ON EXPRESSIVENESS, INFERENCE, AND PARAMETER ESTIMATION OF DISCRETE SEQUENCE MODELS

by

Chu-Cheng Lin

A dissertation submitted to Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

October, 2022

Abstract

Huge neural autoregressive sequence models have achieved impressive performance across different applications, such as NLP, reinforcement learning, and bioinformatics. However, some lingering problems (*e.g.*, consistency and coherency of generated texts) continue to exist, regardless of the parameter count. In the first part of this thesis, we chart a taxonomy of the expressiveness of various sequence model families (§ 3). In particular, we put forth complexity-theoretic proofs that string latent-variable sequence models are strictly more expressive than energy-based sequence models, which in turn are more expressive than autoregressive sequence models. Based on these findings, we introduce residual energy-based sequence models, a family of energy-based sequence models (§ 4) whose sequence weights can be evaluated efficiently, and also perform competitively against autoregressive models. However, we show how unrestricted energy-based sequence models can suffer from uncomputability; and how such a problem is generally unfixable without knowledge of the true sequence distribution (§ 5).

In the second part of the thesis, we study practical sequence model families and algorithms based on theoretical findings in the first part of the thesis. We introduce

neural particle smoothing (§ 6), a family of approximate sampling methods that work with conditional latent variable models. We also introduce neural finite-state transducers (§ 7), which extend weighted finite state transducers with the introduction of mark strings, allowing scoring transduction paths in a finite state transducer with a neural network. Finally, we propose neural regular expressions (§ 8), a family of neural sequence models that are easy to engineer, allowing a user to design flexible weighted relations using Marked FSTs, and combine these weighted relations together with various operations.

Primary Reader and Advisor: Jason Eisner

Secondary Readers: Brian Roark, Matt Post

Acknowledgments

First and foremost, I would like to thank Dr. Jason Eisner for being my academic advisor. Jason is one of the most brilliant people I have known. Like many of my labmates (or *Argonauts*), when I first joined the Argo lab, I was awestruck by the legendary — and legendarily long — technical emails Jason wrote to communicate with his collaborators and students. Some of the first long emails are still very relevant to later chapters in this thesis.

Besides my advisor, I would also like to express my deepest appreciation to Dr. Brian Roark and Dr. Matt Post for serving as members of my thesis committee, and for their encouragement and insightful comments. They went beyond the call of duty to support me on my journey toward this degree.

I have been inspired by many faculty at Johns Hopkins. Specifically, I would like to thank Dr. Kevin Duh, Dr. Ben Van Durme, Dr. Sanjeev Khudanpur, Dr. Joanne Selinski, and Dr. Scott Smith for their conversations (where the topics range from very technical conversations to advice on finding financial support) and help.

I have been very fortunate to be joined by many fellow Argonauts during my PhD study: they are Nicholas Andrews, Jacob Buckman, Ryan Cotterell, Li (Leo) Du, Nathaniel

(Wes) Filardo, Matthew Francis-Landau, Juneki Hong, Xiang (Lisa) Li, Xiaochen Li, Chenxi Liu, Zhichu (Brian) Lu, Becky Marvin, Hongyuan Mei, Sabrina Mielke, Guanghui Qin, Pushpendre Rastogi, Nanyun (Violet) Peng, Adi Renduchintala, Darcey Riley, Tim Vieira, Shijie Wu, Akshay Srivatsan, Steven Tan, Adam Teichert, and Mozhi (Miles) Zhang. Technical discussions in this group have always been deep, and also very rewarding. I have also enjoyed the company of folks in the wider Center for Language and Speech Processing (CLSP) community. They are: Tongfei Chen, Yunmo Chen, Shuoyang Ding, Seth Ebner, Dongji Gao, Lv Hang, Huda Khayrallah, Ke Li, Chunxi Liu, Xutai Ma, Marc Marone, Arya McCarthy, Adam Poliak, Pamela Shapiro, Suzanna Sia, Shuo Sun, Yiming Wang, Winston Wu, Patrick Xia, Hainan Xu, and Xiaohui Zhang.

I also appreciate the institutional support from CLSP and the Department of Computer Science. Many thanks to Ruth Scally, Jennifer Linton, Zack Burwell, and Kim Franklin for providing help on many occasions.

During my internship at Microsoft Research, I have learned much from Dr. Patrick Pantel and Dr. Michael Gamon. Their industry perspectives are invaluable. I am also grateful to Dr. Aaron Jaech, who really went out his way to find crucial resources during my internship at Facebook AI, which forms as the basis of §§ 3 and 4 in this thesis. Finally, I would like to extend my sincere thanks to Dr. Xing Fan, for her excellent mentorship during my last internship at Amazon Alexa, which also provided a much needed financial support for my last semester's tuition and living expenses.

