number of partitions.

[29]I.e., for example, each one of the 1000 second extractions spent at least 314 seconds on at least one hypothesis refinement.

the training data and did not learn the intended language, and our extraction method managed to discover and highlight an example of this 'incorrect' behaviour.

Email Addresses For a seemingly perfect LSTM-acceptor trained on the regular expression

[a-z][a-z0-9]*@[a-z0-9]+.(com|net|co.[a-z][a-z])$

(simple email addresses over the 38 letter alphabet {a-z,0-9,@,.}) to 100% accuracy on a 40,000 sample train set and a 2,000 sample test set, our method quickly returned the counterexamples seen in Table 4.8, showing clearly words that the network misclassified (e.g., 25.net). We ran extraction on this network for 400 seconds, and while we could not extract a representative DFA in this time,[30] our method did show that the network learned a far more elaborate (and incorrect) function than needed. In contrast, given a 400 second overall time limit, the random sampler did not find any counterexample beyond the provided one.

We note that our implementation of k-means clustering and extraction had no success with this network, returning a completely rejecting automaton (representing the empty language), despite trying k values of up to 100 and using all of the network states reached using a train set with a 50:50 ratio between positive and negative samples.

Beyond demonstrating the capabilities of our method, these results also highlight the brittleness in generalisation of trained RNNs, and suggest that evidence based on test-set performance should be interpreted with extreme caution. This reverberates the results of ([GS16]), who trained a neural architecture based on a multi-layer LSTM to mimic a finite state transducer (FST) for number normalisation. They showed that the RNN-based network, trained on 22M samples and validated on a 2.2M sample development set to 0% error on both, still had occasional errors (though with error rate < 0.0001) when applied to a 240,000 sample blind test set.

Limitations and Discussion

L* Optimisation One limitation of the method shown in this work is the polynomial time complexity of L*, which becomes a significant issue as the extracted DFA grows (see Section 4.6.8, *Timing out*). Applying our method with more efficient variants of L* , such as the TTT algorithm presented by [IHS14], may yield better results.

L* and Noise Whenever applied to an RNN that has failed to generalise properly to its target language, our method soon finds several adversarial inputs, and begins to build very large DFAs. As noted above, to L*'s polynomial complexity and intolerance to noise, this quickly becomes extremely slow.[31]

[30] A 134-state DFA $\mathcal{A}$ was proposed by L* after 178 seconds, and the next refinement to $\mathcal{A}$ (initiated 4.43 seconds later) timed out. The accuracy of the 134-state DFA on the train set was nearly random. We suspect that the network learned such a complicated behaviour that it simply could not be represented by any small DFA.

[31] This happened for example to our balanced-parentheses LSTM network, which timed out during

Counter-example	Time (s)	Network Classification	Target Classification
0@m.com	provided	√	√
@@y.net	2.93	×	×
25.net	1.60	√	×
5x.nem	2.34	√	×
0ch.nom	8.01	×	×
9s.not	3.29	×	×
2hs.net	3.56	√	×
@cp.net	4.43	×	×

Table 4.8: Counterexamples generated during extraction from an LSTM email-address network with 100% train and test accuracy. Examples of the network deviating from its target language are shown in bold.

Of course by the nature of L*, any complexity in the final returned automaton is only a result of the inherent complexity of the RNN's learned behaviour, and so we may say that this result is not necessarily incorrect. Nevertheless, it limits us, and seeking a way to recognise and overcome 'noise' in the given network's behaviour is an interesting avenue for future work.

Adversarial Inputs On the bright side, this same limitation does demonstrate the ease with which our method identifies imperfectly trained networks. These cases are annoyingly frequent: for many RNN-acceptors with 100% train and test accuracy on large test sets, our method was able to find many simple misclassified examples (Section 4.6.9).

Note on Heuristics In Section 4.2, we note that existing works consider multiple RNNs, and then must choose the best according to a heuristic. Our method can also be seen as considering multiple DFAs and abstractions, with the equivalence query being the 'heuristic' deciding whether to terminate or consider more DFAs/abstractions. We highlight here our differences. First, in our method, the DFAs considered are always minimal (thanks to L*), and the abstractions used can be much smaller than in other methods. In particular the abstractions can be small because they are dynamically refined by the method on an as-needed basis, and so can afford to be very coarse: 'missed partitions' are discovered and fixed automatically by the method. Secondly, even when the refinement eventually creates a very large abstraction, the equivalence query is applied 'on-the-fly', meaning it can cut off and return counterexamples/refine the abstraction even before $A_{R,p}$ has been fully mapped.

4.7 Learning from Only Positive Samples

Thus far, the method presented here can be used to learn a DFA from a set of positive and negative samples: we train an RNN-acceptor to generalise from them, and then

L* refinement after the last counterexample.

extract a DFA from it.

However, we can also use our method to learn a DFA from positive samples only, by training an RNN using a language-modelling objective, and then extracting from an RNN-acceptor interpretation of it. Such RNNs are trained only on positive samples, attempting to model their distribution rather than classify what is or isn't in the language:

A *language-model RNN (LM-RNN)* over an alphabet Σ and end-of-sequence symbol $\$ \notin \Sigma$ is an RNN with classification component $f_R : S_R \to [0,1]^{\Sigma \cup \{\$\}}$ defining for every RNN-state a distribution over $\Sigma \cup \{\$\}$. An LM-RNN effectively defines for every sequence $w \in \Sigma^*$ and token $\sigma \in \Sigma \cup \{\$\}$ the probability of sampling σ after seeing w: $P(\sigma|w) = f_R(\hat{g_R}(w))(\sigma)$.

LM-RNNs can be interpreted as classifiers by taking a threshold t and defining that they accept exactly the set of sequences $w = w_1 w_2 ... w_n \in \Sigma^*$ which satisfy: 1. $P(\$|w) \geq t$, and 2. for every strict prefix $w' = w_1 w_2 ... w_i$, $i < n$ of w, $P(w_{i+1}|w') \geq t$. This interpretation recently appears as *locally ϵ-truncated support* in the work of [HHG+20], with $\epsilon = t$.

LM-RNNs can therefore be adapted for extraction as classifiers by defining each of their states as accepting or rejecting according to the probability they assign to \$, and introducing an artificial sink-reject state v[32] that is entered whenever a sequence transitions through a token with too low probability. Formally:

Making an RNN acceptor Let R be an LM-RNN with reachable state space $S \subsetneq \mathbb{R}^{d_s}$, initial state $h_{0,R} \in S$, update function g_R, and classification function f_R. Let $t \in [0,1]$ be a threshold and let $v \in \mathbb{R}^{d_s} \setminus S$ be a vector that cannot be reached in R from any input sequence.[33] To create an RNN-acceptor R' from R, we build the components $h'_{0,R} = h_{0,R}$, $f'_R(s) = \begin{cases} Acc & : f_R(s)(\$) \geq t \\ Rej & : \text{else} \end{cases}$, and $g'_R(s,\sigma) = \begin{cases} v & : f_R(s)(\sigma) < t \text{ or } s{=}v \\ g_R(s,\sigma) & : \text{else} \end{cases}$.

The new RNN-acceptor R' can now be passed directly to our algorithm for extraction.

When the language is 'small'—in the sense that uniformly sampled sequences are likely to be rejected—sampling sequences according to the RNN's distribution is likely to hit a sample that has not yet been considered by L*. Hence here random sampling according to the RNN's distribution can be a useful augmentation to the equivalence query—though this can also create overly long counterexamples (Section 4.7.1).

This approach—training an LM-RNN, adapting it as a classifier, and then extracting from it with the method presented in this work—has been recently applied by [YW21] to elicit a sequence of DFAs from trained LM-RNNs, as part of a process for learning context free grammars from trained RNNs.

[32] I.e., an externally maintained state $v \notin S_R$.

[33] For most RNN architectures, finding such a vector v is easy from the architecture definition. For instance, for LSTMs and GRUs, $v = \bar{2}$ is sufficient: both have at least some of their state dimensions bound to the range $[-1,1]$.

Note Extracting from LM-RNNs requires some hyperparameter tuning, as changing the threshold t changes the set of sequences accepted by R'.

4.7.1 Proof of Concept

We provide a small number of example extractions from LM-RNNs trained on non-regular languages, observing the ability of the method to generate increasingly 'complex' DFA approximations of the targets. More examples are also present in [YW21].

$a^n b^n$

We train a 2-layer LSTM-based LM-RNN with hidden dimension $d_s = 50$ on positive samples from the language $a^n b^n = \{a^i b^i \mid i \in \mathbb{N}\}$.[34] We then interpret it as an RNN-acceptor as described above, and extract from it using our extraction method, with $t = 0.1$ and a time limit of 400 seconds.

As expected, the extraction generates a series of DFA approximations of the non-regular target language, we present some of these in Figure 4.3. The extraction ultimately reached DFAs approximating $a^n b^n$ up to $n \leq 20$ before timing out, with the majority of time spent on refining the L* hypotheses, which grew slower as the DFA grew: the final hypotheses returned by L* took 46, 54, and 63 seconds each to generate after their 'prompting' counterexamples, and the next L* refinement after them also timed out after 53 seconds (meanwhile, each of the counterexamples took < 5 seconds to generate). This result suggests that this method may benefit from applying a more efficient implementation of L*, such as the TTT algorithm of [IHS14].

Dyck-3

We consider the language Dyck-3 with 3 additional neutral tokens, i.e.: correctly balanced sequences over the alphabet `{}()[]abc`. For example, `{}a(b[])c` is in the language, but `([)]` and `())` are not.

We use a 2-layer GRU with dimension 50, and train it as a language model on 50000 non-unique samples of lengths 1-100 from Dyck-3 for 20 epochs, reaching a train, test, and validation cross-entropy loss of ≈1.7. We interpret the GRU as a classifier using rejection threshold $t = 0.01$, and extract from it using our method with a time limit of 400 seconds and initial split depth $d = 10$.[35]

The abstraction-based equivalence query provides L* with counterexamples teaching it new 'parantheses nestings' one at a time,[36] creating in 128 seconds the Dyck-3

[34] 20 epochs on 5000 non-unique samples of average length 50.

[35] We also augmented the equivalence queries with random counterexample generation using LM-sampling, to be considered before accepting any DFA. However, this was never used: our abstraction-based method rejected every hypothesis before reaching this stage.

[36] The counterexamples are: 1. `()` 2. `{}` 3. `[}` 4. `(())` 5. `({})` 6. `([])` 7. `{()}` 8. `{{}}` 9. `{[]}` 10. `[{}]` 11. `((()))` 12. `(({}))` 13. `({[]})` 14. `([{}])` 15. `{([])}` 16. `{{[]}}` 17. `{[[]]}` 18. `[(){}]` 19. `([()])` 20. `[(())]` 21. `[[]()]` 22. `(([]))` 23. `([[]])` 24. `[([])]` 25. `[[()]]` 26. `[[{}]]` 27. `[[[]]]` 28. `(((())))`

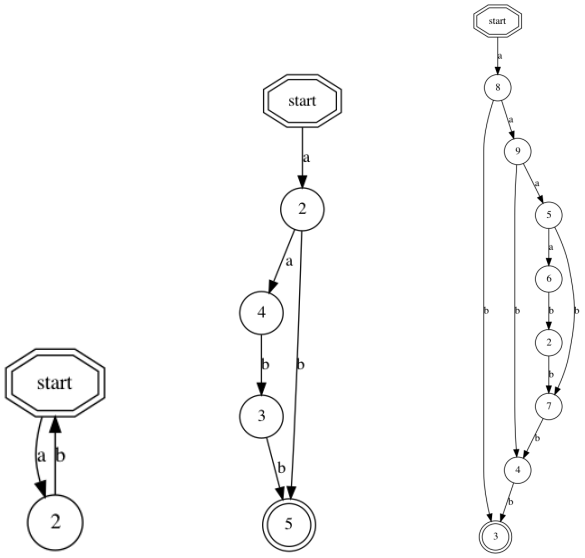


Figure 4.3: Automata approximating the language $a^n b^n$ up to different lengths, extracted from an RNN trained on only positive examples. The extraction created 'correct' approximations up to $n = 20$ before reaching the time limit.

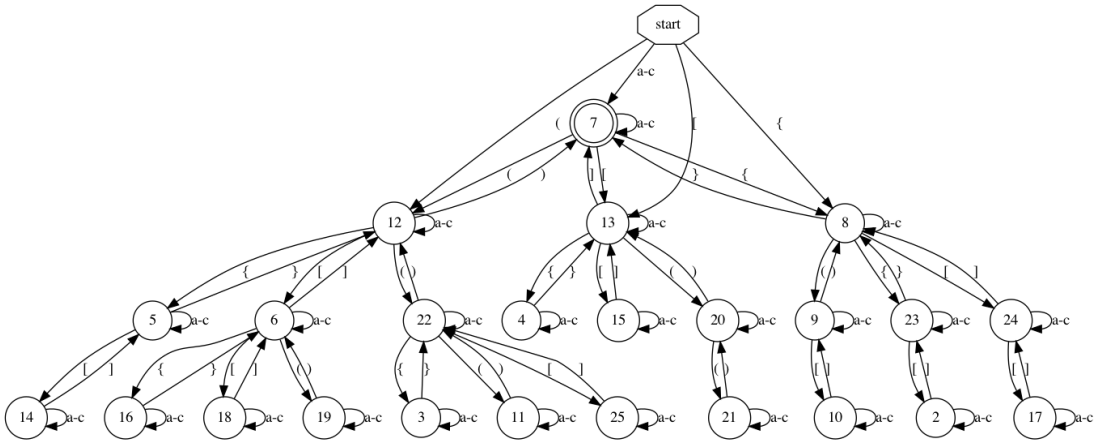


Figure 4.4: An automaton approximating the language Dyck-3 with neutral tokens `a-c`, obtained in 128 seconds as the 24^{th} hypothesis during extraction from a GRU trained on only positive samples from the language. The automaton correctly recognises many (but not all) correct parenthesis nestings up to depth $n = 3$, for example, it accepts the sequence `{([])}()` but not the sequence `({()})`. It rejects the empty sequence, this is an artefact of the RNN's behaviour.

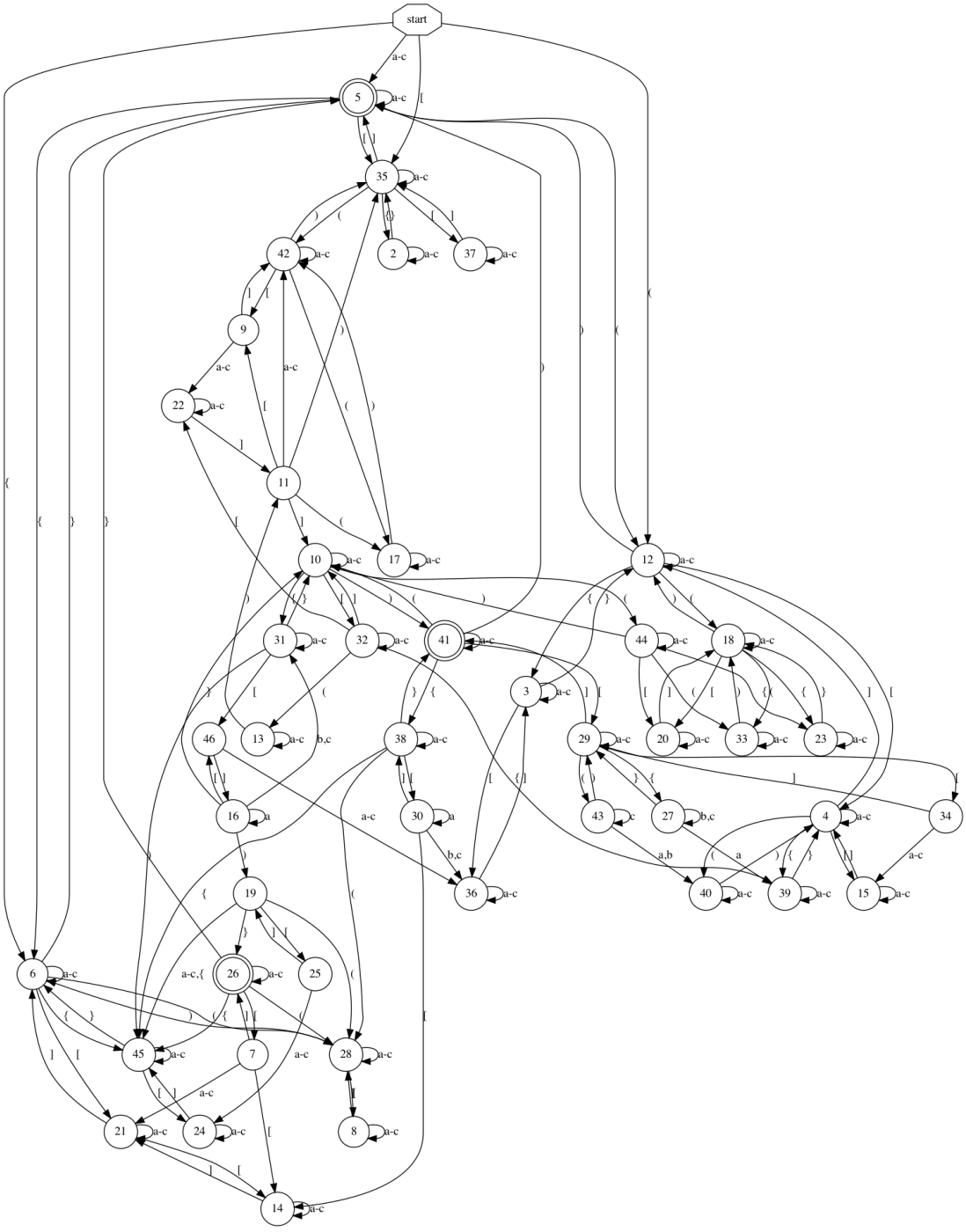


Figure 4.5: The next hypothesis presented by L* after receiving the counterexample `[([])]` to the DFA shown in 4.4, while extracting from our LM-GRU trained on Dyck-3. While the previous hypotheses reflected clear (regular) subsets of Dyck-3 with bounded depth, now L* has found several 'irregularities' in the RNN, and encoded them into a new hypothesis which is much larger and more complicated than those before it.

approximation $\mathcal{A}_{24}$ shown in Figure 4.4 (the 24$^{\text{th}}$ hypothesis created during the extraction). Each of the counterexamples, including those after $\mathcal{A}_{24}$, takes under 3 seconds to find.

After the counterexample `[([])]` returned for $\mathcal{A}_{24}$ however, L* begins to find irregularities in the LSTM's behaviour, and jumps from the 26 state DFA shown in Figure 4.4 to the 47 state DFA shown in Figure 4.5. The new hypothesis shows us how the GRU has overfitted to the training data. For example, one of the shortest sequences reaching the 'new' accepting state 41 is `[([a])`, and indeed checking the GRU shows that it accepts this sequence despite it being incorrectly balanced. Following the transitions for this sequence, the GRU's 'first mistake' appears to be on the neutral tokens of state 9, which instead of sitting on a self-loop now go to the different state 22.

Up until $\mathcal{A}_{24}$, the L* refinement time (time from counterexample to next equivalence query) was < 10 seconds per hypothesis. The next refinement, creating $\mathcal{A}_{25}$, takes 68 seconds however, and from there all remaining refinements take $15 - 35$ seconds each.

Sampling the LM-RNN for Equivalence Queries

Long Samples We take the same Dyck-3 RNN as above and again use L* to extract from it for 400 seconds, but this time with the equivalence query based only on comparison of samples generated from the RNN's distribution. Specifically, for each equivalence query, we sample sequences up to length 100 indefinitely (as the focus here is finding counterexamples, not reaching equivalence quickly) with tokens chosen according to the GRU's next-token distribution.

Sampling the GRU is effective for creating well balanced nested parentheses, and the method rejects the initial hypotheses of L* (in which the parentheses are not yet nested), in under one second. The counterexample has 57 tokens and is:

`{c{}{(b[]){()}c}[]()}({{{}c}ccca}cc[]){b}bbb[]abc[]a[c]()`

which reaches a maximum nesting depth of 4 and shows multiple parentheses nesting combinations. Unfortunately, a second equivalence query is never made before reaching the time limit. The length of the counterexample slows L* down (it has polynomial time complexity in, among other things, the length of its counterexamples), and—possibly more significantly—it is possible that this counterexample has led L* to many 'incorrect' behaviours in the RNN, forcing it to begin working on a large DFA covering all of them at a very early stage in the extraction.

29. `((([])))` 30. `(([{}]))` 31. `(([[]])]]` 32. `(()[])`. Excluding the third counterexample `[}`, which teaches L* of an incorrectly balanced pair that must be rejected, each counterexample describes a new way to nest parentheses pairs in each other, and is accepted by the RNN. Unfortunately towards the end errors show up, and we see in the 30$^{\text{th}}$ counterexample an incorrectly balanced sequence that the RNN accepts.

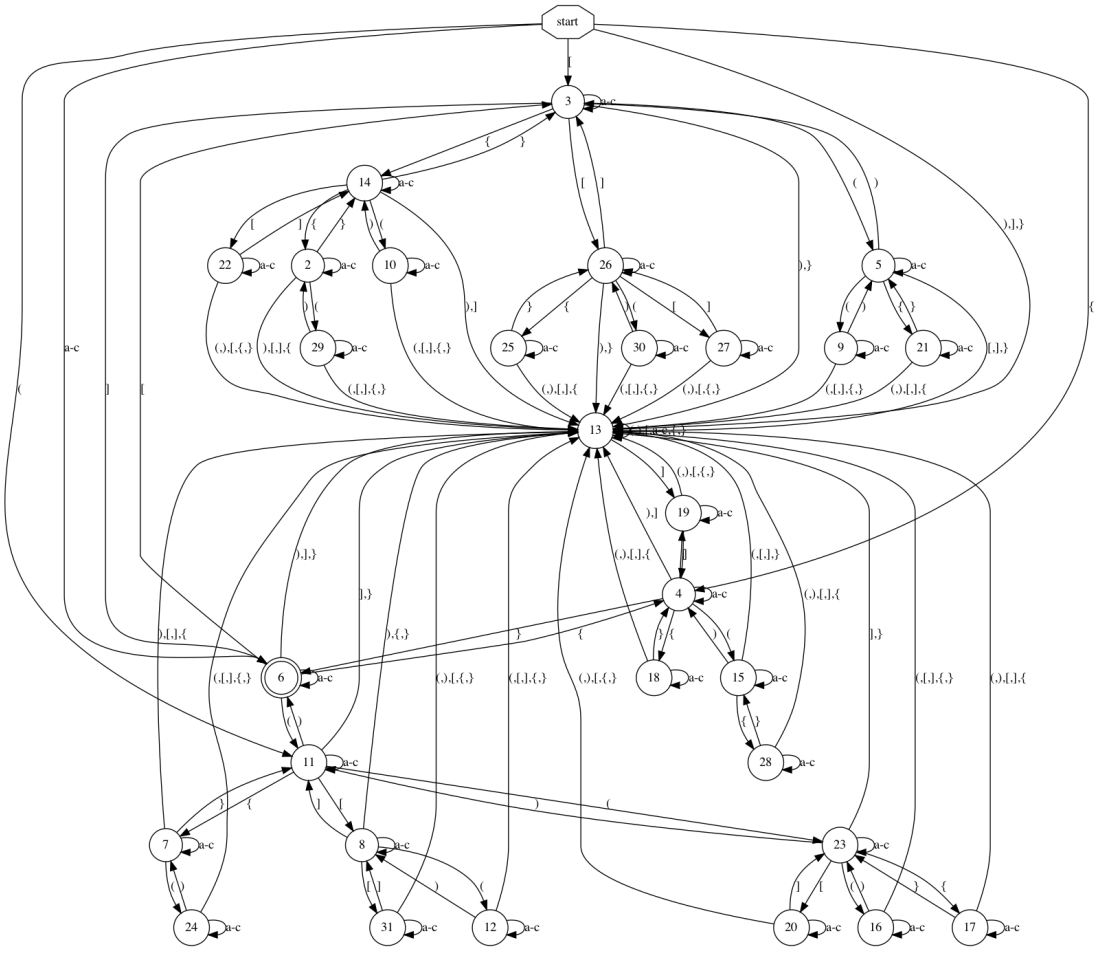


Figure 4.6: The last DFA extracted from the LM-GRU trained on Dyck-3 with neutral tokens `a-c`, when extracting with L* for 400 seconds and only using LM-sampling with maximum length 10 for the equivalence queries. It is not a subset of Dyck-3, for example, it accepts the sequence `]]]}`. This seems to be an oversight in the extraction: the RNN does not accept this sequence, and an appropriate counterexample would fix this.

.

LM Sampling: Short Samples A second attempt at extraction with RNN-sampled counterexamples,[37] this time with maximum sample length 10, creates 23 DFAs. The last of these is shown in Figure 4.6.

The equivalence queries are fast (the first ten take <1 second each, and all take <6 seconds),[38] though the extraction does not as clearly resemble Dyck-3: the DFAs have irregularities relative to those obtained with the abstraction-based L* extraction method. We do not know whether this is due to the random sampling missing key counterexamples (such as the `[}` counterexample in Section 4.7.1) or a reflection of unwanted behaviours in the RNN, but initial checks of misclassified sequences in the last DFA of this extraction show that the RNN actually classifies them correctly, suggesting that at least some key counterexamples could help 'clean' these DFAs.

[37] Again with reject threshold $t = 0.01$ and time limit 400s.

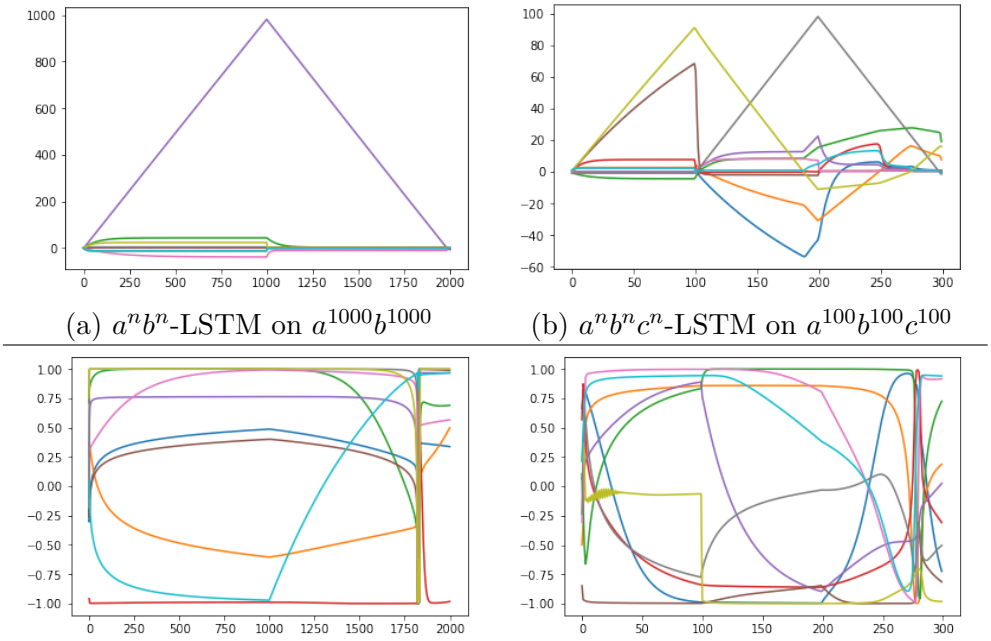
[38] The counterexamples are (examples rejected by the RNN are marked with **R**): 1. **R**: `ab(){c}[(c` 2. `{()}[ac[]]` 3. `[{}]` 4. `a()b([][])` 5. `{a}({})ba` 6. `c{{}c}b(b)` 7. `b[({})b()]` 8. **R**: `b[[][[]{}]` 9. `a([a[]]())` 10. `{()[][]}a` 11. `[acaa{()}]` 12. `b[{{()}}]a` 13. `((()))c{}` 14. **R**: `(c)ca[{[]}` 15. `{[[]]}[]ca` 16. `(({}))` 17. `([()])` 18. `(({}b[])c)` 19. `({}){(){}})` 20. `[]{bb({})}` 21. `[(())c]a` 22. `cbb[[()c]]` 23. `a[(a[])]ab`. The last counterexample is given only 5 seconds before the time limit, and so the 24th equivalence query is not reached.

Chapter 5

On the Practical Computational Power of Finite Precision RNNs for Language Recognition

5.1 Introduction

Recurrent Neural Network (RNNs) emerge as very strong learners of sequential data. A famous result by Siegelmann and Sontag [SS92; SS94], and its extension in [Sie99], demonstrates that an Elman-RNN [Elm90] with a sigmoid activation function, rational weights and infinite precision states can simulate a Turing-machine in real-time, making RNNs Turing-complete. Recently, Chen et al [CGKM17] extended the result to the ReLU activation function. However, these constructions (a) assume reading the entire input into the RNN state and only then performing the computation, using *unbounded time*; and (b) rely on having *infinite precision* in the network states. As argued by Chen et al [CGKM17], this is not the model of RNN computation used in NLP applications. Instead, RNNs are often used by feeding an input sequence into the RNN one item at a time, each immediately returning a state-vector that corresponds to a prefix of the sequence and which can be passed as input for a subsequent feed-forward prediction network operating in constant time. The amount of tape used by a Turing machine under this restriction is linear in the input length, reducing its power to recognition of context-sensitive language. More importantly, computation is often performed on GPUs with 32bit floating point computation, and there is increasing evidence that competitive performance can be achieved also for quantised networks with 4-bit weights or fixed-point arithmetics [HCS+16]. The construction of [Sie99] implements pushing 0 into a binary stack by the operation $g \leftarrow g/4 + 1/4$. This allows pushing roughly 15 zeros before reaching the limit of the 32bit floating point precision. Finally, RNN solutions that rely on carefully orchestrated mathematical constructions are unlikely to be found using backpropagation-based training.



(a) $a^n b^n$-LSTM on $a^{1000} b^{1000}$ (b) $a^n b^n c^n$-LSTM on $a^{100} b^{100} c^{100}$

(c) $a^n b^n$-GRU on $a^{1000} b^{1000}$ (d) $a^n b^n c^n$-GRU on $a^{100} b^{100} c^{100}$

Figure 5.1: Activations—c for LSTM and h for GRU—for networks trained on $a^n b^n$ and $a^n b^n c^n$. The LSTM has clearly learned to use an explicit counting mechanism, in contrast with the GRU.

In this work we restrict ourselves to *input-bound recurrent neural networks with finite-precision states (IBFP-RNN)*, trained using back-propagation. This class of networks is likely to coincide with the networks one can expect to obtain when training RNNs for NLP applications. An IBFP Elman-RNN is finite state. But what about other RNN variants? In particular, we consider the Elman RNN (SRNN) [Elm90] with squashing and with ReLU activations, the Long Short-Term Memory (LSTM) [HS97] and the Gated Recurrent Unit (GRU) [CvG+14; CGCB14a].

The common wisdom is that the LSTM and GRU introduce additional *gating components* that handle the vanishing gradients problem of training SRNNs, thus stabilising training and making it more robust. The LSTM and GRU are often considered as almost equivalent variants of each other.

We show that in the input-bound, finite-precision case, there is a real difference between the computational capacities of the LSTM and the GRU: the LSTM can easily perform unbounded counting, while the GRU (and the SRNN) cannot. This makes the LSTM a variant of a k-counter machine [FMR68], while the GRU remains finite-state. Interestingly, the SRNN with ReLU activation followed by an MLP classifier also has power similar to a k-counter machine.

These results suggest there is a class of formal languages that can be recognised by LSTMs but not by GRUs. In Section 5.5, we demonstrate that for at least two such languages, the LSTM manages to learn the desired concept classes using back-

propagation, while using the hypothesised control structure. Figure 5.1 shows the activations of 10-d LSTM and GRU trained to recognise the languages $a^n b^n$ and $a^n b^n c^n$. It is clear that the LSTM learned to dedicate specific dimensions for counting, in contrast to the GRU.[1]

5.2 The RNN Models

In this work we will discuss several different RNN architectures, we elaborate on their specific definitions here. Recall that the recursive component of the RNN—which receives a current RNN state and input token, and produces the next state—is referred to as g_R

Elman-RNN (SRNN) In the Elman-RNN [Elm90], also called the Simple RNN (SRNN), the function g_R takes the form of an affine transform followed by a tanh nonlinearity:

$$h_t = \tanh(Wx_t + Uh_{t-1} + b) \tag{5.1}$$

Elman-RNNs are known to be at-least finite-state. Siegelmann proved that the tanh can be replaced by any other squashing function without sacrificing computational power [Sie96].

IRNN The IRNN model, explored by [LJH15], replaces the tanh activation with a non-squashing ReLU:

$$h_t = max(0, (Wx_t + Uh_{t-1} + b)) \tag{5.2}$$

The computational power of such RNNs (given infinite precision) is explored in [CGKM17].

Gated Recurrent Unit (GRU) In the GRU [CvG+14], the function g_R incorporates a *gating mechanism*, taking the form:

$$\begin{aligned} z_t &= \sigma(W^z x_t + U^z h_{t-1} + b^z) && (5.3)\\ r_t &= \sigma(W^r x_t + U^r h_{t-1} + b^r) && (5.4)\\ \tilde{h}_t &= \tanh(W^h x_t + U^h(r_t \circ h_{t-1}) + b^h) && (5.5)\\ h_t &= z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t && (5.6) \end{aligned}$$

Where σ is the sigmoid function and $\circ$ is the Hadamard product (element-wise product).

[1] Is the ability to perform unbounded counting relevant to "real world" NLP tasks? In some cases it might be. For example, processing linearised parse trees [VKK+15; CC16; AG17] requires counting brackets and nesting levels. Indeed, previous works that process linearised parse trees report using LSTMs and not GRUs for this purpose. Our work here suggests that this may not be a coincidence.

Long Short Term Memory (LSTM) In the LSTM [HS97], g_R uses a different gating component configuration:

$$
\begin{aligned}
f_t &= \sigma(W^f x_t + U^f h_{t-1} + b^f) && (5.7)\\
i_t &= \sigma(W^i x_t + U^i h_{t-1} + b^i) && (5.8)\\
o_t &= \sigma(W^o x_t + U^o h_{t-1} + b^o) && (5.9)\\
\tilde{c}_t &= \tanh(W^c x_t + U^c h_{t-1} + b^c) && (5.10)\\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t && (5.11)\\
h_t &= o_t \circ g(c_t) && (5.12)
\end{aligned}
$$

where g can be either tanh or the identity.

Equivalences The GRU and LSTM are at least as strong as the SRNN: by setting the gates of the GRU to $z_t = 0$ and $r_t = 1$ we obtain the SRNN computation. Similarly by setting the LSTM gates to $i_t = 1$,$o_t = 1$, and $f_t = 0$. This is easily achieved by setting the matrices W and U to 0, and the biases b to the (constant) desired gate values.

Thus, all the above RNNs can recognise finite-state languages.

5.3 Power of Counting

Power beyond finite state can be obtained by introducing counters. Counting languages and k-counter machines are discussed in depth in [FMR68]. When unbounded computation is allowed, a 2-counter machine has Turing power. However, for computation bound by input length (real-time) there is a more interesting hierarchy. In particular, real-time counting languages cut across the traditional Chomsky hierarchy: real-time k-counter machines can recognise at least one context-free language ($a^n b^n$), and at least one context-sensitive one ($a^n b^n c^n$). However, they *cannot* recognise the context free language given by the grammar $S \to x|aSa|bSb$ (palindromes).

SKCM For our purposes, we consider a simplified variant of k-counter machines (SKCM). A counter is a device which can be incremented by a fixed amount (Inc), decremented by a fixed amount (Dec) or compared to 0 (Comp0). Informally,[2] an SKCM is a finite-state automaton extended with k counters, where at each step of the computation each counter can be incremented, decremented or ignored in an input-dependent way, and state-transitions and accept/reject decisions can inspect the counters' states using Comp0. The results for the three languages discussed above hold for the SKCM variant as well, with proofs provided in Section 5.7.

[2]Formal definition is given in Section 5.7.

5.4 RNNs as SKCMs

In what follows, we consider the effect on the state-update equations on a single dimension, $h_t[j]$. We omit the index $[j]$ for readability.

LSTM The LSTM acts as an SKCM by designating k dimensions of the memory cell c_t as counters. In non-counting steps, set $i_t = 0, f_t = 1$ in Equations 8 and 9. In counting steps, the counter direction (+1 or -1) is set in $\tilde{c_t}$ (Equation 11) based on the input x_t and state h_{t-1}. The counting itself is performed in Equation 12, after setting $i_t = f_t = 1$. The counter can be reset to 0 by setting $i_t = f_t = 0$.

Finally, the counter values are exposed through $h_t = o_t g(c_t)$, making it trivial to compare the counter's value to 0.[3]

We note that this implementation of the SKCM operations is achieved by saturating the activations to their boundaries, making it relatively easy to reach and maintain in practice.

SRNN The finite-precision SRNN cannot designate unbounded counting dimensions.

The SRNN update equation is:

$$h_t = \tanh(Wx + Uh_{t-1} + b)$$

$$h_t[i] = \tanh(\sum_{j=1}^{d_x} W_{ij}x[j] + \sum_{j=1}^{d_h} U_{ij}h_{t-1}[j] + b[i])$$

By properly setting U and W, one can get certain dimensions of h to update according to the value of x, by $h_t[i] = \tanh(h_{t-1}[i] + w_i x + b[i])$. However, this counting behaviour is within a tanh activation. Theoretically, this means unbounded counting cannot be achieved without infinite precision. Practically, this makes the counting behaviour inherently unstable, and bounded to a relatively narrow region. While the network could adapt to set w to be small enough such that counting works for the needed range seen in training without overflowing the tanh, attempting to count to larger n will quickly leave this safe region and diverge.

[3]Some further remarks on the LSTM: LSTM supports both increment and decrement in a single dimension. The counting dimensions in c_t are exposed through a function g. For both $g(x) = x$ and $g(x) = \tanh(x)$, it is trivial to do compare 0. Another operation of interest is comparing two counters (for example, checking the difference between them). This cannot be reliably achieved with $g(x) = \tanh(x)$, due to the non-linearity and saturation properties of the tanh function, but is possible in the $g(x) = x$ case. LSTM can also easily set the value of a counter to 0 in one step. The ability to set the counter to 0 gives slightly more power for real-time recognition, as discussed in [FMR68].

Relation to known architectural variants: Adding peephole connections [GS00] essentially sets $g(x) = x$ and allows comparing counters in a stable way. Coupling the input and the forget gates ($i_t = 1 - f_t$) [GSK+17] removes the single-dimension unbounded counting ability, as discussed for the GRU.

IRNN Finite-precision IRNNs can perform unbounded counting conditioned on input symbols. This requires representing each counter as two dimensions, and implementing INC as incrementing one dimension, DEC as incrementing the other, and COMP0 as comparing their difference to 0. Indeed, Appendix A in [CGKM17] provides concrete IRNNs for recognising the languages $a^n b^n$ and $a^n b^n c^n$. This makes IBFP-RNN with ReLU activation more powerful than IBFP-RNN with a squashing activation. Practically, ReLU-activated RNNs are known to be notoriously hard to train because of the exploding gradient problem.

GRU Finite-precision GRUs cannot implement unbounded counting on a given dimension. The tanh in Equation 6 combined with the interpolation (tying z_t and $1 - z_t$) in Equation 7 restricts the range of values in h to between -1 and 1, precluding unbounded counting with finite precision. Practically, the GRU can learn to count up to some bound m seen in training, but will not generalise well beyond that.[4] Moreover, simulating forms of counting behaviour in Equation 7 require consistently setting the gates z_t, r_t and the proposal $\tilde{h}_t$ to precise, non-saturated values, making it much harder to find and maintain stable solutions.

Summary We show that LSTM and IRNN can implement unbounded counting in dedicated counting dimensions, while the GRU and SRNN cannot. This makes the LSTM and IRNN at least as strong as SKCMs, and strictly stronger than the SRNN and the GRU.[5]

5.5 Experimental Results

Can the LSTM indeed learn to behave as a k-counter machine when trained using backpropagation? We show empirically that:

1. LSTMs can be trained to recognise $a^n b^n$ and $a^n b^n c^n$.
2. These LSTMs generalise to much higher n than seen in the training set (though not infinitely so).
3. The trained LSTM learn to use the per-dimension counting mechanism.

[4] One such mechanism could be to divide a given dimension by $k > 1$ at each symbol encounter, by setting $z_t = 1/k$ and $\tilde{h_t} = 0$. Note that the inverse operation would not be implementable, and counting down would have to be realised with a second counter.

[5] One can argue that other counting mechanisms—involving several dimensions—are also possible. Intuitively, such mechanisms cannot be trained to perform unbounded counting based on a finite sample as the model has no means of generalising the counting behaviour to dimensions beyond those seen in training. We discuss this more in depth in Section 5.6, where we also prove that an SRNN cannot represent a binary counter.

4. The GRU can also be trained to recognise $a^n b^n$ and $a^n b^n c^n$, but they do not have clear counting dimensions, and they generalise to much smaller n than the LSTMs, often failing to generalise correctly even for n within their training domain.

5. Trained LSTM networks outperform trained GRU networks on random test sets for the languages $a^n b^n$ and $a^n b^n c^n$.

Similar empirical observations regarding the ability of the LSTM to learn to recognise $a^n b^n$ and $a^n b^n c^n$ are described also in [GS01].

We train 10-dimension, 1-layer LSTM and GRU networks to recognise $a^n b^n$ and $a^n b^n c^n$. For $a^n b^n$ the training samples went up to $n = 100$ and for $a^n b^n c^n$ up to $n = 50$.[6]

Results On $a^n b^n$, the LSTM generalises well up to $n = 256$, after which it accumulates a deviation making it reject $a^n b^n$ but recognise $a^n b^{n+1}$ for a while, until the deviation grows.[7] The GRU does not capture the desired concept even within its training domain: accepting $a^n b^{n+1}$ for $n > 38$, and also accepting $a^n b^{n+2}$ for $n > 97$. It stops accepting $a^n b^n$ for $n > 198$.

On $a^n b^n c^n$ the LSTM recognises well until $n = 100$. It then starts accepting also $a^n b^{n+1} c^n$. At $n > 120$ it stops accepting $a^n b^n c^n$ and switches to accepting $a^n b^{n+1} c^n$, until at some point the deviation grows. The GRU accepts already $a^9 b^{10} c^{12}$, and stops accepting $a^n b^n c^n$ for $n > 63$.

Figure 5.1a plots the activations of the 10 dimensions of the $a^n b^n$-LSTM for the input $a^{1000} b^{1000}$. While the LSTM misclassifies this example, the use of the counting mechanism is clear. Figure 5.1b plots the activation for the $a^n b^n c^n$ LSTM on $a^{100} b^{100} c^{100}$. Here, again, the two counting dimensions are clearly identified—indicating the LSTM learned the canonical 2-counter solution—although the slightly-imprecise counting also starts to show. In contrast, Figures 5.1c and 5.1d show the state values of the GRU-networks. The GRU behaviour is much less interpretable than the LSTM. In the $a^n b^n$ case, some dimensions may be performing counting within a bounded range, but move to erratic behaviour at around $t = 1750$ (the network starts to misclassify on sequences much shorter than that). The $a^n b^n c^n$ state dynamics are even less interpretable.

[6] Implementation in DyNet, using the SGD Optimiser. Positive examples are generated by sampling n in the desired range. For negative examples we sample 2 or 3 n values independently, and ensuring at least one of them differs from the others. We dedicate a portion of the examples as the dev set, and train up to 100% dev set accuracy.

[7] These fluctuations occur as the networks do not fully saturate their gates, meaning the LSTM implements an imperfect counter that accumulates small deviations during computation, e.g.: increasing the counting dimension by 0.99 but decreasing only by 0.98. Despite this, we see that the its solution remains much more robust than that found by the GRU—the LSTM has learned the essence of the counting based solution, but its implementation is imprecise.

Finally, we created 1000-sample test sets for each of the languages. For $a^n b^n$ we used words with the form $a^{n+i}b^{n+j}$ where $n \in \text{rand}(0, 200)$ and $i, j \in \text{rand}(-2, 2)$, and for $a^n b^n c^n$ we use words of the form $a^{n+i}b^{n+j}c^{n+k}$ where $n \in \text{rand}(0, 150)$ and $i, j, k \in \text{rand}(-2, 2)$. The LSTM's accuracy was 100% and 98.6% on $a^n b^n$ and $a^n b^n c^n$ respectively, as opposed to the GRU's 87.0% and 86.9%, also respectively.

All of this empirically supports our result, showing that IBFP-LSTMs can not only theoretically implement "unbounded" counters, but also learn to do so in practice (although not perfectly), while IBFP-GRUs do not manage to learn proper counting behaviour, even when allowing floating point computations.

5.6 Impossibility of Counting in Binary

While we have seen that the SRNN and GRU cannot allocate individual counting dimensions, the question remains whether they can count using a more elaborate mechanism, perhaps over several dimensions. We show here that one such mechanism—a binary counter—is not implementable in the SRNN.

For the purposes of this discussion, we first define a binary counter in an RNN.

Binary Interpretation In an RNN with hidden state values in the range $(-1, 1)$, the *binary interpretation* of a sequence of dimensions $d_1, ..., d_n$ of its hidden state is the binary number obtained by replacing each positive hidden value in the sequence with a '1' and each negative value with a '0'. For instance: the binary interpretation of the dimensions 3,0,1 in the hidden state vector $(0.5, -0.1, 0.3, 0.8)$ is 110, i.e., 6.

Binary Counting We say that the dimensions $d_1, d_2, ..., d_n$ in an RNN's hidden state implement a *binary counter* in the RNN if, in every transition, their binary interpretation either increases, decreases, resets to 0, or doesn't change.[8]

A similar pair of definitions can be made for state values in the range $(0, 1)$.

We first note intuitively that an SRNN would not generalise binary counting to a counter with dimensions beyond those seen in training—as it would have no reason to learn the 'carry' behaviour between the untrained dimensions. We prove further that we cannot reasonably implement such counters regardless.

We now present a proof sketch that a single-layer SRNN with hidden size $n \geq 3$ cannot implement an n-dimensional binary counter that will consistently increase on one of its input symbols. After this, we will prove that even with helper dimensions, we cannot implement a counter that will consistently increase on one input token and decrease on another—as we might want in order to classify the language of all words

[8] We note that the SKCMs presented here are more restricted in their relation between counter action and transition, but prefer here to give a general definition. Our proof will be relevant even within the restrictions.

w for which $\#_a(w) = \#_b(w)$.[9]

Consistently Increasing Counter: The proof relies on the linearity of the affine transform $Wx + Uh + b$, and the fact that 'carry' is a non-linear operation. We work with state values in the range $(-1, 1)$, but the proof can easily be adapted to $(0, 1)$ by rewriting h as $h' + 0.5$, where $h' = h - 0.5$ is a vector with values in the range $(-0.5, 0.5)$.

Suppose we have a single-layer SRNN with hidden size $n = 3$, such that its entire hidden state represents a binary counter that increases every time it receives the input symbol a. We denote by x_a the embedding of a, and assume w.l.o.g. that the hidden state dimensions are ordered from MSB to LSB, e.g. the hidden state vector $(1, 1, -1)$ represents the number 110=6.

Recall that the binary interpretation of the hidden state relies only on the signs of its values. We use p and n to denote 'some' positive or negative value, respectively. Then the number 6 can be represented by any state vector (p, p, n).

Recall also that the SRNN state transition is

$$h_t = \tanh(Wx_t + Uh_{t-1} + b)$$

and consider the state vectors $(-1, 1, 1)$ and $(1, -1, -1)$, which represent 3 and 4 respectively. Denoting $\tilde{b} = Wx_a + b$, we find that the constants U and $\tilde{b}$ must satisfy:

$$\begin{aligned} tanh(U(-1, 1, 1) + \tilde{b}) &= (p, n, n) \\ tanh(U(1, -1, -1) + \tilde{b}) &= (p, n, p) \end{aligned}$$

As tanh is sign-preserving, this simplifies to:

$$\begin{aligned} U(-1, 1, 1) &= (p, n, n) - \tilde{b} \\ U(1, -1, -1) &= (p, n, p) - \tilde{b} \end{aligned}$$

Noting the linearity of matrix multiplication and that $(1, -1, -1) = -(-1, 1, 1)$, we obtain:

$$U(-1, 1, 1) = U(-(1, -1, -1)) = -U(1, -1, -1)$$

$$(p, n, n) - \tilde{b} = \tilde{b} - (p, n, p)$$

i.e. for some assignment to each p and n, $2\tilde{b} = (p, n, n) + (p, n, p)$, and in particular $\tilde{b}[1] < 0$.

[9] Of course a counter could also be 'decreased' by incrementing a parallel, 'negative' counter, and implementing compare-to-zero as a comparison between these two. As intuitively no RNN could generalise binary counting behaviour to dimensions not used in training, this approach could quickly find both counters outside of their learned range even on a sequence where the difference between them is never larger than in training.

Similarly, for $(-1,-1,1)$ and $(1,1,-1)$, we obtain

$$U(-1,-1,1) = (n,p,n) - \tilde{b}$$
$$U(1,1,-1) = (p,p,p) - \tilde{b}$$

i.e.

$$(n,p,n) - \tilde{b} = \tilde{b} - (p,p,p)$$

or $2\tilde{b} = (p,p,p) + (n,p,n)$, and in particular that $\tilde{b}[1] > 0$, leading to a contradiction and proving that such an SRNN cannot exist. The argument trivially extends to $n > 3$ (by padding from the MSB).

We note that this proof does not extend to the case where additional, non counting dimensions are added to the RNN—at least not without further assumptions, such as the assumption that the counter behave correctly for *all* values of these dimensions, reachable and unreachable. One may argue then that, with enough dimensions, it could be possible to implement a consistently increasing binary counter on a *subset* of the SRNN's state.[10] We now show a counting mechanism that cannot be implemented even with such 'helper' dimensions.

Bi-Directional Counter: We show that for $n \geq 3$, no SRNN can implement an n-dimensional binary counter that increases for one token, σ_{up}, and decreases for another, σ_{down}. As before, we show the proof explicitly for $n = 3$, and note that it can be simply expanded to any $n > 3$ by padding.

Assume by contradiction we have such an SRNN, with $m \geq 3$ dimensions, and assume w.l.o.g. that a counter is encoded along the first 3 of these. We use the shorthand $(v_1, v_2, v3)c$ to show the values of the counter dimensions explicitly while abstracting the remaining state dimensions, e.g. we write the hidden state $(-0.5, 0.1, 1, 1, 1)$ as $(-0.5, 0.1, 1)c$ where $c = (1, 1)$.

Let x_{up} and x_{down} be the embeddings of σ_{up} and σ_{down}, and as before denote $b_{up} = Wx_{up} + b$ and $b_{down} = Wx_{down} + b$. Then for some reachable state $h_1 \in \mathbb{R}$ where the counter value is 1 (e.g., the state reached on the input sequence ${\sigma_{up}}^{11}$)), we find that the constants U, b_{down}, and b_{up} must satisfy:

$$tanh(Uh_1 + b_{up}) = (n,p,n)c_1$$
$$tanh(Uh_1 + b_{down}) = (n,n,n)c_2$$

(i.e., σ_{up} increases the counter and updates the additional dimensions to the values c_1, while σ_{down} decreases and updates to c_2.) Removing the sign-preserving function tanh

[10] (By storing processing information on the additional, 'helper' dimensions)

[11] (Or whichever appropriate sequence if the counter is not initiated to zero.)

we obtain the constraints

$$Uh_1 + b_{up} = (n, p, n)\text{sign}(c_1)$$
$$Uh_1 + b_{down} = (n, n, n)\text{sign}(c_2)$$

i.e. $(b_{up} - b_{down})[0:2] = (n, p, n) - (n, n, n)$, and in particular $(b_{up} - b_{down})[1] > 0$. Now consider a reachable state h_3 for which the counter value is 3. Similarly to before, we now obtain

$$Uh_3 + b_{up} = (p, n, n)\text{sign}(c_3)$$
$$Uh_3 + b_{down} = (n, p, n)\text{sign}(c_4)$$

from which we get $(b_{up} - b_{down})[0:2] = (p, n, n) - (n, p, n)$, and in particular $(b_{up} - b_{down})[1] < 0$, a contradiction to the previous statement. Again we conclude that no such SRNN can exist.

5.7 Simplified K-Counter Machines

In this section, we elaborate on the definition of k-counter machines used in this work, and the power of this model.

We use a simplified variant of the k-counter machines (SKCM) defined in [FMR68], which has no autonomous states and makes classification decisions based on a combination of its current state and counter values. This variant consumes input sequences on a symbol by symbol basis, updating at each step its state and its counters, the latter of which may be manipulated by increment, decrement, zero, or no-ops alone, and observed only by checking equivalence to zero. To define the transitions of this model its accepting configurations, we will introduce the following notations:

Notations We define $z : \mathbb{Z}^k \to \{0,1\}^k$ as follows: for every $n \in \mathbb{Z}^k$, for every $1 \leq i \leq k$, $z(n)_i = 0$ iff $n_i = 0$ (this function masks a set of integers such that only their zero-ness is observed). For a vector of operations, $o \in \{-1, +1, \times 0, \times 1\}^k$, we denote by $o(n)$ the pointwise application of the operations to the vector $n \in \mathbb{Z}^k$, e.g. for $o = (+1, \times 0, \times 1)$, $o((5, 2, 3)) = (6, 0, 3)$.

We now define the model. An *SKCM* is a tuple $M = \langle \Sigma, Q, q_o, k, \delta, u, F \rangle$ containing:

1. A finite input alphabet Σ
2. A finite state set Q
3. An initial state $q_0 \in Q$
4. $k \in \mathbb{N}$, the number of counters
5. A state transition function

$$\delta : Q \times \Sigma \times \{0,1\}^k \to Q$$

6. A counter update function[12]

$$u : \Sigma \to \{-1, +1, \times 0, \times 1\}^k$$

7. A set of accepting masked[13] configurations

$$F \subseteq Q \times \{0, 1\}^k$$

The set of *configurations* of an SKCM is the set $C = Q \times \mathbb{Z}^k$, and the initial configuration is $c_0 = (q_0, \bar{0})$ (i.e., the counters are initiated to zero). The transitions of an SKCM are as follows: given a configuration $c_t = (q, n)$ ($n \in \mathbb{Z}^k$) and input $w_t \in \Sigma$, the next configuration of the SKCM is $c_{t+1} = (\delta(q, w_t, z(n)), u(w_t)(n))$.

The language recognised by a k-counter machine is the set of words w for which the machine reaches an accepting configuration—a configuration $c = (q, n)$ for which $(q, z(n)) \in F$.

Note that while the counters can and are increased to various non-zero values, the transition function δ and the accept/reject classification of the configurations observe only their zero-ness.

5.7.1 Computational Power of SKCMs

We show that the SKCM model can recognise the context-free and context-sensitive languages $a^n b^n$ and $a^n b^n c^n$, but not the context free language of palindromes, meaning its computational power differs from the language classes defined in the Chomsky hierarchy. Similar proofs appear in [FMR68] for their variant of the k-counter machine.

$a^n b^n$: We define the following SKCM over the alphabet $\{a, b\}$:

1. $Q = \{q_a, q_b, q_r\}$
2. $q_0 = q_a$
3. $k = 1$
4. $u(a) = +1,\ u(b) = -1$
5. for any $z \in \{0, 1\}$:
   $\delta(q_a, a, z) = q_a, \quad \delta(q_a, b, z) = q_b,$
   $\delta(q_b, a, z) = q_r, \quad \delta(q_b, b, z) = q_b$
   $\delta(q_r, a, z) = q_r, \quad \delta(q_r, b, z) = q_r$
6. $C = \{(q_b, 0)\}$

[12] We note that in this definition, the counter update function depends only on the input symbol. In practice we see that the LSTM is not limited in this way, and can also update according to some state-input combinations—as can be seen when it it is taught, for instance, the language $a^n b a^n$ We do not explore this here however, leaving a more complete characterisation of the learnable models to future work.

[13] i.e., counters are observed only by zero-ness.

The state q_r is a rejecting sink state, and the states q_a and q_b keep track of whether the sequence is currently in the "a" or "b" phase. If an a is seen after moving to the b phase, the machine moves to (and stays in) the rejecting state. The counter is increased on input a and decreased on input b, and the machine accepts only sequences that reach the state q_b with counter value zero, i.e., that have increased and decreased the counter an equal number of times, without switching from b to a. It follows easily that this machine recognises exactly the language $a^n b^n$.

$a^n b^n c^n$: We define the following SKCM over the alphabet $\{a, b\}$. As its state transition function ignores the counter values, we use the shorthand $\delta(q, \sigma)$ for $\delta(q, \sigma, z)$, for all $z \in \{0, 1\}^2$.

1. $Q = \{q_a, q_b, q_c, q_r\}$
2. $q_0 = q_a$
3. $k = 2$
4. $u(a) = (+1, \emptyset)$,
   $u(b) = (-1, +1)$,
   $u(c) = (\emptyset, -1)$
5. for any $z \in \{0, 1\}$:
   $\delta(q_a, a) = q_a,\ \delta(q_a, b) = q_b,\ \ \delta(q_a, c) = q_r,$
   $\delta(q_b, a) = q_r,\ \ \delta(q_b, b) = q_b,\ \ \delta(q_b, c) = q_c,$
   $\delta(q_c, a) = q_r,\ \ \delta(q_c, b) = q_r,\ \ \delta(q_c, c) = q_c,$
   $\delta(q_r, a) = q_r,\ \ \delta(q_r, b) = q_r,\ \ \delta(q_r, c) = q_r$
6. $C = \{(q_c, 0, 0)\}$

By similar reasoning as that for $a^n b^n$, we see that this machine recognises exactly the language $a^n b^n c^n$. We note that this construction can be extended to build an SKCM for any language of the sort $a_1^n a_2^n ... a_m^n$, using $k = m - 1$ counters and $k + 1$ states.

Palindromes: We prove that no SKCM can recognise the language of palindromes defined over the alphabet $\{a, b, x\}$ by the grammar $S \to x|aSa|bSb$. The intuition is that in order to correctly recognise this language in an one-way setting, one must be able to reach a unique configuration for every possible input sequence over $\{a, b\}$ (requiring an exponential number of reachable configurations), whereas for any SKCM, the number of reachable configurations is always polynomial in the input length.[14]

Let M be an SKCM with k counters. As its counters are only manipulated by steps of 1 or resets, the maximum and minimum values that each counter can attain on any input $w \in \Sigma^*$ are $+|w|$ and $-|w|$, and in particular the total number of possible values a counter could reach at the end of input w is $2|w|+1$. This means that the total number of possible configurations M could reach on input of length n is $c(n) = |Q| \cdot (2n + 1)^k$.

[14]This will hold even if the counter update function can rely on any state-input combination.

$c(n)$ is polynomial in n, and so there exists a value m for which the number of input sequences of length m over $\{a, b\}$—2^m—is greater than $c(m)$. It follows by the pigeonhole principle that there exist two input sequences $w_1 \neq w_2 \in \{a, b\}^m$ for which M reaches the same configuration. This means that for any suffix $w \in \Sigma^*$, and in particular for $w = x \cdot w_1^{-1}$ where w_1^{-1} is the reverse of w_1, M classifies $w_1 \cdot w$ and $w_2 \cdot w$ identically—despite the fact that $w_1 \cdot x \cdot w_1^{-1}$ is in the language and $w_2 \cdot x \cdot w_1^{-1}$ is not. This means that M necessarily does not recognise this palindrome language, and ultimately that no such M exists.

Note that this proof can be easily generalised to any palindrome grammar over 2 or more characters, with or without a clear 'midpoint' marker.

Chapter 6

Learning Deterministic Weighted Automata with Queries and Counterexamples

6.1 Introduction

We address the problem of learning a probabilistic deterministic finite automaton (PDFA) from a trained recurrent neural network (RNN) [Elm90]. RNNs, and in particular their gated variants GRU [CvMBB14; CGCB14b] and LSTM [HS97], are well known to be very powerful for sequence modelling, but are not interpretable. PDFAs, which explicitly list their states, transitions, and weights, are more interpretable than RNNs [HVLS16], while still being analogous to them in behaviour: both emit a single next-token distribution from each state, and have deterministic state transitions given a state and token. They are also much faster to use than RNNs, as their sequence processing does not require matrix operations.

We present an algorithm for reconstructing a PDFA from any given black-box distribution over sequences, such as an RNN trained with a language modelling objective (LM-RNN). The algorithm is applicable for reconstruction of any weighted deterministic finite automaton (WDFA), and is guaranteed to return a PDFA when the target is stochastic—as an LM-RNN is.[1]

In previous works, Ayache et al. [AEG18] and Okudono et al. [OWSH20] show how to apply *spectral learning* [BCLQ14] to an LM-RNN to learn a *weighted finite automaton* (WFA) approximating its behaviour. WFAs are a non-deterministic version of WDFAs, and so not immediately analogous to RNNs. They are also slower to use than WDFAs, as processing each token in an input sequence requires a matrix multiplication (as opposed to a table lookup). Finally, spectral learning algorithms are not guaranteed

[1] The full definitions of WDFAs and PDFAs are presented in the Preliminaries chapter of this work, Chapter 3.

to return stochastic hypotheses even when the target is stochastic—though this can remedied by using quadratic weighted automata [Bai11] and normalising their weights. For these reasons we prefer PDFAs over WFAs for RNN approximation. Formally:

Problem Definition Given an LM-RNN R, find a PDFA W approximating R, such that for any prefix p its next-token distributions in W and in R have low total variation distance between them.

Existing works on PDFA reconstruction assume a sample based paradigm: the target cannot be queried explicitly for a sequence's probability or conditional probabilities [CT04; CO94; BCG13]. As such, these methods cannot take full advantage of the information available from an LM-RNN[2]. Meanwhile, most work on the extraction of finite automata from RNNs has focused on "binary" deterministic finite automata (DFAs) [GMC+92; CSS03; WZO+17; WGY22; MY18], which cannot fully express the behaviour of an LM-RNN.

Our Approach Following the successful application of L^* [Ang87] to RNNs for DFA extraction [WGY22], we develop an adaptation of L^* for the weighted case. The adaptation returns a PDFA when applied to a stochastic target such as an LM-RNN. It interacts with an oracle using two types of queries:

1. *Membership Queries*: requests to give the target probability of the last token in a sequence.
2. *Equivalence Queries*: requests to accept or reject a hypothesis PDFA, returning a *counterexample*—a sequence for which the hypothesis automaton and the target language diverge beyond the tolerance on the next token distribution—if rejecting.

The algorithm alternates between filling an *observation table* with observations of the target behaviour, and presenting minimal PDFAs consistent with that table to the oracle for equivalence checking. This continues until an automaton is accepted. The use of conditional properties in the observation table prevents the observations from vanishing to 0 on low probabilities. To the best of our knowledge, this is the first work on learning PDFAs from RNNs.

A key insight of our adaptation is the use of an *additive variation tolerance* $t \in [0, 1]$ when comparing rows in the table. In this framework, two probability vectors are considered t-equal if their probabilities for each event are within t of each other. Using this tolerance enables us to extract a much smaller PDFA than the original target, while still making locally similar predictions to it on any given sequence. This is necessary because RNN states are real valued vectors, making the potential number of reachable states in an LM-RNN unbounded. The tolerance is non-transitive, making construction of PDFAs from the table more challenging than in L^* . Our algorithm suggests a way to address this.

[2]It is possible to adapt these methods to an active learning setting, in which they may query an oracle for exact probabilities. However, this raises other questions: on which suffixes are prefixes compared? How does one pool the probabilities of two prefixes when merging them? We leave such an adaptation to future work.

Even with this tolerance, reaching equivalence may take a long time for large target PDFAs, and so we design our algorithm to allow anytime stopping of the extraction. The method allows the extraction to be limited while still maintaining certain guarantees on the reconstructed PDFA.