Last but not least, I would like to thank my family and friends for their unconditional

love and support through this special period of my life.

Previous Publications

Portions of this thesis have been published in the following:

- Chu-Cheng Lin and Jason Eisner. 2018. Neural Particle Smoothing for Sampling from Conditional Sequence Models. In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers), pages 929—941, New Orleans, Louisiana. Association for Computational Linguistics. URL: https://aclanthology.org/N18-1085/

- Chu-Cheng Lin, Hao Zhu, Matthew R. Gormley, and Jason Eisner. 2019. Neural Finite-State Transducers: Beyond Rational Relations. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pages 272-–283, Minneapolis, Minnesota. Association for Computational Linguistics. URL: https://aclanthology.org/N19-1024/

- Chu-Cheng Lin, Aaron Jaech, Xin Li, Matthew R. Gormley, and Jason Eisner. 2021.

Limitations of Autoregressive Models and Their Alternatives. In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 5147-–5173, Online. Association for Computational Linguistics. URL: https://aclanthology.org/2021.naacl-main.405/

- Chu-Cheng Lin and Arya D. McCarthy. 2022. On the Uncomputability of Partition Functions in Energy-Based Sequence Models. In Proceedings of the 10th International Conference on Learning Representations, Online. URL: https://openreview.net/forum?id=SsPCtEY6yCl

Contents

List of Tables

List of Figures

Chapter 1

Introduction

String transduction refers to tasks that take a string **x** as input, and output another string **y** as output. Many NLP tasks can be described as transduction tasks. Examples include machine translation (*e.g.*, translate the English sentence '*where is the train station?*' into French '*où est la gare?*') (Sutskever, Vinyals, and Le, 2014), text summarization (Shi et al., 2018b), parsing (Vinyals et al., 2015), generation (Mikolov et al., 2010), and slot filling (Yao et al., 2013).

1.1 Modeling key dependencies in string transduction tasks

For many NLP string transduction tasks, we must model complex interactions among symbols in the input string, across input and output strings, and among interactions among

symbols in output strings. These interactions can be co-occurrence patterns (some symbols may be likely/unlikely to co-occur), ordering patterns (some symbols may be likely/unlikely to appear before some other symbols), among many others. Here we use **slot filling** as an example, to illustrate how these dependencies arise in an NLP task.

The goal of slot filling is to identify from a running dialog different slots, which correspond to different parameters of the user's query (Huang, Chen, and Bigham, 2017). Imagine a user is asking a digital assistant to buy goods at a specific shop with their coupon code, *e.g.*, '*buy two light bulbs at Best Buy using my BOGO offer.*' Within the digital assistant, this command will be mapped to a particular string that specifies the actions required by the agent, something like '`buy␣P_98_BOGO[two␣light␣bulbs]␣at␣S_58[Best␣Buy]␣using␣my␣PROMOTION_BOGO_BEST_BUY[BOGO]␣offer`'[1][2] For the task of slot filling, there are dependencies:

Among input string symbols: input strings in slot filling tasks are natural language utterances, which are known to display various structures (Chomsky, 1957; Fillmore and Baker, 2010; Grice, 1975). Besides being grammatical, human communication with digital assistants may exhibit specific patterns — for example, many of these utterances may be commands. Also note that in our 'buy light bulbs' example above the user buys bulbs from an electronic store (and not from, say, McDonald's).

Many of the patterns that happen in human utterances are neither mandatory nor prohibitive: some patterns are likely to show up, while others are unlikely. This in

[1] See §1.2 for details on the format of such sequences.
[2] We use ␣ to denote space symbols when appropriate.

turn makes some strings more likely human utterances (if they contain common patterns) than others. We say these strings, along with their weights, form a **weighted language** (§2.2).

Between symbols on input and output strings: input and output strings can be *monotonically aligned* in the task of slot filling (for example, the slot-filling example we give in §1.2). Many monotonically aligned transduction tasks, such as speech recognition (Mohri, Pereira, and Riley, 2008), are modeled using weighted finite-state transducers (a special case of linear factor graph grammars).

Among output string symbols: Just as there can be complex interaction between symbols on the input string, the output string can exhibit patterns as well. In our slot filling example, the noun phrase '*two light bulbs*' is annotated with a special promotion type `P_98_BOGO`, which is only valid in the presence of the slot `PROMOTION_BOGO_BEST_BUY`. On the other hand, the slot `PROMOTION_BOGO_BEST_BUY` is only valid in the presence of slot `S_58`. While our slot filling example involves *hard* constraints (slot annotations are either valid or invalid), other string transduction tasks may only have preferences: for example, text summarization tasks may prefer output strings that are more pragmatically informative (Shen et al., 2019).