Note. While this work only discusses RNNs, the algorithm itself is actually agnostic to the underlying structure of the target, and can be applied to any autoregressive language model. In particular it may be applied to transformer decoders [VSP+17; BMR+20]. However, in this case the analogy to PDFAs breaks down.

Contributions The main contributions of this work are:

1. An algorithm for reconstructing a WDFA from any given weighted target, and in particular a PDFA if the target is stochastic.
2. A method for anytime extraction termination while still maintaining correctness guarantees.
3. An implementation of the algorithm[3] and an evaluation over extraction from LM-RNNs, including a comparison to other LM reconstruction techniques.

6.2 Related Work

In [WGY22], we presented a method for applying Angluin's exact learning algorithm L* [Ang87] to RNNs, successfully extracting deterministic finite automata (DFAs) from given binary-classifier RNNs. This work expands on this by adapting L* to extract PDFAs from LM-RNNs. To apply exact learning to RNNs, one must implement equivalence queries: requests to accept or reject a hypothesis. Okudono et al. [OWSH20] show how to adapt the equivalence query presented in [WGY22] to the weighted case.

There exist many methods for PDFA learning, originally for acyclic PDFAs [RV88; RST98; CO99], and later for PDFAs in general [CT04; CO94; TDdlH00; PG07; CG08; BCG13]. These methods split and merge states in the learned PDFAs according to sample-based estimations of their conditional distributions. Unfortunately, they require very large sample sets to succeed (e.g., [CT04] requires ~13m samples for a PDFA with $|Q|, |\Sigma| = 2$).

Distributions over Σ^* can also be represented by WFAs, though these are nondeterministic. These can be learned using *spectral algorithms*, which use SVD decomposition and $|\Sigma| + 1$ matrices of observations from the target to build a WFA [BDR09; BCLQ14; BM15; HKZ08]. Spectral algorithms have recently been applied to RNNs to extract WFAs representing their behaviour [AEG18; OWSH20; RLP19], we compare to [AEG18] in this work. The choice of observations used is also a focus of research in this field [QCG17].

For more on language modelling, see the reviews of Goodman [Goo01] or Rosenfeld [Ros00], or the Sequence Prediction Challenge (SPiCe) [BEL+16] and Probabilistic

[3] Available at `www.github.com/tech-srl/weighted_lstar`

Automaton Challenge (PAutomaC) [VEdlH14].

6.3 Additional Preliminaries

Much of the background for this project—specifically on automata, the L* algorithm, and RNNs—is presented in Chapter 3 of this thesis. For this particular project however, we also introduce a notion of *variation tolerance*, which will allow us to apply our modified version of the L* algorithm—originally designed for the symbolic setting—to a noisy, continuous RNN. The original L* algorithm is presented in Subsection 3.2.1.

Variation Tolerance Given two categorical distributions $\mathbf{p}$ and $\mathbf{q}$, their total variation distance is defined $\delta(\mathbf{p}, \mathbf{q}) \triangleq \|\mathbf{p} - \mathbf{q}\|_\infty$, i.e., the largest difference in probabilities that they assign to the same event. Our algorithm tolerates some variation distance between next-token probabilities, as follows:

Two event probabilities p_1, p_2 are called *t-equal* and denoted $p_1 \approx_t p_2$ if $|p_1 - p_2| \leq t$. Similarly, two vectors of probabilities $\mathbf{v_1}, \mathbf{v_2} \in [0,1]^n$ are called *t-equal* and denoted $\mathbf{v_1} \approx_t \mathbf{v_2}$ if $\|\mathbf{v_1} - \mathbf{v_2}\|_\infty \leq t$, i.e. if $\max_{i \in [n]}(|\mathbf{v_{1_i}} - \mathbf{v_{2_i}}|) \leq t$. For any distribution P over Σ^*, $S \subset \Sigma^{+\$}$, and $p_1, p_2 \in \Sigma^*$, we denote $p_1 \approx_{(P,S,t)} p_2$ if $P^l_S(p_1) \approx_t P^l_S(p_2)$, or simply $p_1 \approx_{(S,t)} p_2$ if P is clear from context. For any two language models A, B over Σ^* and $w \in \Sigma^{+\$}$, we say that A, B are *t-consistent on* w if $P^l_A(u) \approx_t P^l_B(u)$ for every prefix $u \neq \varepsilon$ of w. We call t the *variation tolerance*.

6.4 Learning PDFAs with Queries and Counterexamples

In this section we describe the details of our algorithm. We explain why a direct application of L* to PDFAs will not work, and then present our non-trivial adaptation. Our adaptation does not rely on the target being stochastic, and can in fact be applied to reconstruct any WDFA from an oracle.

Direct application of L* does not work for LM-RNNs: L* is a polynomial-time algorithm for learning a deterministic finite automaton (DFA) from an oracle. It can be adapted to work with oracles giving any finite number of classifications to sequences, and can be naively adapted to a probabilistic target P with finite possible next-token distributions $\{P^n(w) | w \in \Sigma^*\}$ by treating each next-token distribution as a sequence classification. However, *this will not work for reconstruction from RNNs.* This is because the set of reachable states in a given RNN is unbounded, and so also the set of next-token distributions. Thus, in order to practically adapt L* to extract PDFAs from LM-RNNs, we must reduce the number of classes L* deals with.

Variation Tolerance Our algorithm reduces the number of classes it considers by allowing an additive variation tolerance $t \in [0,1]$, and considering t-equality (as presented in Section 6.3) as opposed to actual equality when comparing probabilities. In

introducing this tolerance we must handle the fact that it may be non-transitive: there may exist $a, b, c \in [0, 1]$ such that $a \approx_t b, b \approx_t c$, but $a \not\approx_t c$. [4]

To avoid potentially grouping together all predictions on long sequences, which are likely to have very low probabilities, our algorithm observes only local probabilities. In particular, the algorithm uses an oracle that gives the last-token probability for every non-empty input sequence.

6.4.1 The Algorithm

The algorithm loops over three main steps: 1. expanding an observation table $O_{P,S}$ until it is closed and consistent, 2. constructing a hypothesis automaton, and 3. making an equivalence query about the hypothesis. The loop repeats as long as the oracle returns counterexamples for the hypotheses. In our setting, counterexamples are sequences $w \in \Sigma^*$ after which the hypothesis and the target have next-token distributions that are not t-equal. They are handled by adding all of their prefixes to P.

Our algorithm expects last token probabilities from the oracle, i.e.: $\mathcal{O}(w) = P_T^l(w)$ where P_T is the target distribution. The oracle is not queried on $P_T^l(\varepsilon)$, which is undefined. To observe the entirety of every prefix's next-token distribution, $O_{P,S}$ is initiated with $P = \{\varepsilon\}, S = \Sigma_\$$.

Step 1: Expanding the observation table $O_{P,S}$ is expanded as in L* [Ang87], but with the definition of row equality relaxed. Precisely, it is expanded until:

1. *Closedness* For every $p_1 \in P$ and $\sigma \in \Sigma$, there exists some $p_2 \in P$ such that $p_1{\cdot}\sigma \approx_{S,t} p_2$.
2. *Consistency* For every $p_1, p_2 \in P$ such that $p_1 \approx_{S,t} p_2$, for every $\sigma \in \Sigma$, $p_1{\cdot}\sigma \approx_{S,t} p_2{\cdot}\sigma$.

The table expansion is managed by a queue L initiated to P, from which prefixes p are processed one at a time as follows: If $p \notin P$, and there is no $p' \in P$ s.t. $p \approx_{(t,S)} p'$, then p is added to P. If $p \in P$ already, then it is checked for inconsistency, i.e. whether there exist p', σ s.t. $p \approx_{(t,S)} p'$ but $p{\cdot}\sigma \not\approx_{(t,S)} p'{\cdot}\sigma$. In this case a *separating suffix* $\tilde{s}$, $P_T^l(p{\cdot}\sigma{\cdot}\tilde{s}) \not\approx_t P_T^l(p'{\cdot}\sigma{\cdot}\tilde{s})$ is added to S, such that now $p \not\approx_{t,S} p'$, and the expansion restarts. Finally, if $p \in P$ then L is updated with $p{\cdot}\Sigma$.

As in L* , checking closedness and consistency can be done in arbitrary order. However, if the algorithm may be terminated before $O_{P,S}$ is closed and consistent, it is better to process L in order of prefix probability (see section 6.4.2).

Step 2: PDFA construction Intuitively, we would like to group equivalent rows of the observation table to form the states of the PDFA, and map transitions between these groups according to the table's observations. The challenge in the variation-tolerating setting is that *t-equality is not transitive.*

[4] We could define a variation tolerance by quantisation of the distribution space, which would be transitive. However this may be unnecessarily aggressive at the edges of the intervals.

Formally, let C be a partitioning (*clustering*) of P, and for each $p \in P$ let $c(p) \in C$ be the partition (*cluster*) containing p. C should satisfy:

1. *Determinism* For every $c \in C$, $p_1, p_2 \in c$, $\sigma \in \Sigma$: $p_1{\cdot}\sigma, p_2{\cdot}\sigma \in P \implies c(p_1{\cdot}\sigma) = c(p_2{\cdot}\sigma)$.
2. *t-equality (Cliques)* For every $c \in C$ and $p_1, p_2 \in c$, $p_1 \approx_{(t,S)} p_2$.

For $c \in C$, $\sigma \in \Sigma$, we denote $C_{c,\sigma} = \{c(p{\cdot}\sigma) | p \in c, p{\cdot}\sigma \in P\}$ the next-clusters reached from c with σ, and $k_{c,\sigma} \triangleq |C_{c,\sigma}|$. Note that C satisfies determinism iff $k_{c,\sigma} \leq 1$ for every $c \in C, \sigma \in \Sigma$. Note also that the constraints are always satisfiable by the clustering $C = \{\{p\}\}_{p \in P}$

We present a 4-step algorithm to solve these constraints while trying to avoid excessive partitions: [5]

1. *Initialisation*: The prefixes $p \in P$ are partitioned into some initial clustering C according to the t-equality of their rows, $O_S(p)$.
2. *Determinism I*: C is refined until it satisfies determinism: clusters $c \in C$ with tokens σ for which $k_{c,\sigma} > 1$ are split by next-cluster equivalence into $k_{c,\sigma}$ new clusters.
3. *Cliques*: Each cluster is refined into cliques (with respect to t-equality).
4. *Determinism II*: C is again refined until it satisfies determinism, as in (2).

Note that refining a partitioning into cliques may break determinism, but refining into a deterministic partitioning will not break cliques. In addition, when only allowed to refine clusters (and not merge them), all determinism refinements are necessary. Hence the order of the last 3 stages.

Once the clustering C is found, a PDFA $\mathcal{A} = \langle C, \Sigma, \delta_Q, c(\varepsilon), \delta_W \rangle$ is constructed from it. Where possible, δ_Q is defined directly by C: for every $p{\cdot}\sigma \in P$, $\delta_Q(c(p), \sigma) \triangleq c(p{\cdot}\sigma)$. For c, σ for which $k_{c,\sigma} = 0$, $\delta_Q(c, \sigma)$ is set as the best cluster match for $p{\cdot}\sigma$, where $p = \text{argmax}_{p \in c} P_T^p(p)$. This is chosen according to the heuristics presented in Section 6.4.2. The weights δ_W are defined as follows: for every $c \in C$ and $\sigma \in \Sigma_\$$,

$$\delta_W(c, \sigma) \triangleq \frac{\sum_{p \in c} P_T^p(p) \cdot P_T^l(p{\cdot}\sigma)}{\sum_{p \in c} P_T^p(p)}$$

.

Step 3: Answering Equivalence Queries We sample the target LM-RNN and hypothesis PDFA $\mathcal{A}$ a finite number of times, testing every prefix of each sample to see if it is a counterexample. If none is found, we accept $\mathcal{A}$. Though simple, we find this method to be sufficiently effective in practice. A more sophisticated approach is presented in [OWSH20].

[5] We describe our implementation of these stages in Subsection 6.10.1.

6.4.2 Practical Considerations

We present some methods and heuristics that allow a more effective application of the algorithm to large (with respect to $|\Sigma|$, $|Q|$) or poorly learned grammars.

Anytime Stopping In case the algorithm runs for too long, we allow termination before $O_{P,S}$ is closed and consistent, which may be imposed by size or time limits on the table expansion. If $|S|$ reaches its limit, the table expansion continues but stops checking consistency. If the time or $|P|$ limits are reached, the algorithm stops, constructing and accepting a PDFA from the table as is. The construction is unchanged up to the fact that some of the transitions may not have a defined destination, for these we use a "best cluster match" as described in section 6.4.2. This does not harm the guarantees on t-consistency between $O_{P,S}$ and the returned PDFA discussed in Section 6.5.

Order of Expansion As some prefixes will not be added to P under anytime stopping, the order in which rows are checked for closedness and consistency matters. We sort L by prefix weight. Moreover, if a prefix p_1 being considered is found inconsistent w.r.t. some $p_2 \in P, \sigma \in \Sigma_\$$, then all such pairs p_2, σ are considered and the separating suffix $\tilde{s} \in \sigma{\cdot}S$, $\mathcal{O}(p_1{\cdot}\tilde{s}) \not\approx_t \mathcal{O}(p_2{\cdot}\tilde{s})$ with the highest minimum conditional probability $\max_{p_2}\min_{i=1,2}\frac{P_T^p(p_i\cdot\tilde{s})}{P_T^p(p_i}$) is added to S.

Best Cluster Match Given a prefix $p \notin P$ and set of clusters C, we seek a best fit $c \in C$ for p. First we filter C for the following qualities until one is non-empty, in order of preference: 1. $c' = c \cup \{p\}$ is a clique w.r.t. t-equality. 2. There exists some $p' \in c$ such that $p' \approx_{(t,S)} p$, and c is not a clique. 3. There exists some $p' \in c$ such that $p' \approx_{(t,S)} p$. If no clusters satisfy these qualities, we remain with C. From the resulting group C' of potential matches, the best match could be the cluster c minimising $||O_S(p') - O_S(p)||_\infty$, $p' \in c$. In practice, we choose from C' arbitrarily for efficiency.

Suffix and Prefix Thresholds Occasionally when checking the consistency of two rows $p_1 \approx_t p_2$, a separating suffix $\sigma{\cdot}s \in \Sigma{\cdot}S$ will be found that is actually very unlikely to be seen after p_1 or p_2. In this case it is unproductive to add $\sigma{\cdot}s$ to S. Moreover – especially as RNNs are unlikely to perfectly learn a probability of 0 for some event – it is possible that going through $\sigma{\cdot}s$ will reach a large number of 'junk' states. Similarly when considering a prefix p, if $P_T^l(p)$ is very low then it is possible that it is the failed encoding of probability 0, and that all states reachable through p are not useful.

We introduce thresholds ε_S and ε_P for both suffixes and prefixes. When a potential separating suffix $\tilde{s}$ is found from prefixes p_1 and p_2, it is added to S only if $\min_{i=1,2} {}^{P^p(p_i\cdot\tilde{s})}/_{P^p(p_i)} \geq \varepsilon_S$. Similarly, potential new rows $p \notin P$ are only added to P if $P^l(p) \geq \varepsilon_P$.

Finding Close Rows We maintain P in a KD-tree T indexed by row entries $O_{P,S}(p)$, with one level for every column $s \in S$. When considering of a prefix $p{\cdot}\sigma$, we use T to get the subset of all potentially t-equal prefixes. T's levels are split into equal-length intervals, we find $2t$ to work well.

Choosing the Variation Tolerance In our initial experiments (on SPiCe 0-3), we used $t = 1/|\Sigma|$. The intuition was that given no data, the fairest distribution over $|\Sigma|$ is the uniform distribution, and so this may also be a reasonable threshold for a significant difference between two probabilities. In practice, we found that $t = 0.1$ often strongly differentiates states even in models with larger alphabets – except for SPiCe 1, where $t = 0.1$ quickly accepted a model of size 1. A reasonable strategy for choosing t is to begin with a large one, and reduce it if equivalence is reached too quickly.

6.5 Guarantees

We note some guarantees on the extracted model's qualities and relation to its target model. *Formal statements and full proofs for each of the guarantees listed here are given in Section 6.7.*

Model Qualities The model is guaranteed to be deterministic by construction. Moreover, if the target is stochastic, then the returned model is guaranteed to be stochastic as well.

Reaching Equivalence If the algorithm terminates successfully (i.e., having passed an equivalence query), then the returned model is t-consistent with the target on every sequence $w \in \Sigma^*$, by definition of the query. In practice we have no true oracle and only approximate equivalence queries by sampling the models, and so can only attain a probable guarantee of their relative t-consistency.

t-Consistency and Progress No matter when the algorithm is stopped, the returned model is always t-consistent with its target on every $p \in P{\cdot}\Sigma_\$$, where P is the set of prefixes in the table $O_{P,S}$. Moreover, as long as the algorithm is running, the prefix set P is always increased within a finite number of operations. This means that the algorithm maintains a growing set of prefixes on which any PDFA it returns is guaranteed to be t-consistent with the target. In particular, this means that if equivalence is not reached, at least *the algorithm's model of the target improves for as long as it runs.*

6.6 Experimental Evaluation

We apply our algorithm to 2-layer LSTMs trained on grammars from the SPiCe competition [BEL+16], adaptations of the Tomita grammars [Tom82] to PDFAs, and small

PDFAs representing languages with unbounded history. The LSTMs have input dimensions 2-60 and hidden dimensions 20-100. The LSTMs and their training methods are fully described in Subsection 6.10.2.

Compared Methods We compare our algorithm to the sample-based method ALERGIA [CO94], the spectral algorithm used in [AEG18], and n-grams. An n-gram is a PDFA whose states are a sliding window of length $n-1$ over the input sequence, with transition function $\sigma_1 \cdot ... \cdot \sigma_n, \sigma \mapsto \sigma_2 \cdot ... \sigma_n \cdot \sigma$. The probability of a token σ from state $s \in \Sigma^{n-1}$ is the MLE estimate $\frac{N(s \cdot \sigma)}{N(s)}$, where $N(w)$ is the number of times the sequence w appears as a subsequence in the samples. For ALERGIA, we use the PDFA/DFA inference toolkit FLEXFRINGE [VH17].

Target Languages We train 10 RNNs on a subset of the SPiCe grammars, covering languages generated by HMMs, and languages from the NLP, *software*, and *biology* domains. We train 7 RNNs on PDFA adaptations of the 7 Tomita languages [Tom82], made from the minimal DFA for each language by giving each of its states a next-token distribution as a function of whether it is accepting or not. We give a full description of the Tomita adaptations and extraction results in Section 6.9. As we show in Subsection 6.6.1, the n-gram models prove to be very strong competitors on the SPiCe languages. To this end, we consider three additional languages that need to track information for an unbounded history, and thus cannot be captured by *any* n-gram model. We call these UHLs (unbounded history languages).

UHLs 1 and 2 are PDFAs that cycle through 9 and 5 states with different next token probabilities. UHL 3 is a weighted adaptation of the 5^{th} Tomita grammar, changing its next-token distribution according to the parity of the seen `0`s and `1`s. The UHLs are drawn in Section 6.9.

Extraction Parameters Most of the extraction parameters differ between the RNNs, and are described in the results tables (6.1, 6.2). For our algorithm, we always limited the equivalence query to 500 samples. For the spectral algorithm, we made WFAs for all ranks $k \in [50], k = 50m, m \in [10]$, $k = 100m, m \in [10]$, and $k = rank(H)$. For the n-grams we used all $n \in [6]$. For these two, we always show the best results for NDCG and WER. For ALERGIA in the FLEXFRINGE toolkit, we use the parameters `symbol_count=50` and `state_count=N`, with `N` given in the tables.

Evaluation Measures We evaluate the extracted models against their target RNNs on word error rate (WER) and on normalised discounted cumulative gain (NDCG), which was the scoring function for the SPiCe challenge. In particular the SPiCe challenge evaluated models on $NDCG_5$, and we evaluate the models extracted from the SPiCe RNNs on this as well. For the UHLs, we use $NDCG_2$ as they have smaller alphabets. We do not use probabilistic measures such as perplexity, as the spectral algorithm is not guaranteed to return probabilistic automata.

1. *Word error rate (WER)*: The WER of model A against B on a set of predictions is the fraction of next-token predictions (most likely next token) that are different

in A and B.

2. *Normalised discounted cumulative gain (NDCG)*: The NDCG of A against B on a set of sequences $\{w\}$ scores A's ranking of the top k most likely tokens after each sequence w, $a_1, ..., a_k$, in comparison to the actual most likely tokens given by B, $b_1, ..., b_k$. Formally:

$$NDCG_k(a_1, ..., a_k) = \sum_{n \in [k]} \frac{P_B^l(w \cdot a_n)}{\log_2(n+1)} / \sum_{n \in [k]} \frac{P_B^l(w \cdot b_n)}{\log_2(n+1)}$$

For NDCG we sample the RNN repeatedly, taking all the prefixes of each sample until we have 2000 prefixes. We then compute the NDCG for each prefix and take the average. For WER, we take 2000 full samples from the RNN, and return the fraction of errors over all of the next-token predictions in those samples. An ideal WER and NDCG is 0 and 1, we note this with $\downarrow, \uparrow$ in the tables.

6.6.1 Results and Discussion

Tables 6.1 and 6.2 show the results of extraction from the SPiCe and UHL RNNs, respectively. In them, we list our algorithm as WL* (Weighted L*). For the WFAs and n-grams, which are generated with several values of k (rank) and n, we show the best scores for each metric. We list the size of the best model for each metric. We do not report the extraction times separately, as they are very similar: the majority of time in these algorithms is spent generating the samples or Hankel matrices.

For PDFAs and WFAs the size columns present the number of states, for the WFAs this is equal to the rank k with which they were reconstructed. For n-grams the size is the number of table entries in the model, and the chosen value of n is listed in brackets. In the SPiCe languages, our algorithm did not reach equivalence, and used between 1 and 6 counterexamples for every language before being stopped – with the exception of SPiCe 1 with $t = 0.1$, which reached equivalence on a single state. The UHLs and Tomitas used 0-2 counterexamples each before reaching equivalence.

The SPiCe results show a strong advantage to our algorithm in most of the small synthetic languages (1-3), with the spectral extraction taking a slight lead on SPiCe 0. However, in the remaining SPiCe languages, the n-gram strongly outperforms all other methods. Nevertheless, n-gram models are inherently restricted to languages that can be captured with bounded histories, and the UHLs demonstrate cases where this property does not hold. Indeed, all the algorithms outperform the n-grams on these languages (Table 6.2).

Our algorithm succeeds in perfectly reconstructing the target PDFA structure for each of the UHL languages, and giving it transition weights within the given variation tolerance (when extracting from the RNN and not directly from the original target, the weights can only be as good as the RNN has learned). The sample-based PDFA learning method, ALERGIA, achieved good WER and NDCG scores but did not manage to reconstruct the original PDFA structure. This may be improved by taking a larger

Language ($\lvert\Sigma\rvert, \ell$)	Model	WER ↓	NDCG↑	Time (h)	WER Size	NDCG Size
SPiCe 0	WL*	0.084	0.987	0.3	4988	4988
(4, 1.15)	Spectral	**0.053**	**0.996**	0.3	k=150	k=200
	N-Gram	0.096	0.991	0.8	1118 (n=6)	1118 (n=6)
	ALERGIA	0.353	0.961	2.9	66	66
SPiCe 1	WL* †	**0.093**	**0.971**	0.4	152	152
(20, 2.77)	WL*	0.376	0.891	0.1	1	1
	Spectral	0.319	0.909	2.9	k=12	k=11
	N-Gram	0.337	0.897	0.8	8421 (n=4)	421 (n=3)
	ALERGIA	0.376	0.892	1.2	7	7
SPiCe 2	WL* ‡	**0.08**	**0.972**	0.8	962	962
(10, 2.13)	Spectral	0.263	0.893	1.6	k=7	k=5
	N-Gram	0.278	0.894	0.8	1111 (n=4)	1111 (n=4)
	ALERGIA	0.419	0.844	1.2	11	11
SPiCe 3	WL* ‡	**0.327**	**0.928**	1.0	675	675
(10, 2.15)	Spectral	0.466	0.843	1.2	k=6	k=8
	N-Gram	0.46	0.847	0.8	1111 (n=4)	11110 (n=5)
	ALERGIA ‡‡	0.679	0.79	1.2	8	8
SPiCe 4	WL*	0.301	0.829	0.7	4999	4999
(33, 1.73)	Spectral	0.453	0.727	1.2	k=450	k=250
	N-Gram	**0.099**	**0.968**	0.8	186601 (n=6)	61851 (n=5)
	ALERGIA ‡‡	0.639	0.646	4.4	42	42
SPiCe 6	WL*	0.593	0.644	2.5	5000	5000
(60, 1.66)	Spectral	0.705	0.535	6.1	k=17	k=32
	N-Gram	**0.285**	**0.888**	0.8	127817 (n=5)	127817 (n=5)
	ALERGIA	0.687	0.538	1.9	26	26
SPiCe 7	WL*	0.626	0.642	0.5	4996	4996
(20, 1.8)	Spectral	0.801	0.472	2.4	k=50	k=27
	N-Gram	**0.441**	**0.812**	0.7	133026 (n=5)	133026 (n=5)
	ALERGIA	0.735	0.569	1.4	8	8
SPiCe 9	WL*	0.503	0.721	0.5	4992	4992
(11, 1.15)	Spectral	0.303	0.877	1.9	k=44	k=44
	N-Gram	**0.123**	**0.961**	1.0	44533 (n=6)	44533 (n=6)
	ALERGIA	0.501	0.739	1.1	44	44
SPiCe 10	WL*	0.651	0.593	0.9	4987	4987
(20, 2.1)	Spectral	0.845	0.4	1.7	k=42	k=41
	N-Gram	**0.348**	**0.845**	0.8	153688 (n=5)	153688 (n=5)
	ALERGIA	0.81	0.51	2.0	13	13
SPiCe 14	WL*	0.442	0.716	0.8	4999	4999
(27, 0.89)	Spectral††	0.531	0.653	2.4	k=100	k=100
	N-Gram	**0.079**	**0.977**	0.7	125572 (n=6)	46158 (n=5)
	ALERGIA ‡‡	0.641	0.611	1.2	19	19

Table 6.1: SPiCe results. Each language is listed with its alphabet size $|\Sigma|$ and RNN test loss ℓ. The n-grams and sample-based PDFAs were created from 5,000,000 samples, and shared samples. FLEXFRINGE was run with state_count=5000. Our algorithm was run with t=0.1, $\varepsilon_P, \varepsilon_S$=0.01, $|P|\leq$5000 and $|S|\leq$100, and spectral with $|P|, |S|$=1000, with some exceptions: †:t=0.05, $\varepsilon_S, \varepsilon_P$=0.0, ‡:$\varepsilon_S$=0, ††:$|P|, |S|$=750, ‡‡:state_count=10,000.

Language ($\|\Sigma\|, \ell$)	Model	WER↓	NDCG↑	Time (s)	WER Size	NDCG Size
UHL 1 (2, 0.72)	WL*	**0.0**	**1.0**	15	9	9
	Spectral	**0.0**	**1.0**	56	k=80	k=150
	N-Gram	0.129	0.966	259	63 (n=6)	63 (n=6)
	ALERGIA	0.004	0.999	278	56	56
UHL 2 (5, 1.32)	WL*	**0.0**	**1.0**	73	5	5
	Spectral	0.002	1.0	126	k=49	k=47
	N-Gram	0.12	0.94	269	3859 (n=6)	3859 (n=6)
	ALERGIA	0.023	0.979	329	25	25
UHL 3 (2, 0.86)	WL*	**0.0**	**1.0**	55	4	4
	Spectral	**0.0**	**1.0**	71	k=44	k=17
	N-Gram	0.189	0.991	268	63 (n=6)	63 (n=6)
	ALERGIA	0.02	0.999	319	47	47

Table 6.2: UHL results. Each language is listed with its alphabet size $|\Sigma|$ and RNN test loss ℓ. The n-grams and sample-based PDFAs were created from 500,000 samples, and shared samples. FLEXFRINGE was run with state_count = 50 . Our algorithm was run with t=0.1, $\varepsilon_P, \varepsilon_S$=0.01, $|P|\leq$5000 and $|S|\leq$100, and spectral with $|P|, |S|$=250.

sample size, though it comes at the cost of efficiency.

Tomita Grammars The full results for the Tomita extractions are given in Section 6.9.

All of the methods reconstruct them with perfect or near-perfect WER and NDCG, except for n-gram which sometimes fails. For each of the Tomita RNNs, our algorithm extracted and accepted a PDFA with identical structure to the original target in approximately 1 minute (the majority of this time was spent on sampling the RNN and hypothesis before accepting the equivalence query). These PDFAs had transition weights within the variation tolerance of the corresponding target transition weights.

On the effectiveness of n-grams The n-gram models prove to be a very strong competitors for many of the languages. Indeed, n-gram models are very effective for learning in cases where the underlying languages have strong local properties, or can be well approximated using local properties, which is rather common (see e.g., Sharan et al. [SKLV16]). However, there are many languages, including ones that can be modelled with PDFAs, for which the locality property does not hold, as demonstrated by the UHL experiments.

As n-grams are merely tables of observed samples, they are very quick to create. However, their simplicity also works against them: the table grows exponentially in n and polynomially in $|\Sigma|$. In the future, we hope that our algorithm can serve as a base for creating reasonably sized finite state machines that will be competitive on real world tasks.

6.7 Guarantees

We show that our algorithm returns a PDFA, and discuss the relation between the obtained PDFA $\mathcal{A}$ and the target T when anytime stopping is and isn't used.

6.7.1 Probability

Theorem 6.1. *The algorithm returns a PDFA.*

Proof. Let C be the final clustering of P achieved by the method in Section 6.4.1. By construction, the algorithm returns a finite state machine $A = \langle C, \Sigma, c(\varepsilon), \delta_Q, \delta_W, \beta \rangle$ with well defined states, initial state, transition weights and stopping weights. We show that this machine is deterministic and probabilistic, i.e.:

1. *Deterministic*: for every $c \in C, \sigma \in \Sigma$, $\delta_Q(c, \sigma)$ is uniquely defined
2. *Probabilistic*: for every $c \in C, \sigma \in \Sigma$: $\delta_Q(c, \sigma) \in [0, 1]$, $\beta(c) \in [0, 1]$, and $\beta(c) + \sum_{\sigma \in \Sigma} \delta_W(c, \sigma) = 1$.

Proof of (1): By the final refinement of the clustering (Determinism II), $k_{c,\sigma} \leq 1$ and so by construction $\delta_Q(c, \sigma)$ is assigned at most one value. If, and only if, $k_{c,\sigma} < 1$, then $\delta_Q(c, \sigma)$ is assigned some best available value. So $\delta_Q(c, \sigma)$ is always assigned exactly one value.

Proof of (2): the values of δ_W and β are weighted averages of probabilities, and so also in $[0, 1]$ themselves. They also sum to 1 as they are averages of distributions. Formally, for every $c \in C$:

$$\beta(c) + \sum_{\sigma \in \Sigma} \delta_W(c, \sigma) =$$

$$\frac{\sum_{p \in c} P_T^p(p) P_T^l(p{\cdot}\$)}{\sum_{p \in c} P_T^p(p)} + \sum_{\sigma \in \Sigma} \frac{\sum_{p \in c} P_T^p(p) P_T^l(p{\cdot}\sigma)}{\sum_{p \in c} P_T^p(p)} =$$

$$\frac{\sum_{p \in c} P_T^p(p) P_T^l(p{\cdot}\$)}{\sum_{p \in c} P_T^p(p)} + \frac{\sum_{p \in c} \sum_{\sigma \in \Sigma} P_T^p(p) P_T^l(p{\cdot}\sigma)}{\sum_{p \in c} P_T^p(p)} =$$

$$\frac{\sum_{p \in c} P_T^p(p) \sum_{\sigma \in \Sigma_\$} P_T^l(p{\cdot}\sigma)}{\sum_{p \in c} P_T^p(p)} \underset{(*)}{=} \frac{\sum_{p \in c} P_T^p(p)}{\sum_{p \in c} P_T^p(p)} = 1$$

where $(*)$ follows from the probabilistic behaviour of T: $\sum_{\sigma \in \Sigma_\$} P_T^l(p{\cdot}\sigma) = 1$ for any $p \in \Sigma^*$. ■

6.7.2 Progress

We consider extraction using noise tolerance t from some target $T = \langle Q, \Sigma, q^i, \delta_Q, \delta_W^T \rangle$. For the observation table $O_{P,S}$ at any stage, we denote $n_{P,S}$ the size of the largest set of pairwise t-distinguishable rows $O_S(p), p \in P$.

Let A be an automaton constructed by the algorithm, whether or not it was stopped ahead of time. Let $O_{P,S}$ be the observation table reached before making A, $C \subset \mathbb{P}(P)$ be the clustering of P attained when building A from $O_{P,S}$ (i.e., the states of A), and denote $A = \langle C, \Sigma, c^i, \delta_C, \delta_W^A \rangle$. Denote $c : P \to C$ the cluster for each prefix, i.e. $p \in c(p)$ for every $p \in P$. In addition, for every cluster $c \in C$, denote p_c the prefix $p_c \in c$ from which $\delta_W^A(c, \circ)$ was defined when building A.

We show that as the algorithm progresses, it defines a monotonically increasing

group of sequences $W \subset \Sigma^{+\$}$ on which the target T and the algorithm's automata A are t-consistent, and that this group is $P{\cdot}\Sigma_\$$.

Lemma 6.7.1. *P is always prefix closed.*

Proof. P begins as $\{\varepsilon\}$, which is prefix closed. Only two operations add to P: closedness and counterexamples. When adding from closedness, the new prefix added to P is of the form $p{\cdot}\sigma$ for $p \in P, \sigma \in \Sigma$ and so P remains prefix closed. When adding from a counterexample w, w is added along with all of its prefixes, and so P remains prefix closed. ■

Lemma 6.7.2. *For every $p \in P$, $\hat{\delta_C}(c^i, p) = c(p)$, i.e. $p \in \hat{\delta_C}(c^i, p)$.*

Proof. We show this by induction on the length of p. For $|p| = 0$ i.e. for ε, $\hat{\delta}_C(\varepsilon) = c^i$ by definition of the recursive application of δ_C, and $c^i{=}c(\varepsilon)$ by construction (in the algorithm). We assume correctness of the lemma for $|p| = n, p \in P$. Consider $p \in P$, $|p| = n+1$, denote $p = r{\cdot}\sigma, r \in \Sigma^*, \sigma \in \Sigma$. By the prefix closedness of P, $r \in P$, and so by the assumption $\hat{\delta}_C(r) = c(r)$. Now by the definition of $\hat{\delta}_C$, $\hat{\delta}_C(p) = \delta_C(\hat{\delta}_C(r), \sigma) = \delta_C(c(r), \sigma)$. By the construction of A, $c(r)$ is defined such that $\delta_C(c(r), \sigma) = c(p{\cdot}\sigma)$ for every $s \in c(r)$ s.t. $s{\cdot}\sigma \in P$, and so in particular for $r \in c(r)$, as $r{\cdot}\sigma = p \in P$). This results in $\hat{\delta}_C(p) = \delta_Q(c(r), \sigma) = c(p)$, as desired. ■

Lemma 6.7.3. *For every $p \in P$ and $\sigma \in \Sigma_\$$, $\delta_A(c(p), \sigma) \approx_t P_T^l(p{\cdot}\sigma)$.*

Proof. By construction of A, in particular by the clique requirement for the clusters of C, all of the prefixes $p' \in c(p)$ satisfy $\mathcal{O}_S(p') = O_S(p') \approx_t O_S(p) = \mathcal{O}_S(p)$, and in particular for $\Sigma_\$ \subseteq S$: $\mathcal{O}_{\Sigma_\$}(p') \approx_t \mathcal{O}_{\Sigma_\$}(p)$ (recall that S is initiated to $\Sigma_\$$ and never reduced). $\delta_A(c(p), \sigma)$ is defined as the weighted average of $\mathcal{O}(p'{\cdot}\sigma)$ for each of these $p' \in c(p)$, and so it is also t-equal to $\mathcal{O}(p{\cdot}\sigma)$ i.e. $P_T^l(p{\cdot}\sigma)$, as desired. ■

Theorem 6.2. *For every $p \in P, \sigma \in \Sigma_\$$, A, T are t-consistent on $p{\cdot}\sigma$.*

Proof. let $u \neq \varepsilon$ be some prefix of $p{\cdot}\sigma$. Necessarily $v = u_{:-1}$ is some prefix of $p \in P$, and so by the prefix-closedness of P (Lemma 6.7.1) $v \in P$. Denote $a = u_{-1} \in \Sigma_\$$. Then

$$P_T^l(u) = P_T^l(v{\cdot}a) \approx_t \delta_A(c(v), a) = \delta_A(\hat{\delta}C(v), a) = P_A^l(u)$$

where the second and third transitions are justified for $v \in P$ by Lemma 6.7.3 and Lemma 6.7.2 respectively. This for any prefix $u \neq \varepsilon$ of $p{\cdot}\sigma$, and so by definition A, T are t-consistent on $p{\cdot}\sigma$ as desired. ■

This concludes the proof that A, T are always t-consistent on $P{\cdot}\Sigma_\$$. We now show that the algorithm increases $P{\cdot}\Sigma_\$$ every finite number of operations, beginning with a direct result from Theorem 6.2:

Corollary 6.3. *Every counterexample increases P by at least* 1

Proof. Recall that counterexamples to proposed automata are sequences $w \in \Sigma^{+\$}$ for which $P_T^l(w) \not\approx_t P_A^l(w)$, and that they are handled by adding all their strict prefixes to P. Assume by contradiction some counterexample $w \in \Sigma^{+\$}$ for which P does not increase. Then in particular $w_{:-1} \in P$, and by Theorem 6.2, $P_T^l(w) = P_T^l(w_{:-1} \cdot w_{-1}) \approx_t P_A^l(w_{:-1} \cdot w_{-1}) = P_A^l(w)$, a contradiction. ■

Lemma 6.7.4. *Always, $|S| \leq \frac{|P| \cdot (|P|-1)}{2} + |\Sigma_\$|$. (i.e., every $O_{P,S}$ can only have had up to $\frac{|P| \cdot (|P|-1)}{2}$ inconsistencies in its making.)*

Proof. S is initiated to $\Sigma_\$$, so its initial size is $|\Sigma_\$|$. S is increased only following inconsistencies, cases in which there exist $p_1, p_2 \in P, \sigma \in \Sigma$ s.t. $p_1 \neq p_2$ $O_S(p_1) \approx_t O_S(p_2)$, but $\mathcal{O}_S(p_1) \not\approx_t \mathcal{O}_S(p_2)$. Once some $p_1, p_2 \in P$ cause a suffix s to be added to S, by construction of the algorithm, $O_S(p_1) \not\approx_t O_S(p_2)$ for the remainder of the run (as $s \in S$ is a suffix for which $O(p_1, s) \not\approx_t O(p_2, s)$). There are exactly $\frac{|P| \cdot (|P|-1)}{2}$ pairs $p_1 \neq p_2 \in P$ and so that is the maximum number of possible S may have been increased in any run, giving the maximum size $|S| \leq \frac{|P| \cdot (|P|-1)}{2} + |\Sigma_\$|$. ■

(Note: If the t-equality relation was transitive, it would be possible to obtain a linear bound in the size of S. However as it is not, it is possible that a separating suffix may be added to S that separates p_1 and p_2 while leaving them both t-equal to to some other p_3.)

Corollary 6.4 (Progress)**.** *For as long as the algorithm runs, it strictly expands a group $\mathbb{C} \subset \Sigma^*$ of sequences on which the automata A it returns is t-consistent with its target T.*

Proof. From Theorem 6.2, $\mathbb{C} = P \times \Sigma_\$$ is a group of sequences on which A is always t-consistent with T. We show that $\mathbb{C}$ is strictly expanding as the algorithm progresses, i.e. that every finite number of operations, P is increased by at least one sequence.

The algorithm can be split into 4 operations: searching for and handing an unclosed prefix or inconsistency, building (and presenting) a hypothesis PDFA, or handling a counterexample. We show that each one runs in finite time, and that there cannot be infinite operations without increasing P.

Finite Runtime of the Operations

Building $O_{P,S}$: Finding and handling an unclosed prefix requires a pass over all $P \times \Sigma$, while comparing row values to P – all finite as P is finite (rows are also finite as S is bounded by P's size). Similarly finding and handling inconsistencies requires a pass over rows for all $P^2 \times \sigma$, also taking finite time.

Building an Automaton requires finding a clustering of P satisfying the conditions and then a straightforward mapping of the transitions between these clusters. The clustering is built by one initial clustering (ester-etal-2016-dbscan) over the finite set

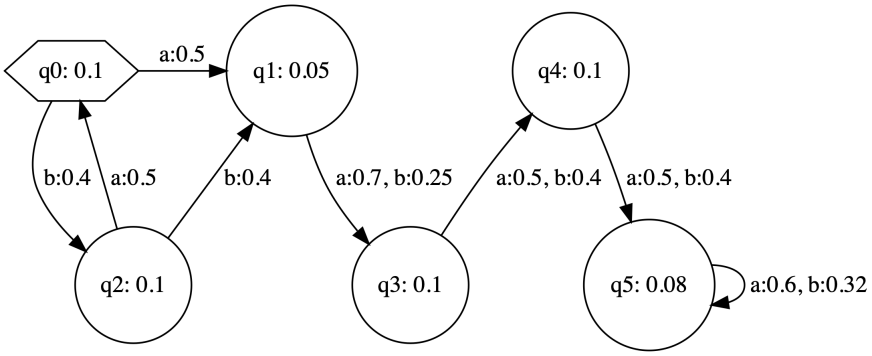


Figure 6.8.1: Target PDFA T

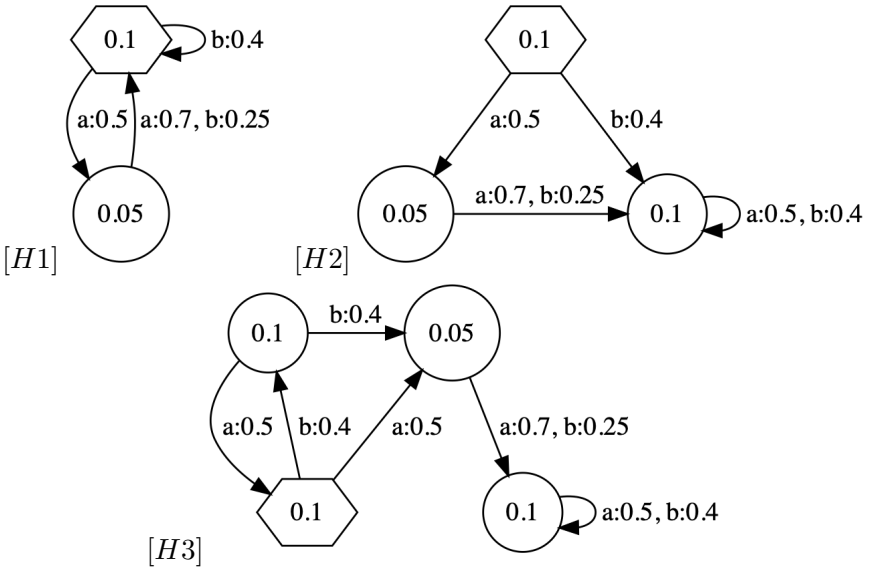


Figure 6.8.2: Hypotheses during extraction from T

P and then only refinement operations (without merges). As putting each prefix in its own cluster is a solution to the conditions, a satisfying clustering will be reached in finite time. *Counterexamples* Handling a counterexample w requires adding at most $|w|$ new rows to $O_{P,S}$. As S is finite, this is a finite operation.

Finite Operations between Additions to P Handling an unclosed prefix by construction increases P, and as shown in Corollary 6.3, so does handling a counterexample. Building a hypothesis is followed by an equivalence query, after which the algorithm will either terminate or a counterexample will be returned (increasing P). Finally, by 6.7.4, the number of inconsistencies between every increase of P is bounded. ■

6.8 Example

We extract from the PDFA T presented in 6.8.1 using prefix and suffix thresholds $\varepsilon_P, \varepsilon_S = 0$ and variation tolerance $t = 0.1$. We limit the number of samples per equivalence query to 500. This extraction will demonstrate both types of table expansions,

both types of clustering refinements, and counterexamples. Notice that in our example, the state $q5$ is t-equal with respect to next-token distribution to both $q1$ and $q3$, though they themselves are not t-equal to each other.

Extraction begins by initiating the table with $P = \{\varepsilon\}, S = \Sigma_{\$}$, and the queue Q with P. We will pop from the queue in order of prefix weight, though this is not necessary when not considering anytime stopping. At this point the table is:

P \ S	a	b	$
ε	0.5	0.4	0.1

The first prefix considered is ε, it is already in P. It is consistent simply as it is not similar to any other $p \in P$. However it might not be closed. Its continuations $\varepsilon{\cdot}\Sigma = \{a, b\}$ are added to Q, to check its closedness later. Q is now $\{a, b\}$.

Next is a (which has prefix weight 0.5). $\mathcal{O}_S(a) = (0.7, 0.25, 0.05)$, which is not t-equal to the only row in the table: $O_S(\varepsilon) = (0.5, 0.4, 0.1)$. It follows that $a_{:-1} = \varepsilon$ was not closed, and a is added to P. The table is now:

P \ S	a	b	$
ε	0.5	0.4	0.1
a	0.7	0.25	0.05

a is also consistent simply as it has no t-equal rows. Its continuations $a{\cdot}\Sigma$ are added to Q to check closedness, giving $Q = \{b, ab, aa\}$.

Now for each of $q \in Q$, $\mathcal{O}_S(q) = O_S(\varepsilon)$, meaning that the table is closed. None of the prefixes in Q are added to P, and so they are also not checked for consistency. The expansion stops and a clustering $C = \{\{\varepsilon\}, \{a\}\}$ is made (ε and a are not t-equal). The transitions are mapped and the automaton $H1$ shown in figure 6.8.2 is presented for an equivalence query.

$H1$ and T are each sampled according to their distributions up to 500 times, and $P^n_T(p), P^n_{H1}(p)$ are compared for every prefix p of each sample. This soon yields the counterexample $c = aaa$, for which $P^n_{H1}(c) = (0.7, 0.25, 0.05) \not\approx_{0.1} (0.5, 0.4, 0.1) = P^n_T(c)$. c's prefixes ε, a, aa, aaa are added to P and the expansion restarts with $Q = P$ and table:

P \ S	a	b	$
ε	0.5	0.4	0.1
a	0.7	0.25	0.05
aa	0.5	0.4	0.1
aaa	0.5	0.4	0.1

Q is processed: ε is already in P, a, b are added to Q. We check its consistency with each of its t-equal rows, aa and aaa, beginning with aa. For $a \in \Sigma$, $\mathcal{O}_S(\varepsilon{\cdot}a) = (0.7, 0.25, 0.05) \not\approx_{0.1} (0.5, 0.4, 0.1) = \mathcal{O}_S(aa{\cdot}a)$, with the biggest difference (0.2) being on the suffix $a \in S$. The separating suffix $a{\cdot}a \in \Sigma{\cdot}S$ is added to S, separating ε and aa in the table:

P \ S	a	b	$	aa
ε	0.5	0.4	0.1	0.7
a	0.7	0.25	0.05	0.5
aa	0.5	0.4	0.1	0.5
aaa	0.5	0.4	0.1	0.6

The expansion is restarted with $Q = P$. Eventually all of $P{\cdot}\Sigma$ are processed and the table is found closed and consistent. The extraction moves to constructing a hypothesis.

An initial clustering is made, in our case using `sklearn.cluster.ester-etal-2016-dbscan` with parameter `min_samples`=1. It returns $C_0 = \{\{\varepsilon, aa, aaa\}, \{a\}\}$. However, this does not satisfy the determinism requirement: for ε and aa, which are both in the same cluster, their continuations with $a \in \Sigma$ are also in P and appear in different clusters. The cluster $\{\varepsilon, aa, aaa\}$ is split such that ε and aa are separated. For aaa, whose continuation $aaaa$ is not in P, it is not important whether it joins ε or aa, and it is equally close (with respect to L_∞ distance on rows) to both. The new clustering $C = \{\{aa, aaa\}, \{a\}, \{\varepsilon\}\}$ is returned. This clustering satisfies t-equality ($aa \approx_{t,S} aaa$), and a hypothesis can be made.

For each cluster $c \in C$ there is a $p \in c$ for which $p{\cdot}a \in P$ and so all of the a-transitions are simple to map. For b, the transitions are mapped according to the closest rows in the table, e.g. the b-transition from the initial state $c(\varepsilon)$ maps to $c(aa)$, as $\mathcal{O}_S(b) = (0.5, 0.4, 0.1, 0.5) \approx_t (0.5, 0.4, 0.1, 0.5) = O_S(aa)$. This yields the PDFA $H2$ shown in 6.8.2.

Sampling $H2$ and T soon yields the counterexample bb, for which $P^n_T(bb) = (0.7, 0.25, 0.05) \not\approx_t (0.5, 0.4, 0.1) = P^n_{H2}(bb)$. All of bb's prefixes are added to P, the queue is again initiated to P, and expansion restarts with the table:

P \ S	a	b	$	aa
ε	0.5	0.4	0.1	0.7
a	0.7	0.25	0.05	0.5
aa	0.5	0.4	0.1	0.5
aaa	0.5	0.4	0.1	0.6
b	0.5	0.4	0.1	0.5
bb	0.7	0.25	0.05	0.5

When the prefix b is processed, an inconsistency is found: $b \approx_{t,S} aa$, but $\mathcal{O}_S(bb) = (0.7, 0.25, 0.05, 0.5) \not\approx_t (0.5, 0.4, 0.1, 0.6) = \mathcal{O}_S(aab)$, in particular on $a \in S$. ba is added to S, Q is reset to P, and the expansion restarts with the table:

P \ S	a	b	$	aa	ba
ε	0.5	0.4	0.1	0.7	0.5
a	0.7	0.25	0.05	0.5	0.5
aa	0.5	0.4	0.1	0.5	0.5
aaa	0.5	0.4	0.1	0.6	0.6
b	0.5	0.4	0.1	0.5	0.7
bb	0.7	0.25	0.05	0.5	0.5

This time the table is found to be closed and consistent. `ester-etal-2016-dbscan` gives the initial clustering $C_0 = \{\{\varepsilon, aa, aaa, b\}, \{a, bb\}\}$, and as before the determinism refinement separates a and ε, giving $C_1 = \{\{aa, aaa, b\}, \{a, bb\}, \{\varepsilon\}\}$. Now the t-equality requirement is checked, and the first cluster does not satisfy it: while $aa \approx_{t,S} aaa$ and $b \approx_{t,S} aaa$, $aa \not\approx_{t,S} b$. The cluster is split across the suffix with the largest range, ba, yielding the new clustering $C = \{\{aa, aaa\}, \{a, bb\}, \{\varepsilon\}, \{b\}\}$. This clustering satisfies both determinism and t-equality and the hypothesis $H3$ is made, with σ-transitions from clusters c for which there is no $p \in c$ such that $p{\cdot}\sigma \in P$ (e.g. b from $\{aa, aaa\}$) being made according to closest rows as described before.

Sampling 500 times from each of $H3$ and T yields no counterexample, and indeed none exists even though the two are not exactly the same: the distributions of states $q5, q4$ and $q3$ of T are $t = 0.1$-equal, and the PDFAs $H3$ and T are t-equal.

A note on prefix and suffix thresholds. Suppose that instead of T, we had a PDFA T' over $\Sigma = \{a, b, c\}$ as follows: T' is identical to T, except that from every state $q \in Q_T$ there is a c-transition with a very small probability ε leading to a different state of an extremely large PDFA L. If ε is very small, developing L will be of little benefit for the approximation, but waste a lot of time and space for the extraction. However, if $\varepsilon_S, \varepsilon_P > \varepsilon$, then no prefix containing c will ever be added to the table, and similarly no suffix containing c will ever be considered a separating suffix (needlessly separating two prefixes). The existence of such transitions is quite possible in RNNs: they are unlikely to perfectly learn to represent 0 even for tokens that have never been seen, and moreover never 'tame' the states that would be reached from such transitions (as they are not seen in training).

6.9 Synthetic Grammars

In this section, we expand on the details and results of the small synthetic grammars considered in this work.

6.9.1 Tomita Grammars

We adapt the Tomita grammars [Tom82] for use as weighted models as follows: for each Tomita grammar and its minimal DFA T we create a PDFA variant T_W which has the same structure as T, and in which accepting/rejecting states are differentiated by their preference for `0` or `1`. Every state in T_W has stopping probability 0.05, the states q have transition weights $0.7 \cdot 0.95 = 0.665$ and $0.3 \cdot 0.95 = 0.285$, such that $\delta_W(q, 0) = 0.665$ iff q is an accepting state in T. We show all of the adaptations in 6.9.1, labelling the weighted variants `T1` through `T7` in the same order as their binary counterparts. The images were generated using graphviz.

We train 7 RNNs on these grammars, their parameters and training routine are described in 6.10.2. We extract from them with the same algorithms as for the SPiCe and UHL languages. The extraction parameters and results are given in table 6.3.

From each of the Tomita RNNs, our algorithm successfully reconstructs a PDFA with the exact same structure as the RNN's target PDFA, and transition weights within tolerance of the corresponding weights in the target. The extracted PDFAs for each Tomita RNN are presented in 6.9.2.

6.9.2 Unbounded History Languages

The UHLs are 3 cyclic PDFAs, shown in 6.9.3. UHL 3 is a weighted adaptation of Tomita 5, where the difference in probabilities between the states is lower than in our original adaptations. This makes it harder for the n-gram to guess the current state from local clues in its window (such as many appearances of one token over another). Precisely:

> UHL1 is a 9-state cycle PDFA over $\Sigma = \{\texttt{0}, \texttt{1}\}$ that loops through all of its states one at a time, regardless of the actual input token. On all states it has stopping probability 0.05, and divides the remaining next-token distribution over `0` and `1` as follows: on all states `0` has next-token probability 0.75 and `1` has 0.15, except for the second, fifth, and ninth states, where this is reversed.
>
> UHL2 is a 5-state cycle PDFA over $\Sigma = \{\texttt{0,1,2,3,4}\}$, that loops through all of its states one at a time regardless of input token. At every state it has stopping probability 0.045, and it gives next-token probability 0.591 to a different token at each state, with the rest of the tokens getting a uniform distribution between themselves.
>
> UHL3 is a 4-state PDFA over $\Sigma = \{\texttt{0,1}\}$ that maintains the parity of the seen `0` and `1` tokens. Every state has stopping probability 0.05, and most states give `0` next-token probability 0.525 and `1` next-token probability 0.425, except for the state where the number of seen `0`s and `1`s is odd, where this is reversed.

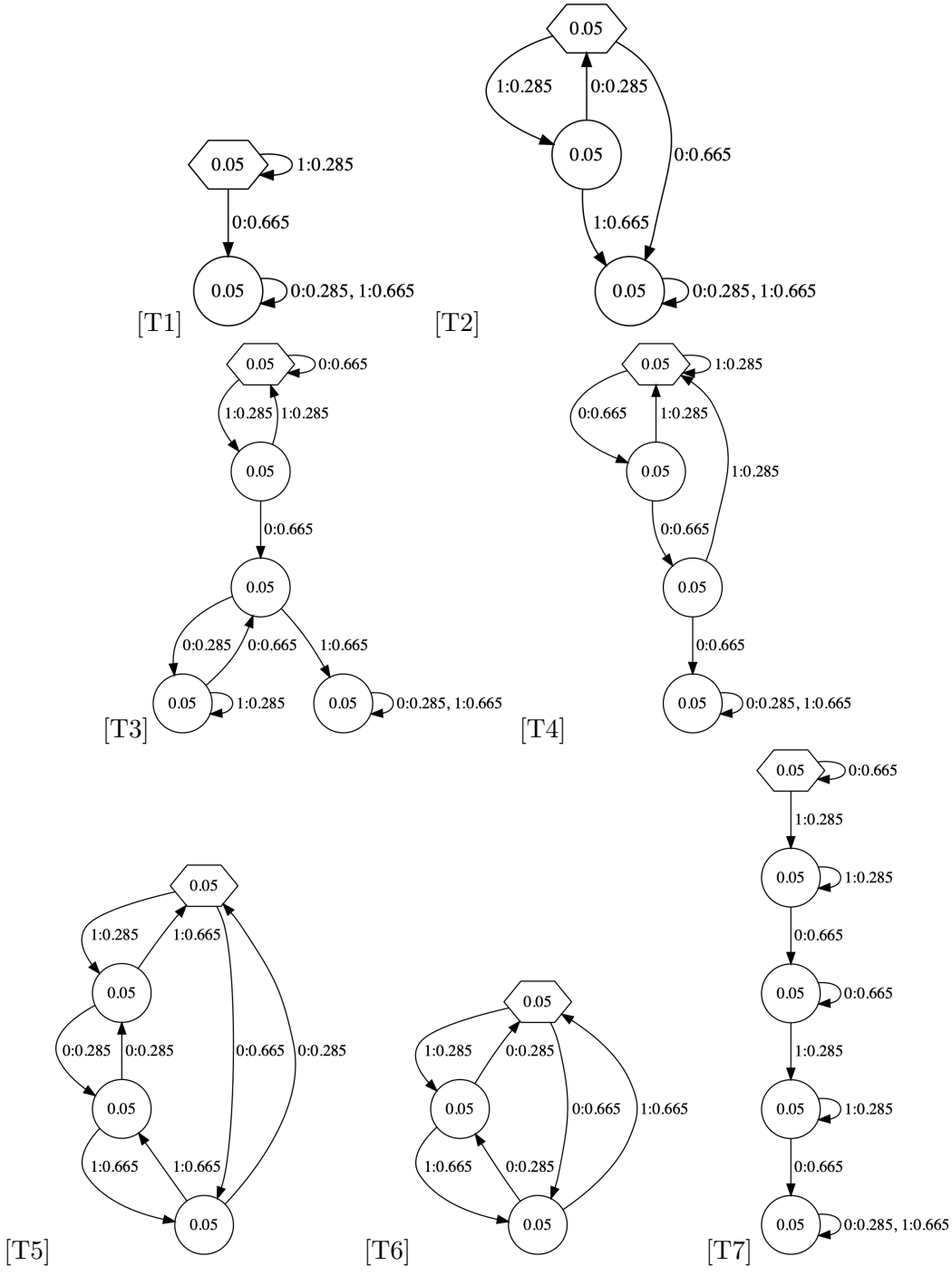


Figure 6.9.1: Weighted variants of the Tomita grammars.

Language ($\|\Sigma\|, \ell$)	Model	WER↓	NDCG↑	Time (s)	WER Size	NDCG Size
Tomita 1 (2, 0.77)	WL*	**0.0**	**1.0**	55	2	2
	Spectral	**0.0**	**1.0**	18	k=10	k=10
	N-Gram	0.0001	0.9998	27	63 (n=6)	31 (n=5)
	ALERGIA	**0.0**	**1.0**	28	8	8
Tomita 2 (2, 0.78)	WL*	**0.0**	**1.0**	55	3	3
	Spectral	**0.0**	**1.0**	13	k=10	k=10
	N-Gram	0.0	**1.0**	27	63 (n=6)	15 (n=4)
	ALERGIA	**0.0**	**1.0**	28	6	6
Tomita 3 (2, 0.78)	WL*	**0.0**	**1.0**	62	5	5
	Spectral	0.0071	0.9945	13	k=7	k=13
	N-Gram	0.0542	0.9918	27	63 (n=6)	63 (n=6)
	ALERGIA	0.0318	0.9963	28	8	8
Tomita 4 (2, 0.79)	WL*	**0.0**	**1.0**	56	4	4
	Spectral	**0.0**	**1.0**	13	k=14	k=12
	N-Gram	0.073	0.9887	27	63 (n=6)	63 (n=6)
	ALERGIA	**0.0**	**1.0**	28	9	9
Tomita 5 (2, 0.79)	WL*	**0.0**	**1.0**	56	4	4
	Spectral	0.0001	**1.0**	11	k=67	k=23
	N-Gram	0.1578	0.9755	27	63 (n=6)	63 (n=6)
	ALERGIA	0.0315	0.991	29	15	15
Tomita 6 (2, 0.78)	WL*	**0.0**	**1.0**	56	3	3
	Spectral	0.0003	0.9999	23	k=36	k=36
	N-Gram	0.1645	0.9695	27	63 (n=6)	63 (n=6)
	ALERGIA	0.0448	0.9983	28	12	12
Tomita 7 (2, 0.78)	WL*	**0.0**	**1.0**	63	5	5
	Spectral	0.0003	0.9999	13	k=32	k=37
	N-Gram	0.0771	0.9857	27	63 (n=6)	63 (n=6)
	ALERGIA	0.0363	0.9936	28	11	11

Table 6.3: Tomita results. Each language is listed with its alphabet size $|\Sigma|$ and RNN test loss ℓ. The n-grams and sample-based PDFAs were created from 50,000 samples, and shared samples. FLEXFRINGE was run with state_count = 50 . Our algorithm was run with t=0.1, $\varepsilon_P, \varepsilon_S$=0, $|P|\leq$5000 and $|S|\leq$100, and spectral with $|P|, |S|$=100.

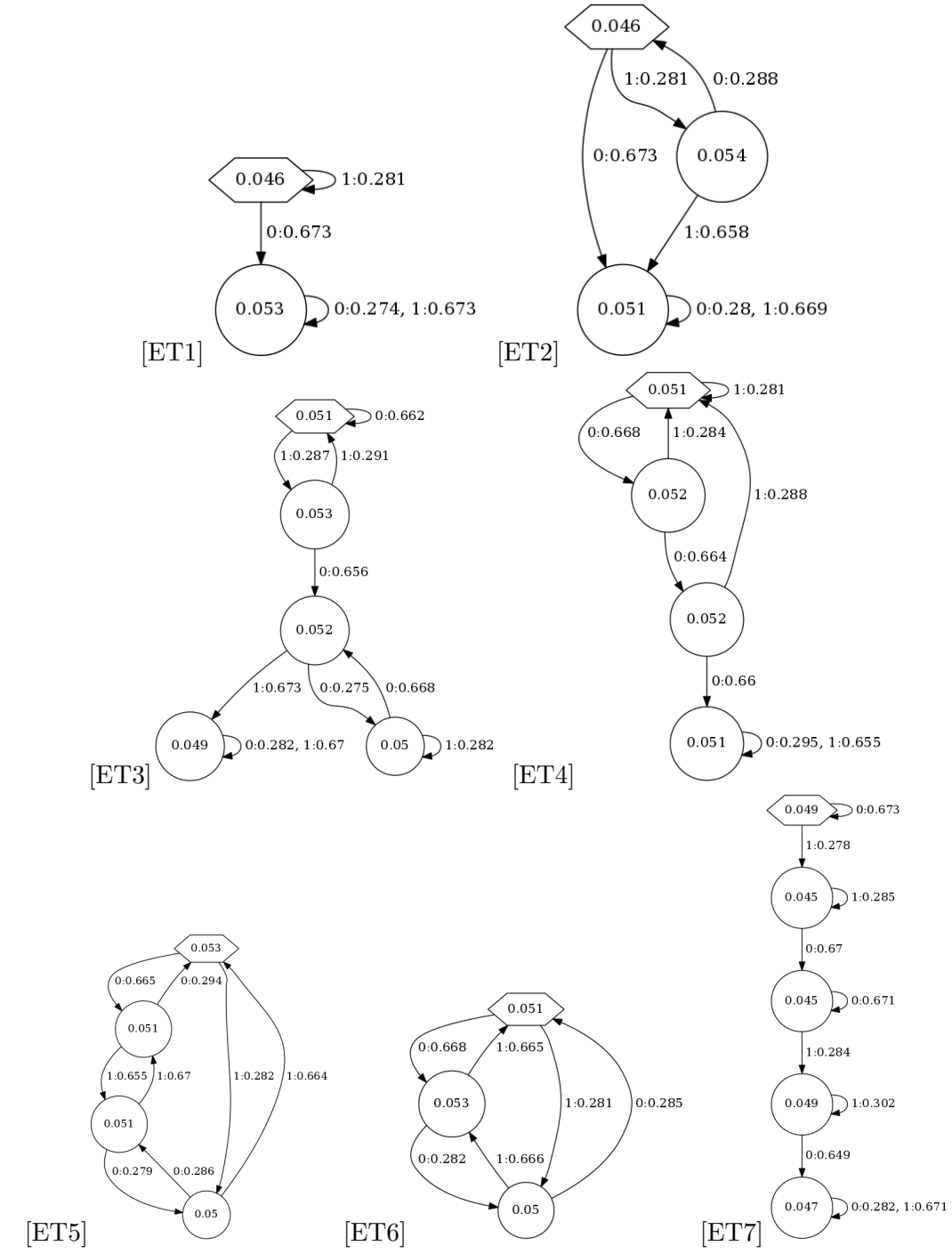


Figure 6.9.2: PDFAs extracted using WL* from the RNNs trained on weighted variants of the Tomita grammars.