These dependencies make string transduction a potentially challenging task in general. In practice, the task of slot filling was historically modeled with weighted finite-state transducers (a special kind of low-treewidth factor graph grammar §2.1.1) (Raymond and

Riccardi, 2007), due to the monotone alignment between input and output symbols as we described above. With the advent of neural sequence models, it is now often addressed using seq2seq models (Yao et al., 2013). In fact, seq2seq models (Sutskever, Vinyals, and Le (2014), §2.1.3) and low-treewidth factor graph grammars (§2.1.1) are currently the most popular string transduction paradigms across most string transduction tasks.

However, these approaches each have key shortcomings, which make them unsuitable for modeling the example above. First, WFSTs and other low-treewidth factor graph grammars will make a Markov assumption that preclude global well-formedness constraints on both input and output strings (*i.e.*, they cannot capture dependency among symbols on the input and output strings well). Seq2seq models on the other hand can capture the dependency among input string symbols; however they cannot capture the dependency among output string symbols without using asymptotically superpolynomially many parameters (§ 3). Moreover, seq2seq models are not inherently modular: model reuse is generally not straightforward.

In this thesis, we propose neural regular expressions (NREs), which generalize both seq2seq and low-treewidth factor graph grammars, and are powerful enough to correctly model difficult sequence-to-sequence mapping problems such as the one illustrated above. In the following section, we first describe the mechanisms of existing approaches, and why they will not work for the example we gave above.

1.2 Problems with existing approaches

In this section we continue to use the slot filling task as our string transduction example, to illustrate the pros and cons of linear factor graph grammars and seq2seq models, and how it motivates our new paradigm.

Specifically, we assume the task is to annotate (transcribed) voice commands for digital assistants. A digital assistant (also known as virtual assistant or voice assistant) is a program that interprets human speech. Users can instruct their digital assistants to do various tasks (*e.g.*, send messages, make phone calls, control home automation devices, etc.) (Hoy, 2018). To actually execute a command, a digital assistant classifies the command's underlying *intent* — which is backed by a function call — and also annotates parts of the (transcribed) voice command, as arguments to the function call. Finally, the digital assistant calls the function, with supplied arguments.

Intent classification and argument annotation together reduce to slot filling. As a concrete example: the transcribed utterance '*buy cereals at Whole Foods*' may get slot-filled as '`BUY[buy]␣P_85[cereals]␣at␣S_31[Whole␣Foods]`'. In this example, the intent is `BUY`, with `P_85[cereals]` and `S_31[Whole Foods]` as arguments.[3] As we described above, the digital assistant would subsequently run the program associated with `BUY`, with these arguments. Such programs are also known as **skills**. Under this design, to map a voice

[3] In our examples the arguments take a format `ARGUMENT_TYPE[gloss]`, where `ARGUMENT_TYPE` is specified by the annotation scheme beforehand, and `gloss` is the span of utterance that is identified as an `ARGUMENT_TYPE`.

command to its skill, we must first transcribe it into text, then slot-fill the text into an appropriate intent, along with valid arguments.

1.2.1 The weighted regular expressions approach

Our first attempt at slot filling is using (weighted) regular expressions, or equivalently weighted finite-state machines. Weighted finite-state machines are a special kind of low-treewidth factor graph grammar (§2.1.1). Extracting information from natural language using finite-state machines has a very long history (Appelt et al., 1993). To recognize and extract useful information from patterns like *'buy* **product name** *at* **shop name***'*, we write the following regular expression BuyGrocery in RE 1.2.1 :[4][5]

RE 1.2.1: BuyGrocery

(buy␣:BUY[buy]␣)Product(at␣:at␣)Shop

where Product and Shop are gazetteers — lists of some predefined named entities — also expressed as regular expressions: for example, they can be

RE 1.2.2: Product

(cereals:P_85[cereals])|(milk:P_28[milk])|...

and

[4]We adopt a PCRE-like formalism that allows transduction: (*a*:*b*) means transducing the finite state machine *a* into *b*. We need a novel formalism because conventional regular expression formalisms, such as vanilla PCRE, only support acceptors.

[5]In this work, we use SmallCaps to indicate a regular expression and the finite state machine it defines.

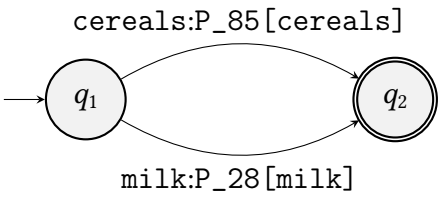
RE 1.2.3: SHOP

(Whole␣Foods:S_31[Whole␣Foods])|(Giant:S_46[Giant])|...

respectively.

A regular expression defines a finite-state transducer. For example, RE 1.2.2 defines the following machine:

FST: An FST defined by RE 1.2.2



...