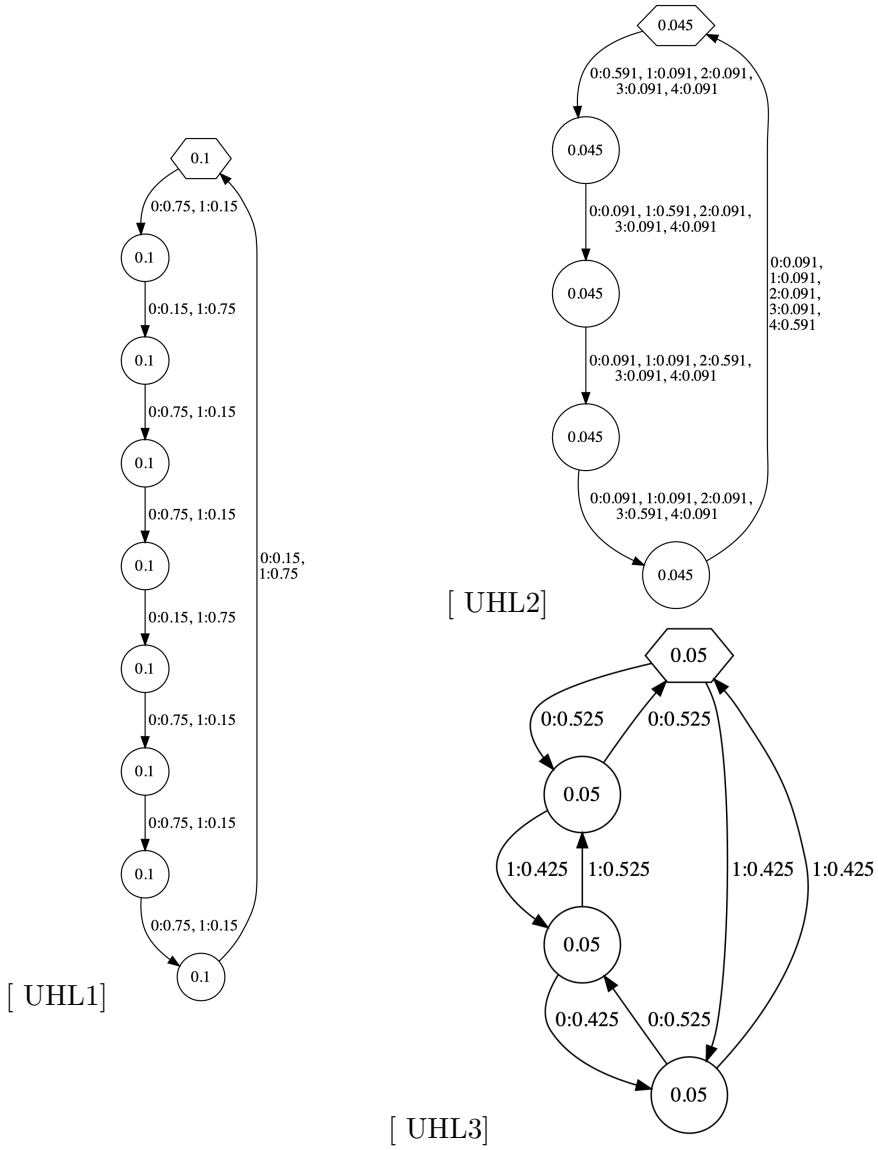


Figure 6.9.3: The UHL PDFAs.

UHL3 is an adaptation of the fifth Tomita grammar similar to our other presented adaptations, except that here the next-token probabilities of `1` and `0` are closer to each other, making it slightly harder to infer which states the PDFA has been in from a finite history[6]

Applied with variation tolerance $t = 0.1$, our algorithm managed to reconstruct every UHLs structure from its trained RNN perfectly, with weights within t of the original[7]. The reconstructed PDFAs are shown in 6.9.4.

[6]This recalls the insight of [SKLV16], who note that unexpected tokens are useful as they convey information about the current state of the model.

[7](When extracting from RNNs, the weights of course can only be as good as those learned by the RNNs)

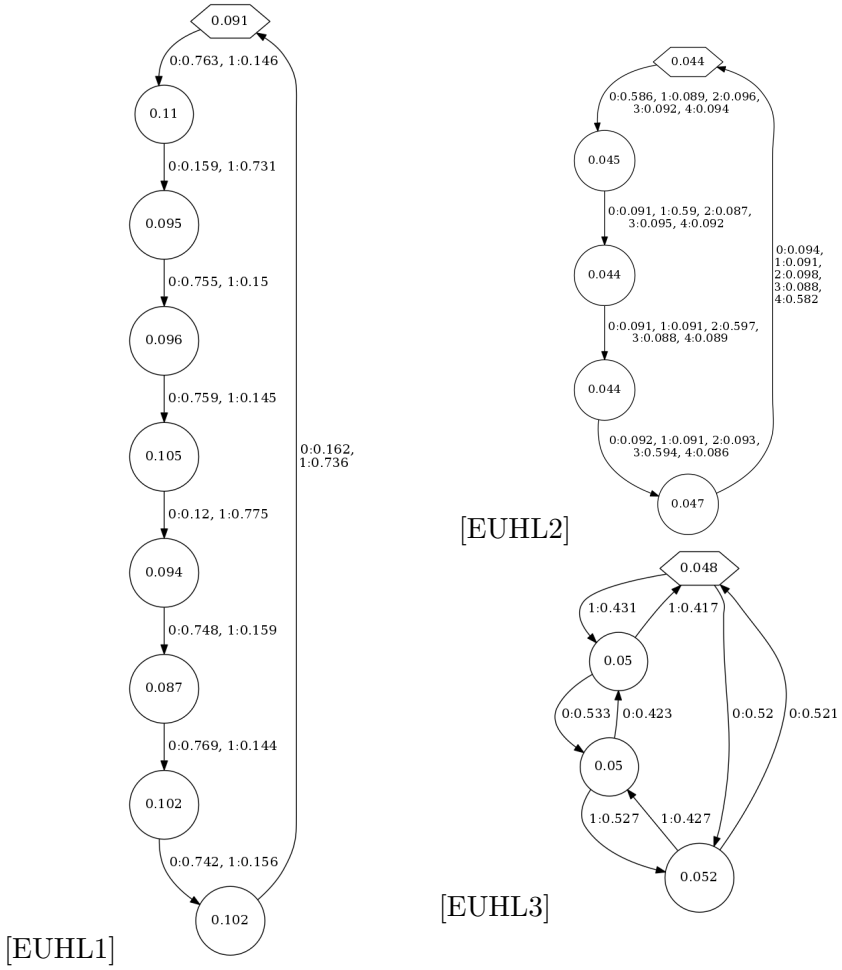


Figure 6.9.4: The UHL PDFAs, as reconstructed by WL* from RNNs trained on the original UHLs.

6.10 Implementation and Training Details

6.10.1 Implementation

Clustering the Prefixes The initial clustering can be done with any clustering algorithm. In our implementation we use ester-etal-2016-dbscan [EKSX96], with t as the noise tolerance and a minimum neighbourhood size 1 for core points. When splitting a cluster into cliques, if its largest range across a single dimension is $n > 1$ times the threshold t, it is split into $\lceil n \rceil$ clusters across that dimension. In the determinism refinement, when splitting a cluster c, there may be some $p \in c$ for which $p{\cdot}\sigma \notin P$. In this case a best match c_σ for $\mathcal{O}_S(p{\cdot}\sigma)$ is found by the heuristic given in section 6.4.2, and p is added to the respective new cluster.

6.10.2 Training Details

All the RNNs are 2-layer pytorch LSTMs with training dropout 0.5 and linear transformation + softmax for the classification. The input token embeddings and initial hidden states were treated as parameters.

The Tomita and UHL RNNs had input (embedding) dimension 2 and hidden dimension 50, except for UHL 2 which had input dimension 5. The SPiCe RNNs had input/hidden dimensions (resp.) as follows: 0. 4/50 1. 20/50 2. 10/50 3. 10/50 4. 33/100 6. 60/100 7. 20/50 9. 11/100 10. 10/20 14. 27/30 .

The RNNs were trained with the ADAM optimiser and varying learning rates, each training for 10 full epochs for learning rate (or less if the validation loss stopped decreasing). The SPiCe and UHL RNNs used a cyclic learning rate, going through 8 values from 0.01 to 0.0001 2 and a half times. The Tomita RNNs simply used the learning rates $0.01, 0.008, 0.006, 0.004, 0.002, 0.001, 0.0005, 0.0001, 5e-05$ once in order.

The SPiCe RNNs were trained with the train samples given by the SPiCe competition [BEL+16]. For the UHL and Tomita RNNs, we generated train sets of size $10,000$ and $20,000$ respectively by sampling from the target PDFAs according to their distributions. For each RNN, we split its given train set into train, validation, and test sets, taking respectively 90%/5%/5% of the original set. We checked each RNN's validation loss after every epoch. Whenever it worsened for 2 consecutive epochs, we reverted to the previous best RNN (by validation loss) and moved to the next learning rate.

For each RNN, in each training epoch we randomly split the train set into batches of equal size (up to the last 'leftover' batch), and trained in these batches. For the UHL and Tomita RNNs we trained with batch size 500 and for the SPiCe RNNs we used $1,000$.

Chapter 7

Thinking Like Transformers

7.1 Introduction

We present a *computational model for the transformer architecture* in the form of a simple language which we dub RASP (*Restricted Access Sequence Processing Language*). Much as the token-by-token processing of RNNs can be conceptualised as finite state automata [CSM89], our language captures the unique information-flow constraints under which a transformer operates as it processes input sequences. Our model helps reason about how a transformer operates at a higher-level of abstraction, reasoning in terms of a composition of *sequence operations* rather than neural network primitives.

We are inspired by the use of automata as an abstract computational model for recurrent neural networks (RNNs). Using automata as an abstraction for RNNs has enabled a long line of work, including extraction of automata from RNNs [OG96; WGY22; AEG18], analysis of RNNs' practical expressive power in terms of automata [WGY18b; RLP19; Mer19; MWG$^+$20], and even augmentations based on automata variants [JM15]. Previous work on transformers explores their computational power, but does not provide a computational model [YBR$^+$20; Hah20; PBM21].

Thinking in terms of the RASP model can help derive computational results. [BAG20] and [EGZ20] explore the ability of transformers to recognise Dyck-k languages, with [BAG20] providing a construction by which Transformer-encoders can recognise a simplified variant of Dyck-k. Using RASP, we succinctly express the construction of [BAG20] as a short program, and further improve it to show, for the first time, that transformers can fully recognise Dyck-k for all k.

Scaling up the complexity, [CTR20] showed empirically that transformer networks can learn to perform multi-step logical reasoning over first order logical formulas provided as input, resulting in "soft theorem provers". For this task, the mechanism of the computation remained elusive: how does a transformer perform even non-soft theorem proving? As the famous saying by Richard Feynman goes, "what I cannot create, I do not understand": using RASP, we were able to write a program that performs similar logical inferences over input expressions, and then "compile" it to the trans-

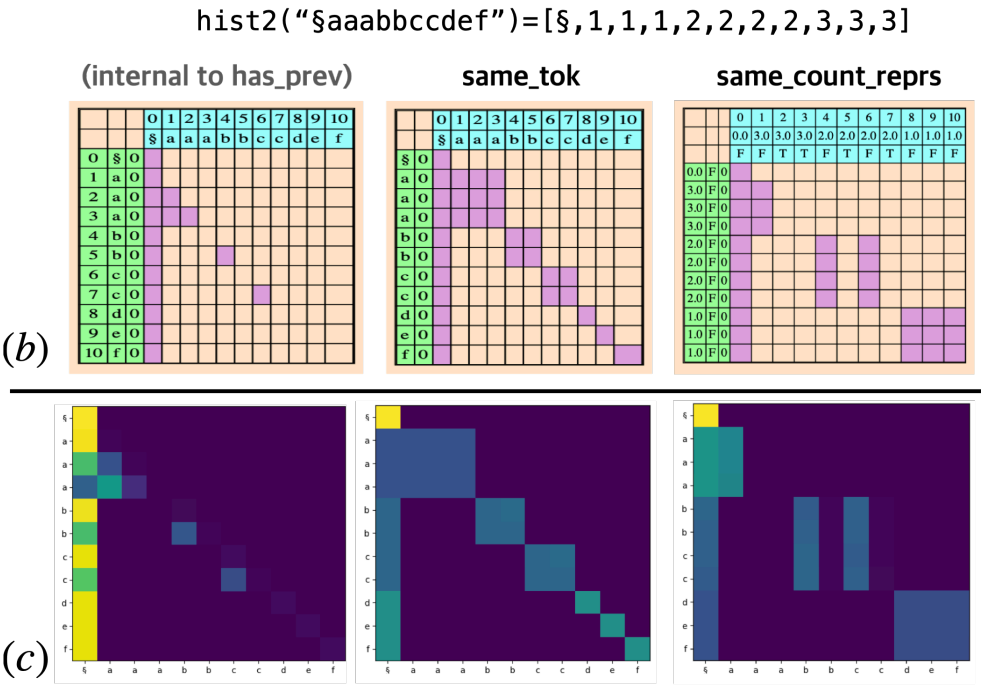
(a)

```
same_tok = select(tokens,tokens,==);
hist = selector_width(same_tok,
                      assume_bos = True);

first = not has_prev(tokens);
same_count = select(hist,hist,==);
same_count_reprs = same_count and
                   select(first,True,==);

hist2 = selector_width(same_count_reprs,
                       assume_bos = True);
```

(b)



(c)

Figure 7.1.1: We consider *double-histogram*, the task of counting for each input token how many unique input tokens have the same frequency as itself (e.g.: `hist2("§aaabbccdef")=[§,1,1,1,2,2,2,2,3,3,3]`). (a) shows a RASP program for this task, (b) shows the selection patterns of that same program, compiled to a transformer architecture and applied to the input sequence `§aaabbccdef`, (c) shows the corresponding attention heatmaps, for the same input sequence, in a 2-layer 2-head transformer trained on double-histogram. This particular transformer was trained using both *target* and *attention* supervision, i.e.: in addition to the standard cross entropy loss on the target output, the model was given an MSE-loss on the difference between its attention heatmaps and those expected by the RASP solution. The transformer reached test accuracy of 99.9% on the task, and comparing the selection patterns in (b) with the heatmaps in (c) suggests that it has also successfully learned to replicate the solution described in (a).

former hardware, defining a sequence of attention and multi-layer perceptron (MLP) operations.

Considering computation problems and their implementations in RASP allows us to "think like a transformer" while abstracting away the technical details of a neural network in favour of symbolic programs. Recognising that a task is representable in a transformer is as simple as finding a RASP program for it, and communicating this solution—previously done by presenting a hand-crafted transformer for the task—is now possible through a few lines of code. Thinking in terms of RASP also allows us to shed light on a recent empirical observation of transformer variants [PSL20], and to find concrete limitations of "efficient transformers" with restricted attention [TDBM20].

In Section 7.5, we show how a compiled RASP program can indeed be realised in a neural transformer (as in Figure 7.1.1), and occasionally is even the solution found by a transformer trained on the task using gradient descent (Figs 7.5.5 and 7.5.4).

Code We provide a RASP read-evaluate-print-loop (REPL) in the repository `http://github.com/tech-srl/RASP`, along with a RASP cheat sheet and link to replication code for our work.

7.2 Overview

We begin with an informal overview of RASP, with examples. The formal introduction is given in Section 7.3.

Intuitively, transformers' computations are applied to their entire input in parallel, using attention to draw on and combine tokens from several positions at a time as they make their calculations [VSP+17; BCB15; LPM15]. The iterative process of a transformer is then not along the length of the input sequence but rather the depth of the computation: the number of layers it applies to its input as it works towards its final result.

The computational model. Conceptually, a RASP computation over length-n input involves manipulation of sequences of length n, and matrices of size $n \times n$. There are no sequences or matrices of different sizes in a RASP computation. The abstract computation model is as follows:

The input of a RASP computation is two sequences, *tokens* and *indices*. The first contains the user-provided input, and the second contains the range $0, 1, ..., n-1$. The output of a RASP computation is a sequence, and the consumer of the output can choose to look only at specific output locations.

Sequences can be transformed into other sequences through element-wise operations. For example, for the sequences $s_1 = [1, 2, 3]$ and $s_2 = [4, 5, 6]$, we can derive $s_1 + s_2 = [5, 7, 9]$, $s_1 + 2 = [3, 4, 5]$, $pow(s_1, 2) = [1, 4, 9]$, $s_1 > 2 = [F, F, T]$, pairwise_mul$(s_1, s_2) = [4, 10, 18]$, and so on.

Sequences can also be transformed using a pair of *select* and *aggregate* operations (Figure 7.2.2). Select operations take two sequences k, q and a boolean predicate p

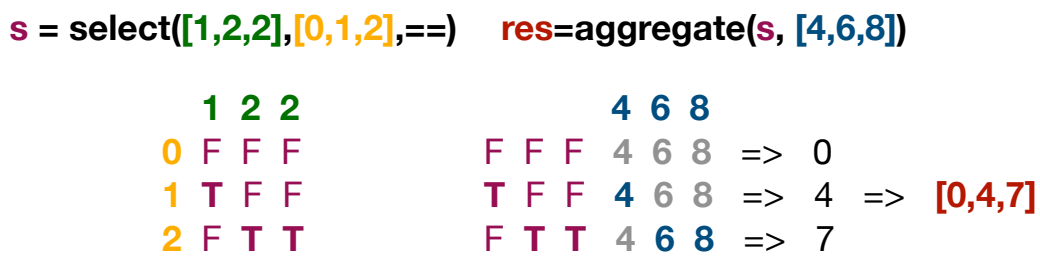


Figure 7.2.2: Visualising the select and aggregate operations. On the left, a selection matrix **s** is computed by select, which marks for each **query** position all of the **key** positions with matching values according to the given comparison operator `==`. On the right, aggregate uses **s** as a filter over its input **values**, averaging only the selected **values** at each position in order to create its output, **res**. Where no **values** have been selected, aggregate substitutes 0 in its output.

over pairs of values, and return a *selection matrix* S such that for every $i, j \in [n]$, $S_{[i][j]} = p(k_{[i]}, q_{[j]})$. Aggregate operations take a matrix S and a numeric sequence v, and return a sequence s in which each position $s_{[i]}$ combines the values in v according to row i in S (see full definition in Section 7.3).

Aggregate operations (over select matrices) are the only way to combine values from different sequence positions, or to move values from one position to another. For example, to perform the python computation: `x = [a[0] for _ in a]`, we must first use $S = \text{select}(\textit{indices}, 0, =)$ to select the first position, and then $x = \text{aggregate}(S, a)$ to broadcast it across a new sequence of the same length.

RASP programs are lazy functional, and thus operate on functions rather than sequences. That is, instead of a sequence indices$= [0, 1, 2]$, we have a function `indices` that returns $[0, 1, 2]$ on inputs of length 3. Similarly, `s3=s1+s2` is a function, that when applied to an input x will produce the value `s3`(x), which will be computed as `s1`(x)+`s2`(x). We call these functions *s-ops (sequence operators)*. The same is true for the selection matrices, whose functions we refer to as *selectors*, and the RASP language is defined in terms of s-ops and selectors, not sequences and matrices. However, the conceptual model to bear in mind is that of operations over sequences and selection matrices.

Example: Double Histograms The RASP program in Figure 7.1.1 solves *double-histogram*, the task of counting for each token how many unique input tokens in the sequence have the same frequency as its own. More precisely, it implements the slightly easier case where there is also a unique Beginning of Sequence (BOS) token `§` that can be ignored:[1]

```
hist2("§aabcd")=[§,1,1,3,3,3]
```

[1]This manifests in the use of a slightly simpler `selector_width` function.

The program begins by creating the the selector `same_tok`, in which each input position focuses on all other positions containing the same token as its own, and then applies the RASP operation `selector_width` to it in order to obtain the s-op `hist`, which computes the frequency of each token in the input:

```
hist("§aabcd")=[§,2,2,1,1,1]
```

Next, the program uses the function `has_prev`[2] to create the s-op `first`, which marks the first appearance of each token in a sequence:

```
first("§aabcd")=[T,T,F,T,T,T]
```

Finally, applying `selector_width` to the selector `same_count_reprs`, which focuses each position on all 'first' tokens with the same frequency as its own, provides `hist2` as desired.

Example: Shuffle-Dyck in RASP As an example of the kind of tasks that are natural to encode using RASP, consider the Shuffle-Dyck language, in which multiple parentheses types must be balanced but do not have to satisfy any order with relation to each other. (For example, `"([)]"` is considered balanced). In their work on transformer expressiveness, [BAG20] present a hand-crafted transformer for this language, including the details of which dimension represents which partial computation. RASP can concisely describe the same solution, showing the high-level operations while abstracting away the details of their arrangement into an actual transformer architecture.

We present this solution in Figure 7.2.3: the code compiles to a transformer architecture using 2 layers and a total of 3 heads, exactly as in the construction of [BAG20]. These numbers are inferred by the RASP compiler: the programmer does not have to think about such details.

A pair of parentheses is balanced in a sequence if their running balance is never negative, and additionally is equal to exactly 0 at the final input token. Lines 13–23 check this definition: lines 13 and 14 use `pair_balance` to compute the running balances of each parenthesis pair, and 17 checks whether these balances were negative anywhere in the sequence. The snippet in 21 (`bal1==0 and bal2==0`) creates an s-op checking at each location whether both pairs are balanced, with the aggregation of line 20 loading the value of this s-op from the last position. From there, a boolean composition of `end_0` and `had_neg` defines shuffle-dyck-2.

Compilation and Abstraction The high-level operations in RASP can be compiled down to execute on a transformer: for example, the code presented in Figure 7.1.1 compiles to a two-layer, 3-head (total) architecture, whose attention patterns when applied to the input sequence `"§aaabbccdef"` are presented in Figure 7.1.1(b). (The full compiled computation flow for this program—showing how its component s-ops interact—is presented in Section 7.7).

RASP abstracts away low-level operations into simple primitives, allowing a programmer to explore the full potential of a transformer without getting bogged down

[2]Presented in Figure 7.7.12 in Section 7.7.

```
def frac_prevs(sop,val){
  prevs = select(indices,indices,<=);
  return aggregate(prevs,indicator(sop == val
     ));
}

def pair_balance(open,close) {
  opens = frac_prevs(tokens,open);
  closes = frac_prevs(tokens,close);
  return opens - closes;
}

bal1 = pair_balance("(",")");
bal2 = pair_balance("{","}");

negative = bal1<0 or bal2<0;
had_neg = aggregate(select_all,
                    indicator(negative))>0;
select_last = select(indices,length-1,==);
end_0 = aggregate(select_last,bal1==0 and
   bal2==0);

shuffle_dyck2 = end_0 and not had_neg;
```

Figure 7.2.3: RASP program for the task shuffle-dyck-2 (balance 2 parenthesis pairs, independently of each other), capturing a higher level representation of the hand-crafted transformer presented by [BAG20].

in the details of how these are realised in practice. At the same time, RASP enforces the information-flow constraints of transformers, preventing anyone from writing a program more powerful than they can express. One example of this is the lack of input-dependent loops in the s-ops, reflecting the fact that transformers cannot arbitrarily repeat operations[3]. Another is in the selectors: for each two positions, the decision whether one selects ('attends to') the other is pairwise.

We find RASP a natural tool for conveying transformer solutions to given tasks. It is modular and compositional, allowing us to focus on arbitrarily high-level computations when writing programs. Of course, we are restricted to tasks for which a human *can* encode a solution: we do not expect any researcher to implement, e.g., a strong language model or machine-translation system in RASP—these are not realisable in any programming language. Rather, we focus on programs that convey concepts that people can encode in "traditional" programming languages, and the way they relate to the expressive power of the transformer.

In Section 7.5, we will show empirically that RASP solutions can indeed translate to real transformers. One example is given in Figure 7.1.1: having written a RASP

[3]Though work exploring such transformer variants exists: [DGV+19] devise a transformer architecture with a control unit, which can repeat its sublayers arbitrarily many times.

program (left) for the double-histograms task, we analyse it to obtain the number of layers and heads needed for a transformer to mimic our solution, and then train a transformer with supervision of both its outputs and its attention patterns to obtain a neural version of our solution (right). We find that the transformer can accurately learn the target attention patterns and use them to reach a high accuracy on the target task.

7.3 The RASP language

RASP contains a small set of primitives and operations built around the core task of manipulating sequence processing functions referred to as *s-ops (sequence operators)*, functions that take in an input sequence and return an output sequence of the same length. Excluding some atomic values, and the convenience of lists and dictionaries, *everything* in RASP is a function. Hence, to simplify presentation, we often demonstrate RASP values with one or more input-output pairs: for example

`identity("hi") = "hi"`[4].

RASP has a small set of built-in s-ops, and the goal of programming in RASP is to compose these into a final s-op computing the target task. For these compositions, the functions `select` (creating selection matrices called *selectors*), `aggregate` (collapsing selectors and s-ops into a new s-ops), and `selector_width` (creating an s-op from a selector) are provided, along with several elementwise operators reflecting the feed-forward sublayers of a transformer. As noted in Section 7.2, while all s-ops and selectors are in fact functions, we will prefer to talk in terms of the sequences and matrices that they create. Constant values in RASP (e.g., 2, T, `h`) are treated as s-ops with a single value broadcast at all positions, and all symbolic values are assumed to have an underlying numerical representation which is the value being manipulated in practice.

The built-in (base) s-ops The simplest s-op is the identity, given in RASP under the name `tokens`: `tokens("hi")="hi"`. The other built-in s-ops are `indices` and `length`, processing input sequences as their names suggest:

```
indices("hi") = [0,1]
length("hi") = [2,2]
```

s-ops can be combined with constants (numbers, booleans, or tokens) or each other to create new s-ops, in either an elementwise or more complicated fashion.

Elementwise combination of s-ops is done by the common operators for the values they contain, for example:

```
(indices+1)("hi") = [1,2]
((indices+1)==length)("hi") = [F,T]
```

This includes also a ternary operator:

[4]We use strings as shorthand for a sequence of characters.

```
(tokens if (indices%2==0) else "-")("hello")="h-l-o"
```

When the condition of the operator is an s-op itself, the result is an s-op that is dependent on all 3 of the terms in the operator creating it.

Select and Aggregate operations are used to combine information from different sequence positions. A selector takes two lists, representing *keys* and *queries* respectively, and a predicate p, and computes from these a selection matrix describing for each key, query pair (k, q) whether the condition $p(k, q)$ holds.

For example:

$$sel([0,1,2],[1,2,3],<) = \begin{bmatrix} \mathbf{T} & F & F \\ \mathbf{T} & \mathbf{T} & F \\ \mathbf{T} & \mathbf{T} & \mathbf{T} \end{bmatrix}$$

An aggregate operation takes one selection matrix and one list, and averages for each row of the matrix the values of the list in its selected columns. For example,

$$agg(\begin{bmatrix} \mathbf{T} & F & F \\ \mathbf{T} & \mathbf{T} & F \\ \mathbf{T} & \mathbf{T} & \mathbf{T} \end{bmatrix}, [10,20,30]) = [10,15,20]$$

Intuitively, a select-aggregate pair can be thought of as a two-dimensional map-reduce operation. The selector can be viewed as performing filtering, and aggregate as performing a reduce operation over the filtered elements (see Figure 7.2.2).

In RASP, the selection operation is provided through the function `select`, which takes two s-ops `k` and `q` and a comparison operator $\circ$ and returns the composition of sel$(\cdot, \cdot, \circ)$ with `k` and `q`, with this sequence-to-matrix function referred to as a *selector*. For example:

```
a = select(indices,indices,<);
```

is a selector, and

$$\texttt{a("hey")} = \begin{bmatrix} F & F & F \\ \mathbf{T} & F & F \\ \mathbf{T} & \mathbf{T} & F \end{bmatrix}$$

Similarly, the aggregation operation is provided through `aggregate`, which takes one selector and one s-op and returns the composition of agg with these. For example:

```
aggregate(a,indices+1)("hey")=[0,1,1.5]
```
[5]

[5]For convenience and efficiency, when averaging the filtered values in an aggregation, for every position where only a single value has been selected, RASP passes that value directly to the output without attempting to 'average' it. This saves the programmer from unnecessary conversion into and out of numerical representations when making simple transfers of tokens between locations: for example, using the selector `load1=select(indices,1,==)`, we may directly create the s-op `aggregate(load1,tokens)("hey")="eee"`. Additionally, in positions when no values are selected, the aggregation simply returns a default value for the output (in Figure 7.2.2, we see this with default value 0), this value may be set as one of the inputs to the `aggregate` function.

Simple select-aggregate examples To create the s-op that reverses any input sequence, we build a selector that requests for each query position the token at the opposite end of the sequence, and then aggregate that selector with the original input tokens:

```
flip = select(indices,length-indices-1,==);
reverse = aggregate(flip,tokens);
```

For example:

$$\texttt{flip("hey")} = \begin{bmatrix} F & F & \mathbf{T} \\ F & \mathbf{T} & F \\ \mathbf{T} & F & F \end{bmatrix}$$

```
reverse("hey") = "yeh"
```

To compute the fraction of appearances of the token `"a"` in our input, we build a selector that gathers information from *all* input positions, and then aggregate it with a sequence broadcasting 1 wherever the input token is `"a"`, and 0 everywhere else. This is expressed as

```
select_all = select(1,1,==);
frac_as = aggregate(select_all,1 if tokens=="a" else 0);
```

Selector manipulations Selectors can be combined elementwise using boolean logic. For example, with the same `flip` from above:

```
load1=select(indices,1,==);
```

$$\texttt{(load1 or flip)("hey")} = \begin{bmatrix} F & \mathbf{T} & \mathbf{T} \\ F & \mathbf{T} & F \\ \mathbf{T} & \mathbf{T} & F \end{bmatrix}$$

Selector width The final operation in RASP is the powerful `selector_width`, which takes as input a single selector and returns a new s-op that computes, for each output position, the number of input values which that selector has chosen for it. This is best understood by example: using the selector

```
same_token=select(tokens,tokens,==);
```

that filters for each query position the keys with the same token as its own, we can compute its width to obtain a histogram of our input sequence:

```
selector_width(same_token)("hello") = [1,1,2,2,1]
```

Additional operations: While the above operations are together sufficient to represent any RASP program, RASP further provides a library of primitives for common operations, such as `in`—either of a value within a sequence:

```
("i" in tokens)("hi") = [T,T]
```

or of each value in a sequence within some static list:

```
(tokens in ["a","b","c"])("hat") = [F,T,F]
```

RASP also provides functions such as `count`, or `sort`.

7.3.1 Relation to a Transformer

We discuss how the RASP operations compile to describe the information flow of a transformer architecture, suggesting how many heads and layers are needed to solve a task.

The built-in (base) s-ops `indices` and `tokens` reflect the initial input embeddings of a transformer, while `length` is actually a library s-op computed in RASP as follows:

```
select_all = select(1,1,==);
length = 1/aggregate(select_all,indicator(indices==0));
```

Elementwise Operations reflect the feed-forward sub-layers of a transformer. These have overall not been restricted in any meaningful way: as famously shown by [HSW89a], MLPs such as those present in the feed-forward transformer sub-layers can approximate with arbitrary accuracy any borel-measurable function, provided sufficiently large input and hidden dimensions.

Selection and Aggregation Selectors translate to attention matrices, defining for each input the selection (attention) pattern used to mix the input values into a new output through weighted averages, and aggregation reflects this final averaging operation. The uniform weights dictated by our selectors reflect an attention pattern in which 'unselected' pairs are all given strongly negative scores, while the selected pairs all have higher, similar, scores. Such attention patterns are supported by the findings of [MRG+20].

Decoupling selection and aggregation in RASP allows `selectors` to be reused in multiple aggregations, abstracting away the fact that these may actually require separate attention heads in the compiled architecture. Making selectors first class citizens also enables functions such as `selector_width`, which take selectors as parameters.

Additional abstractions All other operations, including the powerful `selector_width` operation, are implemented in terms of the above primitives. `selector_width` in particular can be implemented such that it compiles to either one or two selectors, depending on whether or not one can assume a BOS token (marked `§`) is added to the input sequence. Its implementation is given in Section 7.7.

Compilation Converting an s-op to a transformer architecture is as simple as tracing its computation flow out from the base s-ops. Each aggregation is an attention head, which must be placed at a layer later than all of its inputs. Elementwise operations are feedforward operations, and sit in the earliest layer containing all of their dependencies. Some optimisations are possible: for example, aggregations performed at the same layer with the same selector can be merged into the same attention head. A "full" compilation—to concrete transformer weights—requires to e.g. derive MLP

weights for the elementwise operations, and is beyond the scope of this work. RASP provides a method to visualise this compiled flow for any s-op and input pair: Figures 7.5.4 and 7.5.5 were rendered using

```
draw(reverse,"abcde")
```

and

```
draw(hist,"§aabbaabb")
```

7.4 Implications and Insights

Restricted-Attention Transformers Multiple works propose restricting the attention mechanism to create more efficient transformers, reducing the time complexity of each layer from $O(n^2)$ to $O(nlog(n))$ or even $O(n)$ with respect to the input sequence length n (see [TDBM20] for a survey of such approaches). Several of these do so using *sparse attention*, in which the attention is masked using different patterns to reduce the number of locations that can interact ([CGRS19; BPC20; AOA+20; ZGD+20; RSVG21]).

Considering such transformer variants in terms of RASP allows us to reason about the computations they can and cannot perform. In particular, these variants of transformers all impose restrictions on the selectors, permanently forcing some of the n^2 index pairs in every selector to `False`. But does this necessarily weaken these transformers?

In Section 7.7 we present a sorting algorithm in RASP, applicable to input sequences with arbitrary length and alphabet size[6]. This problem is known to require at $\Omega(n \log(n))$ operations in the input length n—implying that a standard transformer can take full advantage of $\Omega(n \log(n))$ of the n^2 operations it performs in every attention head. It follows from this that all variants restricting their attention to $o(n \log(n))$ operations incur a real loss in expressive power.

Sandwich Transformers Recently, [PSL20] showed that reordering the attention and feed-forward sublayers of a transformer affects its ability to learn language modelling tasks. In particular, they showed that: 1. pushing feed-forward sublayers towards the bottom of a transformer weakened it; and 2. pushing attention sublayers to the bottom and feed-forward sublayers to the top strengthened it, provided there was still some interleaving in the middle.

The base operations of RASP help us understand the observations of [PSL20]. Any arrangement of a transformer's sublayers into a fixed architecture imposes a restriction on the number and order of RASP operations that can be chained in a program compilable to that architecture. For example, an architecture in which all feed-forward sublayers appear before the attention sublayers, imposes that no elementwise operations may be applied to the result of any aggregation.

[6] Of course, realising this solution in real transformers requires sufficiently stable word and positional embeddings—a practical limitation that applies to all transformer variants.

In RASP, there is little value to repeated elementwise operations before the first `aggregate`: each position has only its initial input, and cannot generate new information. This explains the first observation of [PSL20]. In contrast, an architecture beginning with several attention sublayers—i.e., multiple `select-aggregate` pairs—will be able to gather a large amount of information into each position early in the computation, even if only by simple rules[7]. More complicated gathering rules can later be realised by applying elementwise operations to aggregated information before generating new selectors, explaining the second observation.

Recognising Dyck-k Languages The Dyck-k languages—the languages of sequences of correctly balanced parentheses, with k parenthesis types—have been heavily used in considering the expressive power of RNNs [SB18; STK18; Ber18; Mer19; HHG+20].

Such investigations motivate similar questions for transformers, and several works approach the task. [Hah20] proves that transformer-encoders with hard attention cannot recognise Dyck-2. [BAG20] and [YPPN21] provide transformer-encoder constructions for recognising simplified variants of Dyck-k, though the simplifications are such that no conclusion can be drawn for unbounded depth Dyck-k with $k > 1$. Optimistically, [EGZ20] train a transformer-encoder with causal attention masking to process Dyck-k languages with reasonable accuracy for several $k > 1$, finding that it learns a stack-like behaviour to complete the task.

We consider Dyck-k using RASP, specifically defining Dyck-k-PTF as the task of classifying for every prefix of a sequence whether it is legal, but not yet balanced (**P**), balanced (**T**), or illegal (**F**). We show that RASP can solve this task in a fixed number of heads and layers for *any* k, presenting our solution in Section 7.7[8].

Symbolic Reasoning in Transformers [CTR20] show that transformers are able to emulate symbolic reasoning: they train a transformer which, given the facts "Ben is a bird" and "birds can fly", correctly validates that "Ben can fly". Moreover, they show that transformers are able to perform several logical 'steps': given also the fact that only winged animals can fly, their transformer confirms that Ben has wings. This finding however does not shed any light on *how* the transformer is achieving such a feat.

RASP empowers us to approach the problem on a high level. We write a RASP program for the related but simplified problem of containment and inference over sets of elements, sets, and logical symbols, in which the example is written as `b∈B, x∈B→x∈F, b∈F?` (implementation available in our repository). The main idea is to store at the position of each set symbol the elements contained and not contained in that set, and

[7]While the attention sublayer of a transformer does do some local manipulations on its input to create the candidate output vectors, it does not contain the powerful MLP with hidden layer as is present in the feed-forward sublayer.

[8]We note that RASP does not suggest the embedding width needed to encode this solution in an actual transformer.

at each element symbol the sets it is and is not contained in. Logical inferences are computed by passing information between symbols in the same 'fact', and propagated through pairs of identical set or element symbols, which share their stored information.

Use of Separator Tokens [CKLM19] observe that many attention heads in BERT [DCLT18] (sometimes) focus on separator tokens, speculating that these are used for "no-ops" in the computation. [EGZ20] find that transformers more successfully learn Dyck-k languages when the input is additionally provided with a beginning-of-sequence (BOS) token, with the trained models treating it as a base in their stack when there are no open parentheses. Our RASP programs suggest an additional role that such separators may be playing: by providing a fixed signal from a 'neutral' position, separators facilitate conditioned counting in transformers, that use the diffusion of the signal to compute how many positions a head was attending to. Without such neutral positions, counting requires an additional head, such that an agreed-upon position may artificially be treated as neutral in one head and then independently accounted for in the other.

A simple example of this is seen in Figure 7.5.5. There, `selector_width` is applied with a BOS token, creating in the process an attention pattern that focuses on the first input position (the BOS location) from all query positions, in addition to the actual positions selected by `select(tokens,tokens,==)`. A full description of `selector_width` is given in Section 7.7.

7.5 Experiments

We evaluate the relation of RASP to transformers on three fronts: 1. its ability to upper bound the number of heads and layers required to solve a task, 2. the tightness of that bound, and 3. its feasibility in a transformer, i.e., whether a sufficiently large transformer can encode a given RASP solution. To do this we train several transformers, we present the exact details of these transformers and their training in Section 7.6.

For this section, we consider the following tasks:

1. Reverse, e.g.:

   `reverse("abc")="cba"`

2. Histograms, with a unique (and ignored) BOS token `§` and without it. E.g.:

   `hist_bos("§aba")=[§,2,1,2]`

   `hist_nobos("aba")=[2,1,2]`

3. Double-Histograms, with BOS: for each token, the number of unique tokens with same histogram value as itself. E.g.:

Language	Layers	Heads	Test Acc.	Attn. Matches?
Reverse	2	1	99.99%	✓
Hist BOS	1	1	100%	✓
Hist no BOS	1	2	99.97%	✓
Double Hist	2	2	99.58%	✓
Sort	2	1	99.96%	✗
Most Freq	3	2	95.99%	✗
Dyck-1 PTF	2	1	99.67%	✓
Dyck-2 PTF [9]	3	1	99.85%	✗

Table 7.1: Does a RASP program correctly upper bound the number of heads and layers needed for a transformer to solve a task? In the left columns, we show the compilation size of our RASP programs for each considered task, and in the right columns we show the best (of 4) accuracies of transformers trained on these same tasks, and evaluate whether their attention mechanisms appear to match (using a ✓ for partially similar patterns: see Figure 7.5.4 for an example). For RASP programs compiling to varying number of heads per layer, we report the maximum of these.

```
hist2("§abbc")=[§,2,1,1,2]
```

4. Sort, with BOS: ordering the input tokens lexicographically. E.g.:

```
sort("§cba")="§abc"
```

5. Most-Freq, with BOS: returning the unique input tokens in order of decreasing frequency, with original position as a tie-breaker and the BOS token for padding. E.g.:

```
most_freq("§abbccddd")="§dbca§§§§"
```

6. Dyck-i PTF, for $i = 1, 2$: the task of returning, at each output position, whether the input prefix up to and including that position is a legal Dyck-i sequence (`T`), and if not, whether it can (`P`) or cannot (`F`) be continued into a legal Dyck-i sequence. E.g:

```
Dyck1_ptf("()())")="PTPTF"
```

We refer to double-histogram as 2-hist, and to each Dyck-i PTF problem simply as Dyck-i. The full RASP programs for these tasks, and the computation flows they compile down to, are presented in Section 7.7. The size of the transformer architecture each task compiles to is presented in Table 7.1.

[9] The actual optimal solution for Dyck-2 PTF cannot be realised in RASP as is, as it requires the addition of a select-*best* operator to the language—reflecting the power afforded by softmax in the transformer's self-attention. In this paper, we always refer to our analysis of Dyck-2 with respect to this additional operation.

```
opp_index = length - indices - 1;
flip = select(indices, opp_index,==);
reverse = aggregate(flip, tokens);
```

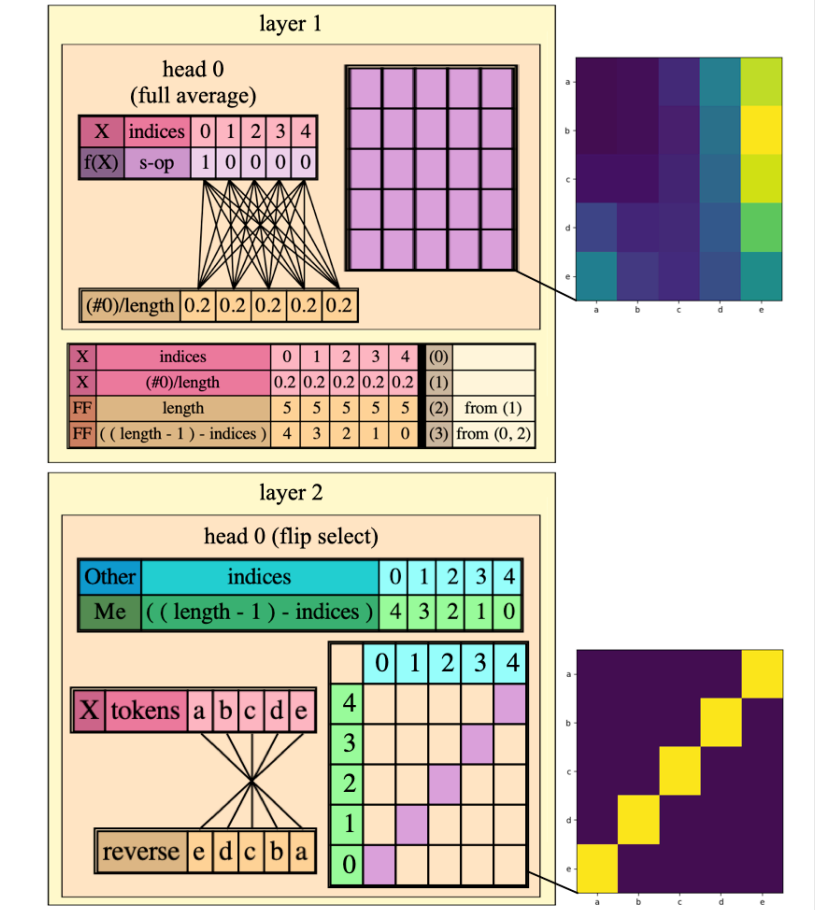


Figure 7.5.4: Top: RASP code for computing reverse (e.g., `reverse("abc")="cba"`). Below, its compilation to a transformer architecture (left, obtained through `draw(reverse,"abcde")` in the RASP REPL), and the attention heatmaps of a transformer trained on the same task (right), both visualised on the same input. Visually, the attention head in the second layer of this transformer corresponds perfectly to the behaviour of the `flip` selector described in the program. The head in the first layer, however, appears to have learned a different solution from our own: instead of focusing uniformly on the entire sequence (as is done in the computation of `length` in RASP), this head shows a preference for the last position in the sequence.

Upper bounding the difficulty of a task Given a RASP program for a task, e.g. double-histogram as described in Figure 7.1.1, we can compile it down to a transformer architecture, effectively predicting the maximum number of layers and layer width (number of heads in a layer) needed to solve that task in a transformer. To evaluate whether this bound is truly sufficient for the transformer, we train 4 transformers of the prescribed sizes on each of the tasks.

```
same_tok = select(tokens, tokens, ==);
hist = selector_width(same_tok,
                      assume_bos = True);
```

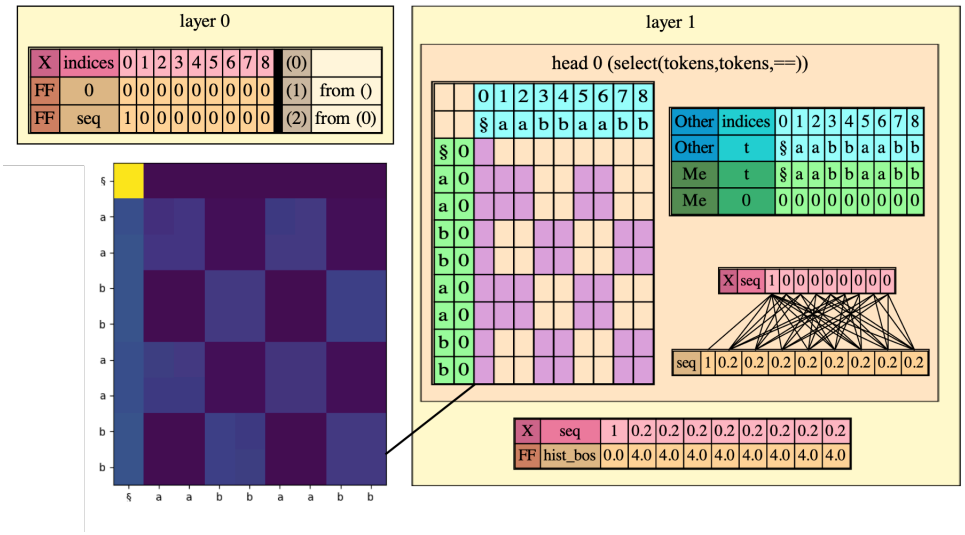


Figure 7.5.5: The RASP program for computing with-BOS histograms (top), alongside its compilation to a transformer architecture (cream boxes labelled *layer 0* and *layer 1*) and the attention head (bottom) of a transformer trained on the same task, without attention supervision. The compiled architecture and the trained head are both presented on the same input sequence, `"§aabbaabb"`. The transformer architecture was generated in the RASP REPL using `draw(hist,"§aabbaabb")`.

We report the accuracy of the best trained transformer for each task in Table 7.1. Most of these transformers reached accuracies of 99.5% and over, suggesting that the upper bounds obtained by our programs are indeed sufficient for solving these tasks in transformers. For some of the tasks, we even find that the RASP program is the same as or very similar to the 'natural' solution found by the trained transformer. In particular, Figures 7.5.4 and 7.5.5 show a strong similarity between the compiled and learned attention patterns for the tasks Reverse and Histogram-BOS, though the transformer trained on Reverse appears to have learned a different mechanism for computing `length` than that given in RASP.

Tightness of the bound We evaluate the tightness of our RASP programs by training smaller transformers than those predicted by our compilation, and observing the drop-off in test accuracy. Specifically, we repeat our above experiments, but this time we also train each task on up to 4 different sizes. In particular, denoting L, H the number of layers and heads predicted by our compiled RASP programs, we train for each task transformers with sizes (L, H), $(L-1, H)$, $(L, H-1)$, and $(L-1, 2H)$ (where possible).[10]

[10]The transformers of size $(L-1, 2H)$ are used to validate that any drop in accuracy is indeed due to the reduction in number of layers, as opposed to the reduction in total heads that this entails. However, doubling H means the embedding dimension will be divided over twice as many heads. To counteract

Language	RASP	Average test accuracy (%) with...			
	L, H	L, H	$H-1$	$L-1$	$L-1, 2H$
Reverse	$2, 1$	**99.9**	-	23.1	41.2
Hist	$1, 2$	**99.9**	91.9	-	-
2-Hist	$2, 2$	**99.0**	73.5	40.5	83.5
Sort	$2, 1$	99.8	-	99.0	**99.9**
Most Freq	$3, 2$	**93.9**	92.1	84.0	90.2
Dyck-1	$2, 1$	**99.3**	-	96.9	96.4
Dyck-2	$3, 1$	**99.7**	-	98.8	94.1

Table 7.2: Accuracy dropoff in transformers when reducing their number of heads and layers relative to the compiled RASP solutions for the same tasks. The transformers trained on the size predicted by RASP have very high accuracy, and in most cases there is a clear drop as that size is reduced. Cases creating an impossible architecture (H or L zero) are marked with -. Histogram with BOS uses only 1 layer and 1 head, and so is not included. As in Table 7.1, Dyck-2 is considered with the addition of `select_best` to RASP.

We report the average test accuracy reached by each of these architectures in Table 7.2. For most of the tasks, the results show a clear drop in accuracy as the number of heads or layers is reduced below that obtained by our compiled RASP solutions for the same tasks—several of these reduced transformers fail completely to learn their target languages.

The main exception to this is sort, which appears unaffected by the removal of one layer, and even achieves its best results in this case. Drawing the attention pattern for the single-layer sort transformers reveals relatively uniform attention patterns. It appears that the transformer has learned to take advantage of the bounded input alphabet size, effectively implementing bucket sort for its task. This is because a single full-attention head is sufficient to compute for every token its total appearances in the input, from which the correct output can be computed locally at every position.

Feasibility of a RASP program We verify that a given RASP program can indeed be represented in a transformer. For this, we return to the tougher tasks above, and this time train the transformer with an additional loss component encouraging it to learn the attention patterns created in our compiled solution (i.e., we supervise the attention patterns in addition to the target output). In particular, we consider the tasks double-histogram, sort, and most-freq, all with the assumption of a BOS token in the input. After training each transformer for 250 epochs with both target and attention supervision, they all obtain high test accuracies on the task (99+%), and appear to encode attention patterns similar to those compiled from our solutions. We present the obtained patterns for double-histogram, alongside the compiled RASP solution, in Figure 7.1.1. We present its full computation flow, as well as the learned attention patterns and full flow of sort and most-freq, in Section 7.6.

any negative effect this may have, we also double the embedding dimension for these transformers.

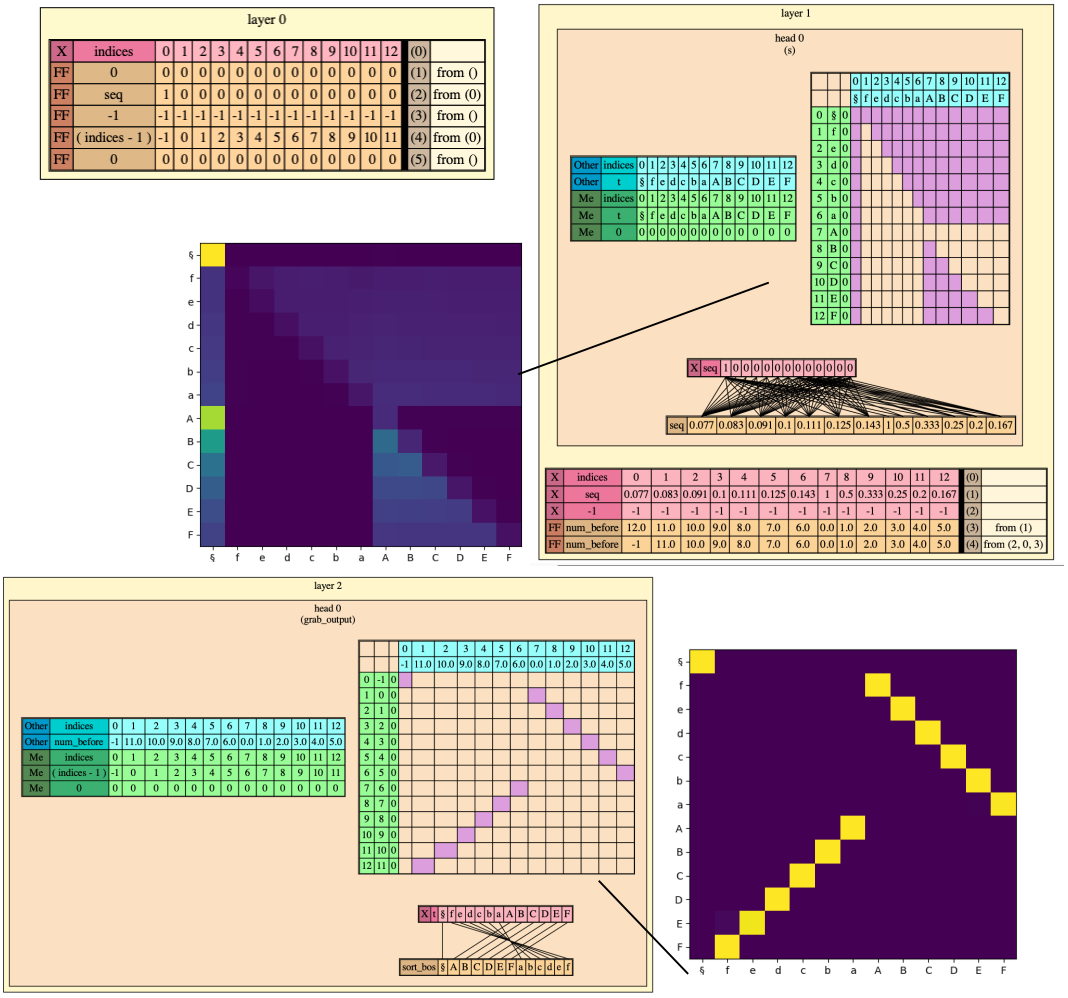


Figure 7.6.6: Computation flow in compiled architecture from RASP solution for sort (with BOS token), alongside heatmaps from the corresponding heads in a transformer trained with both target and attention supervision on the same task and RASP solution. The RASP solution is simply written `sort(tokens,tokens,assume_bos=True)`, using the function `sort` shown in Figure 7.7.15. Both the RASP architecture and the transformer are applied to the input sequence `"§fedcbaABCDEF"`.

7.6 Experiment Details and Additional Results

In this section we present the training details of the experiments in this work, as well as additional results from the transformers trained to mimic RASP-predicted attention patterns.

7.6.1 Results: Attention-regularised transformers

We trained 3 transformers with a target attention pattern according to our RASP solutions, these 3 being for the tasks double-histogram, sort, and most-freq as described in the paper. All of these reached high (99+%) accuracy on their sequence-to-sequence

task, computed as fraction of output tokens predicted correctly. Plotting their attention patterns also shows clear similarity to those of the compiled RASP programs:

For the *double-histogram* task, a full compiled architecture is presented on the sequence `§aabbaa` in Figure 7.7.17. Additionally, in Figure 7.1.1, just its attention patterns are presented alongside the corresponding attention heads from its attention-regularised transformer, this time both on the sequence `§aabbaabb`.

For the *sorting* task, we present a full computation flow on the input sequence `§fedcbaABCDEF`, alongside the corresponding attention heads of the regularised transformer on the same sequence, in Figure 7.6.6. The regularised transformer had input alphabet of size 52 and reached test accuracy 99.0% on the task (measured as percentage of output positions where the correct output token had the maximum probability).

For the *most-freq* task (returning each unique token in the input, by descending order of frequency, and padding the rest with the BOS token) we do the again show a computation flow alongside the regularised transformer, this time in Figure 7.6.7 and with the sequence `§aabbcddd`. On this task the regularised transformer had input alphabet of size 26 and reached test accuracy 99.9%.

7.6.2 Training Details

In the upper bound and tightness experiments (Section 7.5), for each task and layer/head specification, we train transformers with embedding dimension 256 and feed-forward dimension 512 on the task for 100 epochs. We use learning rates 0.0003 and 0.0001, and learning rate decay $\gamma = 0.98$ and 0.99, training 4 transformers overall for each task. We use the ADAM optimiser and no dropout. Each transformer is trained on sequences of length $0 - -100$, with train/validation/test set sizes of $50,000$, $1,000$, and $1,000$ respectively. Excluding the BOS token, the alphabet sizes are: 3 and 5 and for Dyck-1 and Dyck-2 (the parentheses, plus one neutral token), 100 for reverse and sort, and 26 for the rest (to allow for sufficient repetition of tokens in the input sequences). All input sequences are sampled uniformly from the input alphabet and length, with exception of the Dyck languages, for which they are generated with a bias towards legal prefixes to avoid most outputs being `F`.

For the attention regularised transformers, we make the following changes: first, we only train one transformer per language, with learning rate 0.0003 and decay 0.98. We train each transformer for 250 epochs (though they reach high validation accuracy much earlier than that). The loss this time is added to an MSE-loss component, computed from the differences between each attention distribution and its expected pattern. As this loss is quite small, we scale it by a factor of 100 before adding it to the standard output loss.

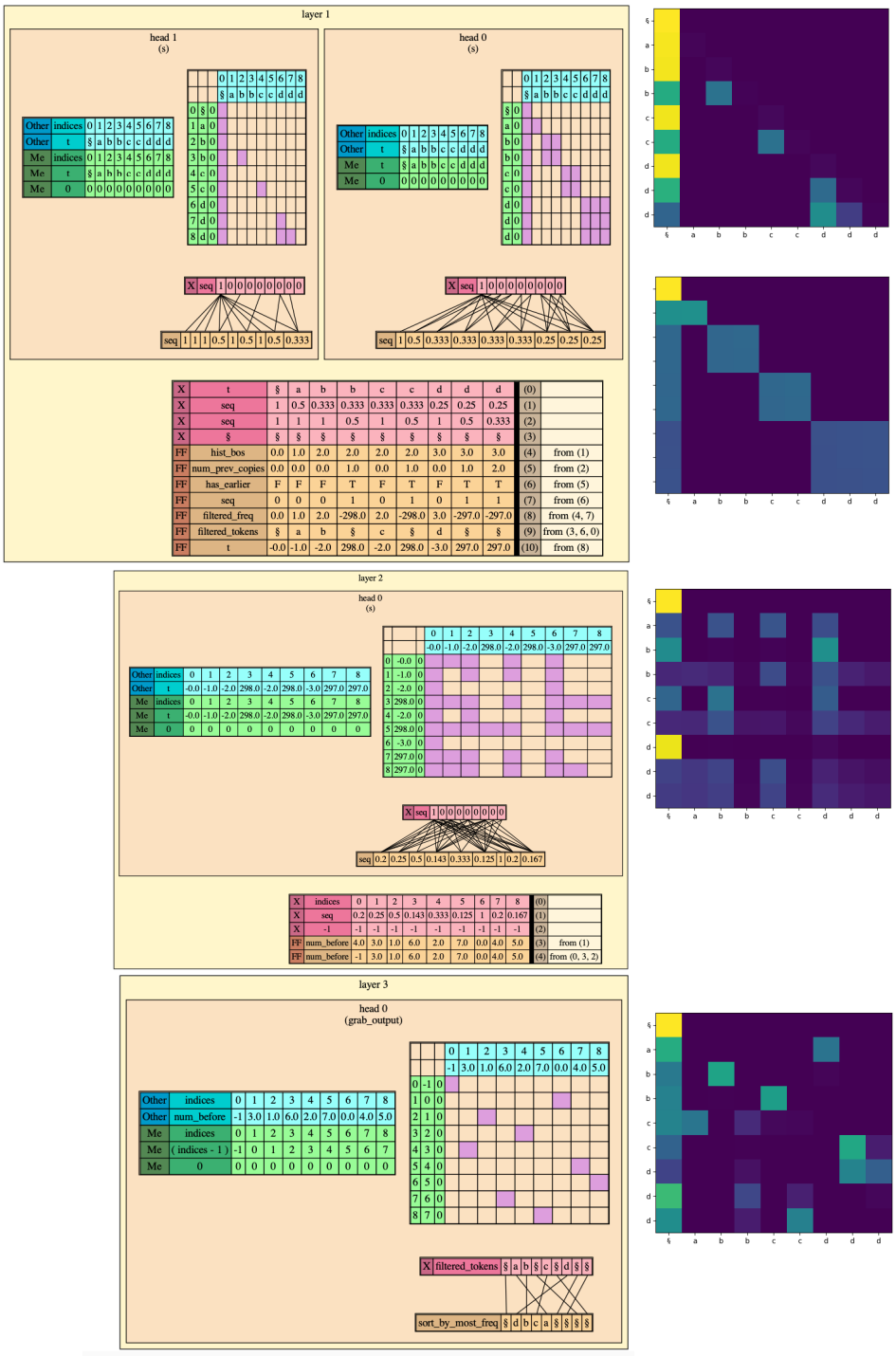


Figure 7.6.7: Computation flow in compiled architecture from RASP solution for sorting by frequency (returning all unique tokens in an input sequence, sorted by decreasing frequency), alongside heatmaps from attention heads in transformer trained on same task and regularised to create same attention patterns. Both are presented on the input sequence `§abbccddd`, for which the correct output is `§dbca`. The transformer architecture has 3 layers with 2 heads apiece, but the RASP architecture requires only 1 head for each of the second and third layers. We regularised only one for each of these and present just that head.

7.7 RASP programs and computation flows for the tasks considered

In this section we present the exact RASP solutions for all tasks considered in this work, as well as an implementation of the operation `selector_width` in terms of other operations (which have direct translation to a transformer). We also present the computation flows in compiled architectures for several of these solutions.

7.7.1 `selector_width`

The RASP implementation of `selector_width` is presented in Figure 7.7.9. The core observation is that, by using a selector that always focuses on zero (`or0` in the presented code), we can compute the inverse of that selector's width by aggregating a 1 from position 0 and 0 from everywhere else. It then remains only to make a correction according to whether or not the selector was actually focused on 0, using the second selector `and0` (if there isn't a beginning-of-sequence token) or our prior knowledge about the input (if there is).

7.7.2 RASP solutions for the paper tasks

We now present the RASP solutions for each of the tasks considered in the paper, as well as an implementation of the RASP primitive `selector_width` in terms of only the primitives `select` and `aggregate`.

The solution for histograms, with or without a BOS token, is given in Figure 7.7.11. The code for double-histograms (e.g., `hist2("aaabbccdef")=[1,1,1,2,2,2,2,3,3,3]`) is given in Figure 7.7.12. The general sorting algorithm (sorting any one sequence by the values ('keys') of any other sequence) is given in Figure 7.7.13, and sorting the tokens by their frequency ("Most freq") is given in Figure 7.7.14. Descriptions of these solutions are in their captions.

The Dyck-PTF Languages *Dyck*-1-*PTF* First each position attends to all previous positions up to and including itself in order to compute the balance between opening and closing braces up to itself, not yet considering the internal ordering of these. Next, each position again attends to all previous positions, this time to see if the ordering was problematic at some point (i.e., there was a negative balance). From there it is possible to infer for each prefix whether it is balanced (T), could be balanced with some more closing parentheses (P), or can no longer be balanced (F). We present the code in Figure 7.7.15.

Dyck-2-*PTF* For this descripition we differentiate between instances of an opening and closing parenthesis (*opener* and *closer*) matching each other with respect to their position within a given sequence, e.g. as `(,>` and `{,]` do in the sequence `({]>`, and of the actual tokens matching with respect to the pair definitions, e.g. as the token

```
pairs = ["()","{}","[]"]; # etc ...
openers = [p[0] for p in pairs];
closers = [p[1] for p in pairs];
opens = tokens in openers;
closes = tokens in closers;
n_opens = num_prevs(opens);
n_closes = num_prevs(closes);

depth = n_opens - n_closes;
adjusted_depth =
    depth + indicator(closes);
earlier_same_depth =
    select(adjusted_depth,adjusted_depth,==)
        and
    select(indices,indices,<=);
depth_index =
    selector_width(earlier_same_depth);
open_for_close =
        select(opens,True,==) and
        earlier_same_depth and
        select(depth_index,
               depth_index-1,==);
matched_opener =
    aggregate(open_for_close,tokens,"-");
opener_matches =
    (matched_opener+tokens) in pairs;
mismatch = closes and not opener_matches;
had_problem =
    num_prevs(mismatch or depth<0 )>0;
dyck3 = "F" if had_problem else
        ("T" if depth==0 else "P");
```

Figure 7.7.8: Pure RASP code (as opposed to with an additional select-best operation) for computing Dyck-3-PTF with the parentheses `(,)`, `{,}` and `[,]`. The code can be used for any Dyck-n by extending the list `pairs`, without introducing additional layers or heads.

pairs `{,}` and `(,)` are defined. For clarity, we refer to these as structure-match and pair-match, respectively.

For a Dyck-n sequence to be balanced, it must satisfy the balance checks as described in Dyck-1 (when treating all openers and all closers as the same), and additionally, it must satisfy that every structure-matched pair is also a pair-match.

We begin by using the function `num_prevs` from Figure 7.7.15 to compute balances as for Dyck-1, ignoring which token pair each opener or closer belongs to. Next, we create an attention pattern `open_for_close` that focuses each closer on its structure-matched opener, and use that pattern to pull up the structure-matched opener for each closer (the behaviour of that pattern on closers that do not have structure-matched openers is not important: in this case there will anyway be a negative balance at that closer). For each location, we then check that it does not have an earlier negative balance, and it does not have an earlier closer whose structure-matched opener is not a pair-match. If it fails these conditions the output is F, otherwise it is T if the current balance is 0 and P otherwise. The remaining challenge is in computing `open_for_close`.

In pure RASP—i.e., within the language as presented in this work—this is realisable in two steps. First, we number each parenthesis according to how many previous parentheses have the same depth as itself, taking for openers the depth after their appearance and for closers the depth before. For example, for `(())()`, the depths are `[1,2,2,1,1,1]`, and the depth-index is `[1,1,2,2,3,3]`. Then, each closer's structure-matched opener is the opener with the same depth as itself, and depth-number immediately preceding its own. This solution is given in Figure 7.7.8, and compiles to 4 layers with maximum width 2.

However, by adding the theoretical operation `select_best`, and a scorer object similar to selectors (with number values as opposed to booleans), we can reduce the computation of `open_for_close` to simply: "find the last opener with the same depth as the closer, that is still before the closer". In this case, the depth-index of each position does not need to be computed in order to obtain `open_for_close`, saving the layer and 2 heads that its compilation creates. We now elaborate on `select_best` and this alternative computation of `open_for_close`.

`select_best` (Theoretical) First, we imagine a new operator `score` that behaves similarly to `select`, except that: instead of using predicates such as `==`, `<`, or `>` to compare values and create *selectors*, it expects the values to be numbers and simply multiplies them to create *scorers*. For example (presented on concrete input for clarity, similar to the presentations in Section 7.3):

$$\text{score}([0,1,2],[1,-1,1]) = \begin{bmatrix} 0 & 1 & 2 \\ 0 & -1 & -2 \\ 0 & 1 & 2 \end{bmatrix}$$

Next we define `select_best` as a function taking one selector *sel* and one scorer *sc*, and returning a new selector in which, for each position, only the selected value of *sel* with the highest score in *sc* remains. For example (presented again on concrete input for clarity, as opposed to function-building syntax of RASP):

$$\text{sel\_best}(\begin{bmatrix} \mathbf{T} & \mathbf{T} & \mathbf{T} \\ \mathbf{T} & \mathbf{T} & F \\ F & F & F \end{bmatrix}, \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{bmatrix}) = \begin{bmatrix} F & F & \mathbf{T} \\ F & \mathbf{T} & F \\ F & F & F \end{bmatrix}$$

With this definition of `score` and `select_best` we obtain `open_for_close` using the alternative approach described above (the last opener with the same depth as the closer's, that is still before the closer) as follows:

```
possible_open_for_close =
    select(indices,indices,<) and
    select(opens,True,==) and
    select(adjusted_depth,adjusted_depth,==);
open_for_close = select_best(
                          open_for_close,
                          score(indices,1) );
```

This approach does not use `depth_index` to obtain `open_for_close`, allowing us to save a layer in our calculation.

7.7.3 Computation flows for select solutions

RASP can compile the the architecture of any s-op, and display it with an example input sequence. The command is `draw(s2s,inp)` where `s2s` is the target s-op and `inp` is the example sequence to display, e.g., `draw(dyck1,"(())")`.

Example computation flows for `hist_bos` and `reverse` are given in the main paper in Figures 7.5.5 and 7.5.4, respectively.

An example computation flow for `hist_nobos` is given in Figure 7.7.16. The double-histogram flow partially shown in Figure 7.1.1 is shown in full in Figure 7.7.17. Computation flows for the compiled architectures of `sort` and for `most_freq` (as solved in Figures 7.7.13 and 7.7.14) are shown in full, alongside the attention patterns of respectively attention-regularised transformers, in Section 7.6. Computation flows for Dyck-1-PTF and Dyck-2-PTF are shown in Figure 7.7.18 and Figure 7.7.19.

```
    def selector_width(sel,
                       assume_bos = False) {

        light0 = indicator(
                         indices == 0);
        or0 = sel or select_eq(indices,0);
        and0 =sel and select_eq(indices,0);
        or0_0_frac =aggregate(or0, light0);
        or0_width = 1/or0_0_frac;
        and0_width =
            aggregate(and0,light0,0);

        # if has bos, remove bos from width
        # (doesn't count, even if chosen by
        #  sel) and return.
        bos_res = or0_width - 1;

        # else, remove 0-position from or0,
        # and re-add according to and0:
        nobos_res = bos_res + and0_width;

        return bos_res if assume_bos else
                         nobos_res;
    }

```

Figure 7.7.9: Implementation of the powerful RASP operation `selector_width` in terms of other RASP operations. It is through this implementation that RASP compiles `selector_width` down to the transformer architecture.

```
reverse = aggregate(
    select(indices,
           length-indices-1,==)
    tokens );
```

Figure 7.7.10: RASP one-liner for reversing the original input sequence, `tokens`. This compiles to an architecture with two layers: `length` requires an attention head to compute, and `reverse` applies a `select-aggregate` pair that uses (among others) the s-op `length`.

```
def histf(seq, assume_bos = False) {
    same_tok = select(seq,seq,==);
    return selector_width(same_tok,
            assume_bos= assume_bos);
}
```

Figure 7.7.11: RASP program for computing histograms over any sequence, with or without a BOS token. Assuming a BOS token allows compilation to only one layer and one head, through the implementation of `selector_width` as in Figure 7.7.9. The `hist_bos` and `hist_nobos` tasks in this work are obtained through `histf(tokens)`, with or without `assume_bos` set to `True`.

```
def has_prev(seq) {
    prev_copy =
        select(seq,seq,==) and
        select(indices,indices,<);
    return aggregate(prev_copy,1,0)>0;
}

is_repr = not has_prev(tokens);
same_count =
    select(hist_bos, hist_bos,==);
same_count_reprs = same_count and
    select(isnt_repr, False,==);
hist2 =selector_width(
        same_count_reprs,
        assume_bos = True);
```

Figure 7.7.12: RASP code for hist-2, making use of the previously computed `hist` s-op created in Figure 7.7.11. We assume there is a BOS token in the input, though we can remove that assumption by simply using `hist_nobos` and removing `assume_bos=True` from the call to `selector_width`. The segment defines and uses a simple function `has_prev` to compute whether a token already has an copy earlier in the sequence.

```
def sort(vals,keys,assume_bos=False) {
    smaller = select(keys,keys,<) or
        (select(keys,keys,==) and
         select(indices,indices,<) );
    num_smaller =
        selector_width(smaller,
            assume_bos=assume_bos);
    target_pos = num_smaller if
                    not assume_bos else
    (0 if indices==0 else (num_smaller+1));
    sel_new =
        select(target_pos,indices,==);
    sort = aggregate(sel_new,vals);
}
```

Figure 7.7.13: RASP code for sorting the s-op `vals` according to the order of the tokens in the s-op `keys`, with or without a BOS token. The idea is for every position to focus on all positions with keys smaller than its own (with input position as a tiebreaker), and then use `selector_width` to compute its target position from that. A further select-aggregate pair then moves each value in `val` to its target position. The sorting task considered in this work's experiments is implemented simply as `sort_input=sort(tokens,tokens)`.

```
max_len = 20000;
freq = hist(tokens,assume_bos=True);
is_repr = not has_prev(tokens);
keys = freq -
    indicator(not is_repr) * max_len;
values = tokens if is_repr else "§"
most_freq = sort(values,keys,
                 assume_bos=True);
```

Figure 7.7.14: RASP code for returning the unique tokens of the input sequence (with a BOS token), sorted by order of descending frequency (with padding for the remainder of the output sequence). The code uses the functions `hist` and `sort` defined in Figures 7.7.11 and 7.7.13, as well as the utility function `has_prev` defined in Figure 7.7.12. First, `hist` computes the frequency of each input token. Then, each input token with an earlier copy of the same token (e.g., the second `"a"` in `"baa"`) is marked as a duplicate. The key for each position is set as its token's frequency, minus the maximum expected input sequence length if it is marked as a duplicate. The value for each position is set to its token, unless that token is a duplicate in which case it is set to the non-token `§`. The values are then sorted by the keys, using `sort` as presented in Figure 7.7.13.

```
def num_prevs(bools) {
    prevs = select(indices,indices,<=);
    return (indices+1) *
           aggregate(prevs,
                     indicator(bools));
}
n_opens = num_prevs(tokens=="(");
n_closes = num_prevs(tokens==")");
balance = n_opens - n_closes;
prev_imbalances = num_prevs(balance<0);
dyck1PTF = "F" if prev_imbalances > 0
                 else
           ("T" if balance==0 else "P");
```

Figure 7.7.15: RASP code for computing Dyck-1-PTF with the parentheses `(` and `)`.

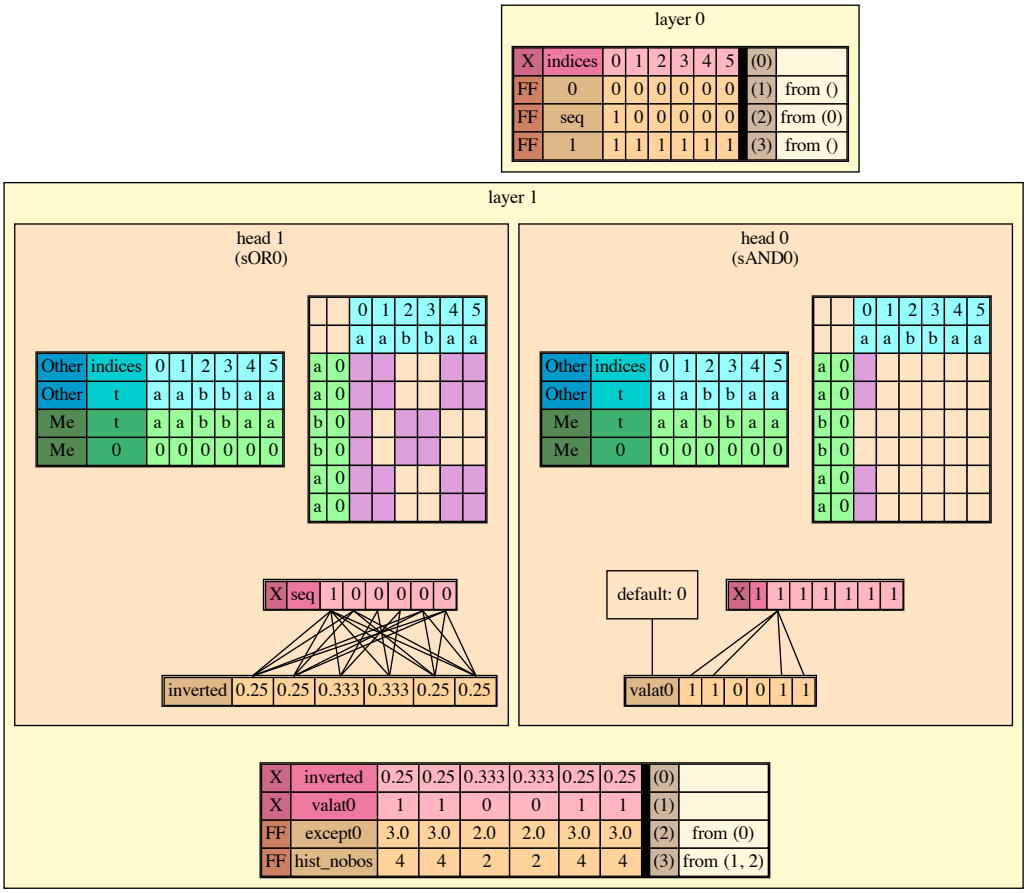


Figure 7.7.16: Computation flow in compiled architecture from RASP solution for histogram without a beginning-of-sequence token (using `histf(tokens)` with `histf` from Figure 7.7.11). We present the short sequence `"aabbaa"`, in which the counts of `a` and `b` are different.

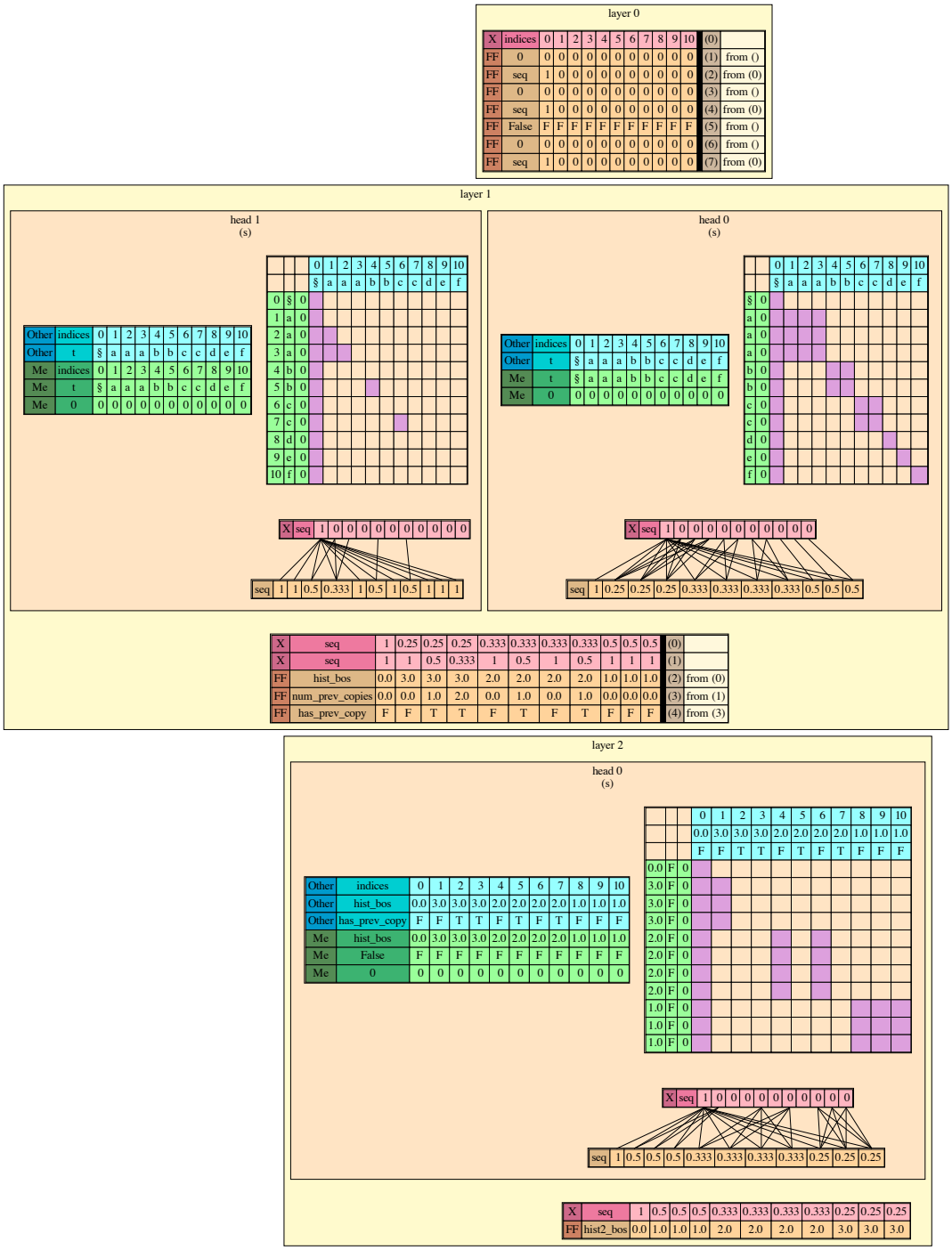


Figure 7.7.17: Computation flow in compiled architecture from RASP solution for double-histogram, for solution shown in Figure 7.7.12. Applied to `"§aaabbccdef"`, as in Figure 7.1.1.

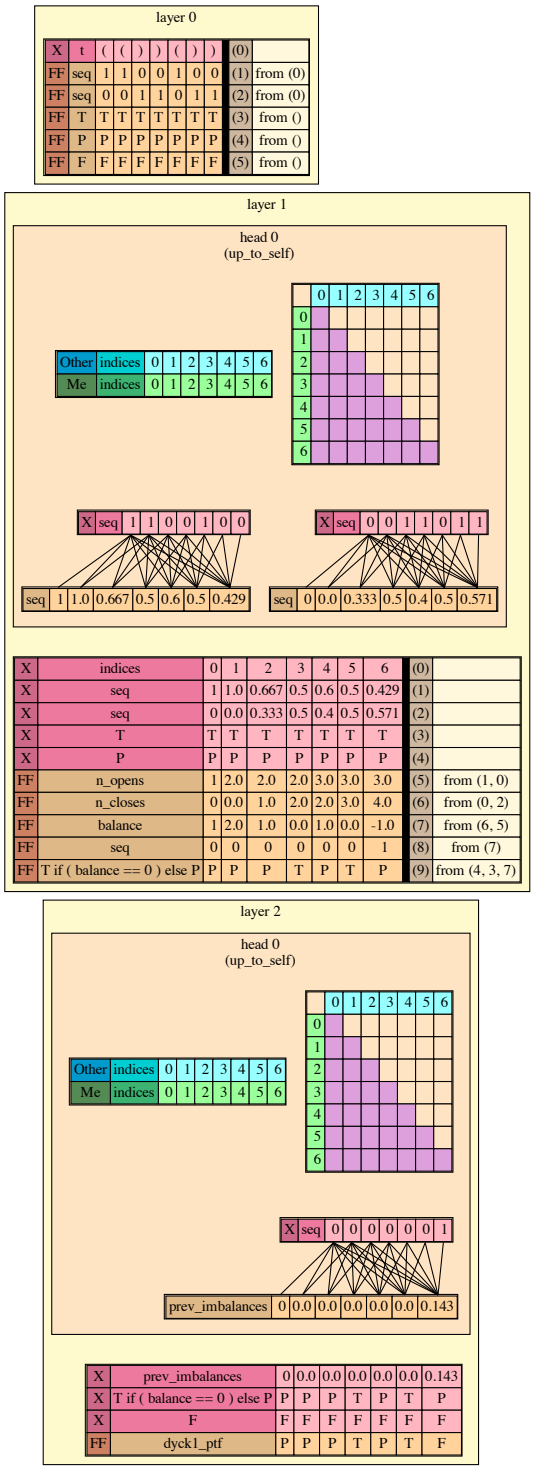


Figure 7.7.18: Computation flow in compiled architecture from RASP solution for Dyck-1, for solution shown in Figure 7.7.15. Applied to the unbalanced input sequence "(())())".

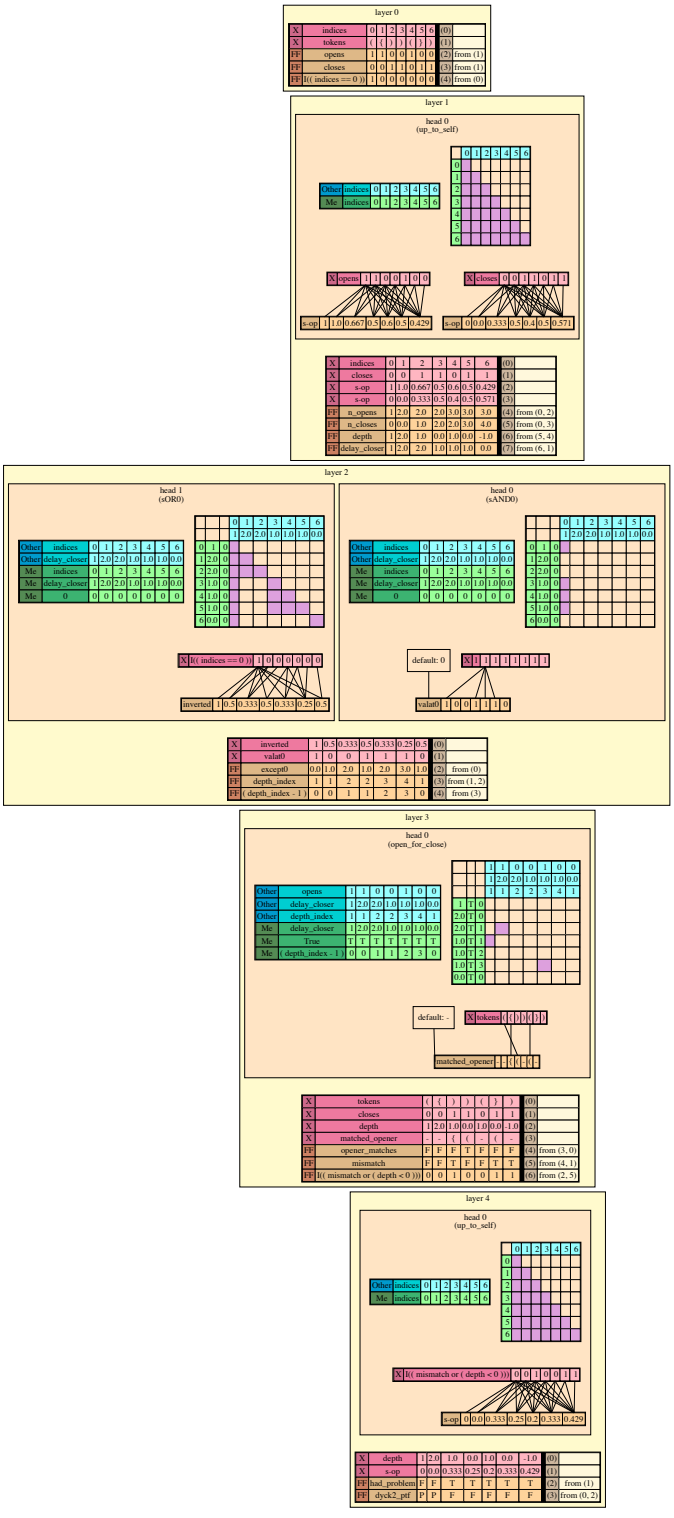


Figure 7.7.19: Computation flow in compiled architecture from RASP solution for Dyck-2, for solution shown in Figure 7.7.8. Applied to the unbalanced and 'incorrectly matched' (with respect to structure/pair-matches) sequence `"())()"`.

Chapter 8

Conclusions

We approach the problem of understanding neural sequence models (NSMs), using the framework of formal languages and models. We begin with an extensive exploration of the connection between RNNs and various automata variants, which we use to recover several types of formal languages and rules from RNNs and also to predict the kinds of formal languages and rules which they can encode. We then lay the groundwork for similar explorations in transformers, proposing RASP as a model with a strong intuitive connection to transformers. Our extraction work diverges from previous methods by applying *exact learning* (and in particular the L^* algorithm) to neural networks, and our RNN analysis work differs from previous results by taking into account the constraints under which RNNs are used in practice. Finally, while many works analyse the expressive power of transformers, our work on RASP is the first to propose an intuitive model for reasoning about and communicating the mechanisms they use in practice.

We evaluate our extraction works on various RNNs trained on multiple formal and even (in the case of PDFA extraction) some real-world languages. On simple regular languages, our DFA extraction approach takes orders of time less magnitude to complete than other methods, and regularly recovers smaller, more accurate DFAs—regardless of the size of the underlying RNNs. Our PDFA extraction work is also optimal for languages defined by small target PDFAs. For the real world tasks however, extracting n-grams maintains a strong advantage over PDFAs. Both methods will benefit from future work to make them more efficient, for example by ignoring 'unimportant' parts of the input space.

In our experiments on DFA extraction, we play with some non-regular languages and observe some RNNs achieving surprising success. On closer inspection of their internal architectures, we see that the LSTM networks are well adapted to implementing a counting mechanism in individual dimensions in their memory vectors, while GRUs cannot do this at all (and similarly for the IRNN and SRNN architectures). We demonstrate the resulting difference in the practical expressive power of the GRU and LSTM architectures.

Our work on RNNs makes the case that knowing the appropriate symbolic models for different NSM architectures can guide us both in extracting symbolic models from trained networks and in understanding the different expressive capacities of their variants. We are guided by this to our next project, in which we seek the appropriate symbolic model for the transformer.

We abstract the computation model of the transformer-encoder as a simple sequence processing language, RASP, that captures the unique constraints on information flow present in a transformer. Considering computation problems and their implementation in RASP allows us to "think like a transformer" while abstracting away the technical details of a neural network in favour of symbolic programs. We show several examples of programs written in the RASP language, showing how operations can be implemented by a transformer, and train several transformers on these tasks, finding that RASP helps predict the number of transformer heads and layers needed to solve them.

For RNNs, we show that taking into consideration the constraints with which an RNN is used in practice (in particular, $O(n)$ or even exactly n run time, and bounded precision) can reveal much weaker expressive power than previously believed, and even differentiates between different RNN architectures previously considered equivalent. Specifically, we show that the LSTM and IRNN architectures can implement counters in individual dimensions of their states, and use them to recognise various counting-based languages, a set of languages that intersects the Chomsky hierarchy. In contrast, we find that the GRU and SRNN are unable to implement such counters in practice, and struggle on counting tasks in general. Our results highlight the importance of relevant practical assumptions for theoretical analyses.

For transformers, we begin without any model for reasoning about their behaviour: there is no intuitive way to describe the transfer of information from one input position to another, or We show several examples of programs written in the RASP language, showing how operations can be implemented by a transformer, and train several transformers on these tasks, finding that RASP helps predict the number of transformer heads and layers needed to solve them. Additionally, we use RASP to shed light on an empirical observation over transformer variants, and find concrete limitations for some "efficient transformers".

Resources

The codebases for the works presented in this thesis are available at (in order):

`www.github.com/tech-srl/lstar_extraction` (DFA extraction from RNNs)
`www.github.com/tech-srl/counting_dimensions` (LSTMs can count)
`www.github.com/tech-srl/weighted_lstar` (W- and P-DFA learning)
`www.github.com/tech-srl/rasp` (RASP: REPL)
`www.github.com/tech-srl/RASP-exps` (RASP: transformer experiments)

Bibliography

[AEG18] Stéphane Ayache, Rémi Eyraud, and Noé Goudian. Explaining black boxes on sequential data using weighted automata. In Olgierd Unold, Witold Dyrka, and Wojciech Wieczorek, editors, *Proceedings of the 14th International Conference on Grammatical Inference, ICGI 2018, Wrocław, Poland, September 5-7, 2018*, volume 93 of *Proceedings of Machine Learning Research*, pages 81–103. PMLR, 2018.

[AG17] Roee Aharoni and Yoav Goldberg. Towards string-to-tree neural machine translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 132–140, Vancouver, Canada. Association for Computational Linguistics, 2017.

[AKB+16] Yossi Adi, Einat Kermany, Yonatan Belinkov, Ofer Lavi, and Yoav Goldberg. Fine-grained analysis of sentence embeddings using auxiliary prediction tasks. *CoRR*, abs/1608.04207, 2016. arXiv: `1608.04207`.

[AMMS17] Leila Arras, Grégoire Montavon, Klaus-Robert Müller, and Wojciech Samek. Explaining recurrent neural network predictions in sentiment analysis. *CoRR*, abs/1706.07206, 2017. arXiv: `1706.07206`.

[And15] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`, 2015.

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[AOA+20] Joshua Ainslie, Santiago Ontañón, Chris Alberti, Philip Pham, Anirudh Ravula, and Sumit Sanghai. ETC: encoding long and structured data in transformers. *CoRR*, abs/2004.08483, 2020. arXiv: `2004.08483`.

[BAG20] Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the ability and limitations of transformers to recognize formal languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP*, 2020.

[Bai11] Raphael Bailly. Quadratic weighted automata:spectral algorithm and likelihood maximization. In Chun-Nan Hsu and Wee Sun Lee, editors, *Proceedings of the Asian Conference on Machine Learning*, volume 20 of *Proceedings of Machine Learning Research*, pages 147–163. PMLR, 2011.

[BBF+21] Benoît Barbot, Benedikt Bollig, Alain Finkel, Serge Haddad, Igor Khmelnitsky, Martin Leucker, Daniel Neider, Rajarshi Roy, and Lina Ye. Extracting context-free grammars from recurrent neural networks using tree-automata learning and a* search. In Jane Chandlee, Rémi Eyraud, Jeff Heinz, Adam Jardine, and Menno van Zaanen, editors, *Proceedings of the Fifteenth International Conference on Grammatical Inference*, volume 153 of *Proceedings of Machine Learning Research*, pages 113–129. PMLR, 2021.

[BCB15] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR*, 2015.

[BCG13] Borja Balle, Jorge Castro, and Ricard Gavaldà. Learning probabilistic automata: A study in state distinguishability. *Theor. Comput. Sci.*, 473:46–60, 2013.

[BCLQ14] Borja Balle, Xavier Carreras, Franco M. Luque, and Ariadna Quattoni. Spectral learning of weighted automata - A forward-backward perspective. *Mach. Learn.*, 96(1-2):33–63, 2014.

[BDR09] Raphael Bailly, François Denis, and Liva Ralaivola. Grammatical inference as a principal component analysis problem. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 33–40. ACM, 2009.

[BEL+16] Borja Balle, Rémi Eyraud, Franco M. Luque, Ariadna Quattoni, and Sicco Verwer. Results of the sequence prediction challenge (spice): a competition on learning the next symbol in a sequence. In *Proceedings of the 13th International Conference on Grammatical Inference,ICGI*, pages 132–136, 2016.

[Ber18] Jean-Phillipe Bernardy. Can recurrent neural networks learn nested recursion? In *Linguistic Issues in Language Technology, Volume 16*. CSLI Publications, 2018.

[BGV92] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 144–152, Pittsburgh, Pennsylvania, USA. ACM, 1992.

[BJLS05] Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena. Insights to angluin's learning. *Electr. Notes Theor. Comput. Sci.*, 118:3–18, 2005.

[BKH16] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016. arXiv: 1607.06450.

[BM15] Borja Balle and Mehryar Mohri. Learning weighted automata. In *Algebraic Informatics - 6th International Conference, CAI 2015, Stuttgart, Germany, September 1-4, 2015. Proceedings*, pages 1–21, 2015.

[BMR+20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[BPC20] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: the long-document transformer. *CoRR*, abs/2004.05150, 2020. arXiv: 2004.05150.

[Cas98] Michael Casey. Correction to proof that recurrent neural networks can robustly recognize only regular languages. *Neural Computation*, 10(5):1067–1069, 1998.

[CC16] Do Kook Choe and Eugene Charniak. Parsing as language modeling. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2331–2336, Austin, Texas. Association for Computational Linguistics, 2016.

[CCRB17] M. Cohen, A. Caciularu, I. Rejwan, and J. Berant. Inducing Regular Grammars Using Recurrent Neural Networks. *ArXiv e-prints*, 2017. arXiv: 1710.10453 [cs.CL].

[CE07] Alexander Clark and Rémi Eyraud. Polynomial identification in the limit of substitutable context-free languages. *J. Mach. Learn. Res.*, 8:1725–1745, 2007.

[CG08] Jorge Castro and Ricard Gavaldà. Towards feasible pac-learning of probabilistic deterministic finite automata. In *Grammatical Inference: Algorithms and Applications, 9th International Colloquium, ICGI 2008, Saint-Malo, France, September 22-24, 2008, Proceedings*, pages 163–174, 2008.

[CGCB14a] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv:1412.3555 [cs]*, 2014. (Visited on 09/17/2015).

[CGCB14b] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014. arXiv: `1412.3555`.

[CGKM17] Yining Chen, Sorcha Gilroy, Kevin Knight, and Jonathan May. Recurrent neural networks as weighted language recognizers. *CoRR*, abs/1711.05408, 2017. arXiv: `1711.05408`.

[CGRS19] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *CoRR*, abs/1904.10509, 2019. arXiv: `1904.10509`.

[Chr15] Christopher Olah. Understanding lstm networks. `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2015.

[CKLM19] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. What does BERT look at? an analysis of bert's attention. *CoRR*, abs/1906.04341, 2019. arXiv: `1906.04341`.

[Cla10] Alexander Clark. Distributional learning of some context-free languages with a minimally adequate teacher. In José M. Sempere and Pedro García, editors, *Grammatical Inference: Theoretical Results and Applications, 10th International Colloquium, ICGI 2010, Valencia, Spain, September 13-16, 2010. Proceedings*, volume 6339 of *Lecture Notes in Computer Science*, pages 24–37. Springer, 2010.

[CMGS10] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural Comput.*, 22(12):3207–3220, 2010.

[CO94] Rafael C. Carrasco and José Oncina. Learning stochastic regular grammars by means of a state merging method. In Rafael C. Carrasco and José Oncina, editors, *Grammatical Inference and Applications*, pages 139–152, Berlin, Heidelberg. Springer Berlin Heidelberg, 1994.

[CO99] Rafael C. Carrasco and José Oncina. Learning deterministic regular grammars from stochastic samples in polynomial time. *ITA*, 33(1):1–20, 1999.

[CSM89] Axel Cleeremans, David Servan-Schreiber, and James L. McClelland. Finite state automata and simple recurrent networks. *Neural Comput.*, 1(3):372–381, 1989.

[CSS03] Adelmo Luis Cechin, Denise Regina Pechmann Simon, and Klaus Stertz. State automata extraction from recurrent neural nets using k-means and fuzzy clustering. In *Proceedings of the XXIII International Conference of the Chilean Computer Science Society*, SCCC '03, pages 73–78, Washington, DC, USA. IEEE Computer Society, 2003.

[CT04] Alexander Clark and Franck Thollard. Pac-learnability of probabilistic deterministic finite state automata. *Journal of Machine Learning Research*, 5:473–497, 2004.

[CTR20] Peter Clark, Oyvind Tafjord, and Kyle Richardson. Transformers as soft reasoners over language. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI*, 2020.

[CvG+14] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar. Association for Computational Linguistics, 2014. (Visited on 09/09/2015).

[CvMBB14] KyungHyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: encoder-decoder approaches. *CoRR*, abs/1409.1259, 2014. arXiv: `1409.1259`.

[CY16] Alexander Clark and Ryo Yoshinaka. *Distributional learning of context-free and multiple context-free grammars*. In *Topics in Grammatical Inference*. Jeffrey Heinz and José M. Sempere, editors. Springer Berlin Heidelberg, 2016, pages 143–172.

[CZH+21] Yu-An Chung, Yu Zhang, Wei Han, Chung-Cheng Chiu, James Qin, Ruoming Pang, and Yonghui Wu. W2v-bert: combining contrastive learning and masked language modeling for self-supervised speech pre-training. In *IEEE Automatic Speech Recognition and Understanding Workshop, ASRU 2021, Cartagena, Colombia, December 13-17, 2021*, pages 244–250. IEEE, 2021.

[DCK20] Raj Dabre, Chenhui Chu, and Anoop Kunchukuttan. A survey of multilingual neural machine translation. 53(5), September 2020.

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. arXiv: `1810.04805`.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: a large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[DFG10] Arianna D'ulizia, F. Ferri, and P. Grifoni. A survey of grammatical inference methods for natural language learning. *Artificial Intelligence Review*, 36:1–27, 2010.

[DGV+19] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. In *7th International Conference on Learning Representations, ICLR*, 2019.

[EGZ20] Javid Ebrahimi, Dhruv Gelda, and Wei Zhang. How can self-attention networks recognize Dyck-n languages? In *Findings of the Association for Computational Linguistics: EMNLP*. Association for Computational Linguistics, 2020.

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231, Portland, Oregon. AAAI Press, 1996.

[Elm90] Jeffrey L. Elman. Finding Structure in Time. en. *Cognitive Science*, 14(2):179–211, 1990. (Visited on 09/10/2015).

[FMR68] Patrick C. Fischer, Albert R. Meyer, and Arnold L. Rosenberg. Counter machines and counter languages. *Mathematical systems theory*, 2(3):265–283, 1968.

[GBC16] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016.

[GGCC94] Mark W. Goudreau, C. Lee Giles, Srimat T. Chakradhar, and D. Chen. First-order versus second-order single-layer recurrent neural networks. *IEEE Trans. Neural Networks*, 5(3):511–513, 1994.

[GK95] Sally A. Goldman and Michael J. Kearns. On the complexity of teaching. *J. Comput. Syst. Sci.*, 50(1):20–31, 1995.

[GMC+92] C. L. Giles, C. B. Miller, D. Chen, H. H. Chen, G. Z. Sun, and Y. C. Lee. Learning and extracting finite state automata with second-order recurrent neural networks. *Neural Comput.*, 4(3):393–405, 1992.

[Gol16] Yoav Goldberg. A primer on neural network models for natural language processing. *J. Artif. Intell. Res.*, 57:345–420, 2016.

[Gol17] Yoav Goldberg. *Neural Network Methods for Natural Language Processing*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2017.

[Goo01] Joshua T Goodman. A bit of progress in language modeling. *Computer Speech & Language*, 15(4):403–434, 2001.

[GS00] F. A. Gers and J. Schmidhuber. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, 189–194 vol.3, 2000.

[GS01] F. A. Gers and E. Schmidhuber. Lstm recurrent networks learn simple context-free and context-sensitive languages. *Transactions on Neural Networks*, 12(6):1333–1340, 2001.

[GS16] Kyle Gorman and Richard Sproat. Minimally supervised number normalization. *Transactions of the Association for Computational Linguistics*, 4:507–519, 2016.

[GSC+90] C. Lee Giles, Guo-Zheng Sun, Hsing-Hen Chen, Yee-Chun Lee, and Dong Chen. Higher order recurrent networks and grammatical inference. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 380–387. Morgan-Kaufmann, 1990.

[GSK+17] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber. Lstm: a search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10):2222–2232, 2017.

[HAF22] Yiding Hao, Dana Angluin, and Robert Frank. Formal language recognition by hard attention transformers: perspectives from circuit complexity. *Trans. Assoc. Comput. Linguistics*, 10:800–810, 2022.

[Hah20] Michael Hahn. Theoretical limitations of self-attention in neural sequence models. *Trans. Assoc. Comput. Linguistics*, 2020.

[HCS+16] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 4107–4115. Curran Associates, Inc., 2016.

[HDY+12] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *Signal Processing Magazine*, 2012.

[HHG+20] John Hewitt, Michael Hahn, Surya Ganguli, Percy Liang, and Christopher D. Manning. RNNs can generate bounded hierarchical languages with optimal memory. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1978–2010, Online. Association for Computational Linguistics, 2020.

[HKZ08] Daniel J. Hsu, Sham M. Kakade, and Tong Zhang. A spectral algorithm for learning hidden markov models. *CoRR*, abs/0811.4413, 2008. arXiv: `0811.4413`.

[HRP+22] Adi Haviv, Ori Ram, Ofir Press, Peter Izsak, and Omer Levy. Transformer language models without positional encodings still learn positional information. *CoRR*, abs/2203.16634, 2022. arXiv: `2203.16634`.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. eprint: `https://doi.org/10.1162/neco.1997.9.8.1735`.

[HSW89a] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[HSW89b] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[HVLS16] Christian Albert Hammerschmidt, Sicco Verwer, Qin Lin, and Radu State. Interpreting Finite Automata for Sequential Data. *arXiv e-prints*, 2016. arXiv: `1611.07100 [stat.ML]`.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[IHS14] Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, volume 8734 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 2014.

[Jac05] Henrik Jacobsson. Rule extraction from recurrent neural networks: a taxonomy and review. *Neural Comput.*, 17(6):1223–1263, 2005.

[JEP+21] John M. Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Zídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Andy Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David A. Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with alphafold. *Nature*, 596:583–589, 2021.

[JM15] Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems, NeurIPS*, 2015.

[KCA16] Ákos Kádár, Grzegorz Chrupala, and Afra Alishahi. Representation of linguistic form and function in recurrent neural networks. *CoRR*, abs/1602.08952, 2016. arXiv: 1602.08952.

[KJL15] Andrej Karpathy, Justin Johnson, and Fei-Fei Li. Visualizing and understanding recurrent networks. *CoRR*, abs/1506.02078, 2015. arXiv: 1506.02078.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[LBJ16] Tao Lei, Regina Barzilay, and Tommi S. Jaakkola. Rationalizing neural predictions. *CoRR*, abs/1606.04155, 2016. arXiv: 1606.04155.

[LCHJ15] Jiwei Li, Xinlei Chen, Eduard H. Hovy, and Dan Jurafsky. Visualizing and understanding neural models in NLP. *CoRR*, abs/1506.01066, 2015. arXiv: 1506.01066.

[LDG16] Tal Linzen, Emmanuel Dupoux, and Yoav Goldberg. Assessing the ability of LSTMs to learn syntax-sensitive dependencies. *Transactions of the Association for Computational Linguistics*, 4:521–535, 2016.

[LJH15] Quoc V. Le, Navdeep Jaitly, and Geoffrey E. Hinton. A Simple Way to Initialize Recurrent Networks of Rectified Linear Units. *arXiv:1504.00941 [cs]*, 2015. (Visited on 09/15/2015).

[LJL+21] Chu-Cheng Lin, Aaron Jaech, Xin Li, Matthew R. Gormley, and Jason Eisner. Limitations of autoregressive models and their alternatives. In Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tür, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty,

and Yichao Zhou, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 5147–5173. Association for Computational Linguistics, 2021.

[LPM15] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP*, 2015.

[LRM+12] Quoc V. Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Greg Corrado, Kai Chen, Jeffrey Dean, and Andrew Y. Ng. Building high-level features using large scale unsupervised learning. In *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*. icml.cc / Omnipress, 2012.

[Mer19] William Merrill. Sequential neural networks as automata. *CoRR*, 2019. arXiv: `1906.01615`.

[MRG+20] William Merrill, Vivek Ramanujan, Yoav Goldberg, Roy Schwartz, and Noah A. Smith. Parameter norm growth during training of transformers. *CoRR*, 2020. arXiv: `2010.09697`.

[MS17] W. James Murdoch and Arthur Szlam. Automatic rule extraction from long short term memory networks. *CoRR*, abs/1702.02540, 2017. arXiv: `1702.02540`.

[MSS22] William Merrill, Ashish Sabharwal, and Noah A. Smith. Saturated transformers are constant-depth threshold circuits. *Trans. Assoc. Comput. Linguistics*, 10:843–856, 2022.

[MWG+20] William Merrill, Gail Weiss, Yoav Goldberg, Roy Schwartz, Noah A. Smith, and Eran Yahav. 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, 2020.

[MY18] Franz Mayr and Sergio Yovine. Regular Inference on Artificial Neural Networks. In Andreas Holzinger, Peter Kieseberg, A Min Tjoa, and Edgar Weippl, editors, *2nd International Cross-Domain Conference for Machine Learning and Knowledge Extraction (CD-MAKE)*, volume LNCS-11015 of *Machine Learning and Knowledge Extraction*, pages 350–369, Hamburg, Germany. Springer International Publishing, 2018. Part 5: MAKE Explainable AI.

[OG00] Christian W. Omlin and C. Lee Giles. Symbolic knowledge representation in recurrent neural networks: insights from theoretical models of computation. In Ian Cloete and Jacek M. Zurada, editors, *Knowledge-based Neurocomputing*, pages 63–116. MIT Press, Cambridge, MA, USA, 2000.

[OG96] Christian W. Omlin and C. Lee Giles. Extraction of rules from discrete-time recurrent neural networks. *Neural Networks*, 9(1):41–52, 1996.

[OWSH20] Takamasa Okudono, Masaki Waga, Taro Sekiyama, and Ichiro Hasuo. Weighted automata extraction from recurrent neural networks via regression on state spaces. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 5306–5314. AAAI Press, 2020.

[PBM21] Jorge Pérez, Pablo Barceló, and Javier Marinkovic. Attention is turing-complete. *Journal of Machine Learning Research*, 22, 2021.

[PG07] Nick Palmer and Paul W. Goldberg. Pac-learnability of probabilistic deterministic finite state automata in terms of variation distance. *Theor. Comput. Sci.*, 387(1):18–31, 2007.

[PGC+17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[PSL20] Ofir Press, Noah A. Smith, and Omer Levy. Improving transformer models by reordering their sublayers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL*, 2020.

[PSY18] Hila Peleg, Sharon Shoham, and Eran Yahav. Programming not only by example. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1114–1124. ACM, 2018.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[QCG17] Ariadna Quattoni, Xavier Carreras, and Matthias Gallé. A maximum matching algorithm for basis selection in spectral learning. *CoRR*, abs/1706.02857, 2017. arXiv: `1706.02857`.

[RBL+22] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*, pages 10674–10685. IEEE, 2022.

[RDN+22] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with CLIP latents. *CoRR*, abs/2204.06125, 2022. arXiv: `2204.06125`.

[RLP19] Guillaume Rabusseau, Tianyu Li, and Doina Precup. Connecting weighted automata and recurrent neural networks through spectral learning. In Kamalika Chaudhuri and Masashi Sugiyama, editors, *The 22nd International Conference on Artificial Intelligence and Statistics, AISTATS*, volume 89 of *Proceedings of Machine Learning Research*, pages 1630–1639. PMLR, 2019.

[Ros00] Ronald Rosenfeld. Two decades of statistical language modeling: where do we go from here? *Proceedings of the IEEE*, 88(8):1270–1278, 2000.

[RST98] Dana Ron, Yoram Singer, and Naftali Tishby. On the learnability and usage of acyclic probabilistic finite automata. *J. Comput. Syst. Sci.*, 56(2):133–152, 1998.

[RSVG21] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Trans. Assoc. Comput. Linguistics*, 9, 2021.

[Rus18] Alexander Rush. The annotated transformer. `https://nlp.seas.harvard.edu/2018/04/03/attention.html`, 2018.

[RV88] H. Rulot and E. Vidal. An efficient algorithm for the inference of circuit-free automata. In Gabriel Ferraté, Theo Pavlidis, Alberto Sanfeliu, and Horst Bunke, editors, *Syntactic and Structural Pattern Recognition*, pages 173–184. Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[RZLL16] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. In Jian Su, Xavier Carreras, and Kevin Duh, editors, *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, pages 2383–2392. The Association for Computational Linguistics, 2016.

[Sak92] Yasubumi Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97(1):23–60, 1992.

[SB18] Luzi Sennhauser and Robert C. Berwick. Evaluating the ability of lstms to learn context-free grammars. In *Proceedings of the Workshop: Analyzing and Interpreting Neural Networks for NLP, BlackboxNLP@EMNLP*, pages 115–124, 2018.

[SGBS19] Mirac Suzgun, Sebastian Gehrmann, Yonatan Belinkov, and Stuart M. Shieber. LSTM networks can perform dynamic counting. *CoRR*, abs/1906.03648, 2019. arXiv: 1906.03648.

[SGH+16] Hendrik Strobelt, Sebastian Gehrmann, Bernd Huber, Hanspeter Pfister, and Alexander M. Rush. Visual analysis of hidden state dynamics in recurrent neural networks. *CoRR*, abs/1606.07461, 2016. arXiv: 1606.07461.

[SHS+17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017. arXiv: 1712.01815.

[Sie96] Hava T. Siegelmann. Recurrent neural networks and finite automata. *Computational Intelligence*, 12:567–574, 1996.

[Sie99] Hava Siegelmann. *Neural Networks and Analog Computation: Beyond the Turing Limit*. Birkhäuser Basel, 1st edition, 1999.

[SKLV16] Vatsal Sharan, Sham M. Kakade, Percy Liang, and Gregory Valiant. Prediction with a short memory. *CoRR*, abs/1612.02526, 2016. arXiv: 1612.02526.

[SPK16] Xing Shi, Inkit Padhi, and Kevin Knight. Does string-based neural mt learn source syntax? In *EMNLP*, pages 1526–1534, 2016.

[SS92] Hava T. Siegelmann and Eduardo D. Sontag. On the computational power of neural nets. In *Proceedings of the Fifth Annual ACM Conference on Computational Learning Theory, COLT 1992, Pittsburgh, PA, USA, July 27-29, 1992*. Pages 440–449, 1992.

[SS94] Hava T. Siegelmann and Eduardo D. Sontag. Analog computation via neural networks. *Theor. Comput. Sci.*, 131(2):331–360, 1994.

[STK18] Natalia Skachkova, Thomas Trost, and Dietrich Klakow. Closing brackets with recurrent neural networks. In *Proceedings of the Workshop: Analyzing and Interpreting Neural Networks for NLP, BlackboxNLP@EMNLP*, pages 232–239, 2018.

[Sut22] Ilya Sutskever. “it may be that today’s large neural networks are slightly conscious”. https://twitter.com/ilyasut/status/1491554478243258368, February 2022.

[SY16] Chihiro Shibata and Ryo Yoshinaka. Probabilistic learnability of context-free grammars with basic distributional properties from positive examples. *Theoretical Computer Science*, 620:46–72, 2016. Algorithmic Learning Theory.

[TDBM20] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *CoRR*, abs/2009.06732, 2020. arXiv: 2009.06732.

[TDdlH00] Franck Thollard, Pierre Dupont, and Colin de la Higuera. Probabilistic DFA inference using kullback-leibler divergence and minimality. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000*, pages 975–982, 2000.

[Tel06] Isabelle Tellier. Learning recursive automata from positive examples. *Rev. d’Intelligence Artif.*, 20(6):775–804, 2006.

[Tom82] M. Tomita. Dynamic construction of finite automata from examples using hill-climbing. In *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, pages 105–108, Ann Arbor, Michigan, 1982.

[VEdlH14] Sicco Verwer, Rémi Eyraud, and Colin de la Higuera. Pautomac: a probabilistic automata and hidden markov models learning competition. *Machine Learning*, 96(1):129–154, 2014.

[VH17] S. Verwer and C. A. Hammerschmidt. Flexfringe: a passive automaton learning package. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 638–642, 2017.

[VKK+15] Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS’15, pages 2773–2781, Montreal, Canada. MIT Press, 2015.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. arXiv: 1706.03762.

[WGY18a] Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting automata from recurrent neural networks using queries and counterexamples. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stock-*

holm, Sweden, July 10-15, 2018, volume 80 of *Proceedings of Machine Learning Research*, pages 5244–5253. PMLR, 2018.

[WGY18b] Gail Weiss, Yoav Goldberg, and Eran Yahav. 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, 2018.

[WGY19] Gail Weiss, Yoav Goldberg, and Eran Yahav. Learning deterministic weighted automata with queries and counterexamples. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[WGY21] Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like transformers. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 11080–11090. PMLR, 2021.

[WGY22] Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting automata from recurrent neural networks using queries and counterexamples (extended version). *Machine Learning*, 2022.

[WK91] Raymond L. Watrous and Gary M. Kuhn. Induction of finite-state automata using second-order recurrent networks. In John E. Moody, Stephen Jose Hanson, and Richard Lippmann, editors, *Advances in Neural Information Processing Systems 4, [NIPS Conference, Denver, Colorado, USA, December 2-5, 1991]*, pages 309–317. Morgan Kaufmann, 1991.

[WN19] Cheng Wang and Mathias Niepert. State-regularized recurrent neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6596–6606. PMLR, 2019.

[WZO+17] Qinglong Wang, Kaixuan Zhang, Alexander G. Ororbia II, Xinyu Xing, Xue Liu, and C. Lee Giles. An empirical evaluation of recurrent neural network rule extraction. *CoRR*, abs/1709.10380, 2017. arXiv: 1709.10380.

[WZO+18] Qinglong Wang, Kaixuan Zhang, Alexander G. Ororbia II, Xinyu Xing, Xue Liu, and C. Lee Giles. A comparison of rule extraction for different recurrent neural network models and grammatical complexity. *CoRR*, abs/1801.05420, 2018. arXiv: 1801.05420.

[YBR+20] Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J. Reddi, and Sanjiv Kumar. Are transformers universal approximators of sequence-to-sequence functions? In *8th International Conference on Learning Representations, ICLR*, 2020.

[YLCT20] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A survey of autonomous driving: common practices and emerging technologies. *IEEE Access*, 8:58443–58469, 2020.

[Yok03] Takashi Yokomori. Polynomial-time identification of very simple grammars from positive data. *Theor. Comput. Sci.*, 298(1):179–206, 2003.

[Yos19] Ryo Yoshinaka. Distributional learning of conjunctive grammars and contextual binary feature grammars. *J. Comput. Syst. Sci.*, 104:359–374, 2019.

[YPPN21] Shunyu Yao, Binghui Peng, Christos Papadimitriou, and Karthik Narasimhan. Self-attention networks can process bounded hierarchical languages. *CoRR*, 2021. arXiv: `2105.11115`.

[YW21] Daniel M. Yellin and Gail Weiss. Synthesizing context-free grammars from recurrent neural networks (extended version). *CoRR*, abs/2101.08200, 2021. arXiv: `2101.08200`.

[YWC20] Shuoheng Yang, Yuxin Wang, and Xiaowen Chu. A survey of deep learning techniques for neural machine translation. *CoRR*, abs/2002.07526, 2020. arXiv: `2002.07526`.

[ZDX+21] Xiyue Zhang, Xiaoning Du, Xiaofei Xie, Lei Ma, Yang Liu, and Meng Sun. Decision-guided weighted automata extraction from recurrent neural networks. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 11699–11707. AAAI Press, 2021.

[ZGD+20] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: transformers for longer sequences. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems, NeurIPS*, 2020.

[ZGS93] Zheng Zeng, Rodney M. Goodman, and Padhraic Smyth. Learning finite state machines with self-clustering recurrent networks. *Neural Computation*, 5(6):976–990, 1993. eprint: `https://doi.org/10.1162/neco.1993.5.6.976`.

יכולת לממש מונה במצב הפנימי שלו, בעוד שגרסא פופולרית אחרת---ה GRU ---לא יכולה. היכולת הזאת מקנה כוחות חישוביים ל- LSTM של- GRU אין: אנו מוצאים מספר שפות מבוססות ספירה שה LSTM מצליח לזהות במידה הרבה יותר טובה מה GRU , ואף רואים שה LSTM עושה זאת בעזרת מימוש מונים ברורים. אנו מרחיבים על המנגנונים מאוחרי היכולות האלה בעבודתנו.

לבסוף, על מנת להמשיך את קו המחקר הזה עבור טרנספורמרים, אנו מחפשים את המודל שייתן לנו אינטואיציה עליהם כמו שאוטומטים הכווינו אותנו עם רנ"חים. אנו מהר משתכנעים שאוטומטים ומודלים דומים אינם המודל המתאים במקרה הזה. במקום, הסתכלות עמוקה על מבנה הטרנספורמר נותן אינטואיציה של מודל אשר מבצע מספר סופי של פעולות מוגבלות על כל רצף שהוא מקבל כקלט: מעין שפת תכנות חסרת לולאות, עם מספר קטן של פעולות אפשריות. בעבודה המוצגת פה נבנה ונתאר את השפת תכנות הזאת, הפעולות האפשריות שלה, ומספר חישובים מעניינים שניתן לעשות בעזרתה: נציג את השפת תכנות הסימבולית שלנו, RASP. שפה זו מתארת את מנגנוני עיבודי הרצפים השונים העומדים לרשות הטרנספורמר, ומאפשרת אינטואיציה עמוקה על סוג החישובים שניתן לבצע בעזרתו, והעומק ורוחב שיידרש ממנו לבצעם.

תקציר

בשנים האחרונות היה עניין רב בשימוש ברשתות נירונים לעיבוד "רצפים"---כגון טקסט, הקלטות, ורצפים גנטיים---בפרט באמצעות רשתות נירונים חזרתיות (רנ"חים, או Recurrent Neural Networks באנגלית) וטרנספורמרים. רשתות נירונים אלה---פונקציות גזירות ורבות פרמטרים---מאומנות על אוספים גדולים של דוגמאות קלט-פלט ממשימת יעד, ומשיגות תוצאות מרשימות במגוון משימות עיבוד רצפים. כיום הן משומשות להכל מתעתוק דיבור ותרגום אוטומטי ועד חיקוי שפה טבעית באופן חופשי, מסחר אוטומוטי, ועוד. אבל בעוד שהתוצאות שלהן מרשימות מאוד, הייצוג שלהן עמום, וקשה להבין מה בפועל למדה כל רשת.

העבודה בתזה הזאת מתרכזת בעיקר בפירוש של אותן רשתות לעיבוד רצפים. בפרט, נציג שיטות לייצוא חוקים ניתנים לפירוש מרנ"חים מאומנות, ניתוחים של מימושי רנ"חים שונות על מנת להבין אותן טוב יותר, ואף מודל חישובי שבעזרתו ניתן יהיה לבצע את אותם הייצואים וניתוחים עבור טרנספורמים, שהם ארכיטקטורה חדשה, חזקה, ומסובכת יותר.

עבור רנ"חים יש משפחת חוקים שטבעי להתמקד בה: לרנ"חים קשר אינטואיטיבי ברור עם אוטומטים סופיים דטרמניסטיים, ולמעשה הרבה עבודות מנסות לשחזר אוטומטים מתוך רנ"חים בשל אותה סיבה. בעוד שאלגוריתמים קיימים לשחזור (ייצוא) כזה התמקדו בפיצול מרחב המצבים של הרנ"חים לתוך מספר מקבצים סופי, על מנת למפות את המעברים ביניהם, אנו נציג שיטה חדשה לייצוא אוטומטים מרנ"חים שמשלבת גם למידה מדויקת. בפרט, השיטה שלנו תשתמש באלגוריתם המדויק L^* ללמידת אוטומטים, אשר לא הופעל על רנ"חים בעבר. על מנת לעשות זאת, נראה איך לקרב באופן יעיל תשובות לשאילתות ה"שקילות" של L^*, בעזרת שיטות הפיצול ומקבץ מצבים הקיימים עבור רנ"חים. ניתן לראות את האלגוריתם הכולל כתהליך איטרטיבי, בו שתי האלגוריתמים הקיימים משחזרים את האוטומט זה בעזרת זה.

לאחר מכן נרחיב למקרה ממושקל, בו נפתח גרסא ממושקלת לאלגוריתם L^*. בפרט, נעבור ממצב שבו לכל קלט יש רק שתי סיווגים אפשריים---מתקבל או נדחה---למצב שבו ה"סיווג" של קלט הוא התפלגות רציפה מעל האותיות האפשריות הצפויות להמשיך את הקלט---מה שנותן אינסוף סיווגים אפשריים. היות ו L^* בנוי עבור סיווגים בדידים, האתגר כאן יהיה הכנסת סובלנות להבדלים קטנים בין סיווגים לתוך L^*. סובלנות כזו תאפשר לשתי מצבים בעלי תיוגים דומים אך לא זהים עדיין לקבל יחס של מצב יחיד. רק בעזרת סובלנות כזו נוכל להפעיל את L^*, שבגרסתו המקורית מחבר רק מצבים זהים, על רשת נירונים מאומנת וכל הרעש שבה.

כחלק מהעבודה שלנו על שחזור אוטומטים מרשתות מאומנות, אנו מגיעים גם לתובנות מעניינות על ההתנהגות שלהן. בפרט, אנו מגלים שאחת הגרסאות הפופולריות של הרנ"ח, ה LSTM , הוא בעל

המחקר בוצע בהנחייתם של פרופסור ערן יהב (מדעי המחשב, טכניון) ופרופסור יואב גולדברג (מדעי המחשב, בר אילן), בפקולטה למדעי המחשב.

חיבור זה מבוסס על הפרסומים העיקריים הבאים:

Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting automata from recurrent neural networks using queries and counterexamples. In *ICML 2018*, volume 80 of *PMLR*.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting automata from recurrent neural networks using queries and counterexamples (extended version). *Machine Learning, Springer*, 2022.

Gail Weiss, Yoav Goldberg, and Eran Yahav. On the practical computational power of finite precision rnns for language recognition. In *ACL 2018, Volume 2: Short Papers*.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Learning deterministic weighted automata with queries and counterexamples. In *NeurIPS 2019*.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like transformers. In *ICML 2021*, volume 139 of *PMLR*.

הפרסומים הבאים היוו חלק ממחקר הדוקטורט שלי, והם מציגים תוצאות משלימות לחיבור זה:

William Merrill, Gail Weiss, Yoav Goldberg, Roy Schwartz, Noah A. Smith, and Eran Yahav. A formal hierarchy of RNN architectures. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault, editors, *ACL 2020*, pages 443–459.

Daniel M. Yellin and Gail Weiss. Synthesizing context-free grammars from recurrent neural networks. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *TACAS 2021, as part of ETAPS 2021*, volume 12651 of *Lecture Notes in Computer Science*, pages 351–369.

מחברת חיבור זה מצהירה כי המחקר, כולל איסוף הנתונים, עיבודם והצגתם, התייחסות והשוואה למחקרים קודמים וכו', נעשה כולו בצורה ישרה, כמצופה ממחקר מדעי המבוצע לפי אמות המידה האתיות של העולם האקדמי. כמו כן, הדיווח על המחקר ותוצאותיו בחיבור זה נעשה בצורה ישרה ומלאה, לפי אותן אמות מידה.

תודות

אני מודה מעמק ליבי על המימון הנדיב של מחקר זה לטכניון, לפקולטה למדעי המחשב על שם הנרי ומרילין טאוב, למלגת ות"ת, ולגופים הבאים אשר נתנו גרנטים: תכנית FP7 של האיחוד האירופי; הקרן הלאומית למדע, מכון אלן למחקר בינה מלאכותית; מכון ICRI-CI של אינטל; והמועצה האירופאית למחקר (ERC).

רשתות נירונים לרצפים: מבט פורמלי

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

גייל וויס

הוגש לסנט הטכניון — מכון טכנולוגי לישראל
כסלו התשפ"ג חיפה דצמבר 2022

רשתות נירונים לרצפים: מבט פורמלי

גייל וויס