(other arcs omitted)

A user may ask their digital assistant to buy things, using different constructions. We generally want to recognize the most frequent ones, out of all these constructions, while implementing our skill. Let's suppose that we wish to capture commands of the pattern *'go to* **shop name** *and buy* **product name***'*:

RE 1.2.4: BUYGROCERY2

(Go␣to␣:Go␣to␣)SHOP(and␣buy␣:and␣BUY[buy]␣)PRODUCT

Note that we reuse REs 1.2.2 and 1.2.3 in RE 1.2.4. In practice, these gazetteers are likely automatically generated by queries into large databases (rather than a manually curated list of named entities).

REs 1.2.1 and 1.2.4 have their corresponding finite-state transducers. These automata capture natural language sentences (*e.g. 'Buy cereals at Whole Foods'*), and annotate their slots (*e.g.* `BUY[...] ...P_85[...] ...S_31[...] ...`).

However, REs 1.2.1 and 1.2.4 are *brittle*: they do not handle typos (when a user input their command on a keyboard) or ASR errors (when a user said their command to a microphone). Of course, we can add common typos and ASR errors directly into gazetteers RE 1.2.2 and RE 1.2.3 (and have them transduce into the correct canonical names) — but doing so can introduce additional brittleness problems (where do we get a list of common ASR errors? How should we maintain that list?). Alternatively, we can make our finite-state machines *weighted*, enabling them to consider inexact matches (where strings that are closer to the right answer get higher scores). In other words, we wish to recognize and transduce strings that are *similar* enough. To account for similarity systematically, we not only change our regular expressions into ***weighted* regular expressions (WREs)**, but also introduce a new Similar WRE:[6]

WRE 1.2.1: BuyGroceryWeighted: a *weighted* version of RE 1.2.1

(Buy$_{\sqcup}$:buy$_{\sqcup}$/1.)(.+◦Similar◦Product)(at$_{\sqcup}$:at$_{\sqcup}$/1.)(.+◦Similar◦Shop)

[6] We use the notation '(a:b/w)' to indicate that 'string a transduces to string b with weight w'.

where SIMILAR is a pre-engineered (or pre-trained) weighted FST that transduces all non-empty strings to all non-empty strings, with similar-sounding string pairs getting higher weights[7]:

WRE 1.2.2: SIMILAR

((a:e/.7)|(a:a/.9)|(a:c/.01)|(a:ϵ/.2)|...|(z:s/.4)...)+

Note that we compose the same regular expression WRE 1.2.2 against REs 1.2.2 and 1.2.3.[8] By reusing WRE 1.2.2 in different contexts, the weighted regular expression is (hopefully) resilient to typos and ASR errors. Note that WRE 1.2.2 can be used in future weighted regular expressions we write, too.

WREs define real-weighted relations between input and output strings: WRE 1.2.2 defines a weighted relation between non-empty string pairs; and WRE 1.2.1 defines a weighted relation between strings of the pattern '`Buy .+ at .+`', and all valid 'buy' action strings. To get the relation weight between string pair $(\mathbf{x}, \mathbf{y})$ under a WRE τ, one can compose the two strings $\mathbf{x}$ and $\mathbf{y}$ against τ, to get a new WRE $\mathbf{x} \circ \tau \circ \mathbf{y}$ that only recognizes a single string pair $(\mathbf{x}, \mathbf{y})$. Finally, the relation weight between string pair $(\mathbf{x}, \mathbf{y})$ under τ is the sum of all paths' $\mathbf{z}$ weights $w_\mathbf{z}$: $\sum_{\mathbf{z}:\mathbf{z} \text{ is an accepting path of } \mathbf{x}\circ\tau\circ\mathbf{y}} w_\mathbf{z}$, which can be obtained in polytime using dynamic programming techniques.

[7] The weights we give here are not necessarily close to those used in real production systems.

[8] We define that when we 'coerce' a regular expression into a weighted one, every string-to-string transduction (*i.e.* (a:b) where a and b are both strings) has weight 1.

Taking advantage of the expressiveness of WREs over REs. WRE 1.2.1 is likely more robust to typos and ASR errors than RE 1.2.1: the *weighted* relation defined by WRE 1.2.2 (used in WRE 1.2.1 as a component) can assign differently graded weights to different string pairs according to their similarity, whereas it is either 1 or 0 in the case of RE 1.2.1. As a result, WRE 1.2.1 can deal with typos and ASR errors with a much shorter/more compact topology, than an equivalent unweighted regular expression, if we were to modify RE 1.2.1 to cope with typos and ASR errors — the unweighted RE equivalent would have to memorize similar-sounding string pairs in a huge gazetteer. On the other hand, suppose we already had such a huge unweighted 'similarity gazetter' RE curated using human knowledge, we have the option of 'weighting it' under the more expressive WRE formalism, and fine-tune the new WRE's weights on empirical data. Nonetheless, it is quite likely that such a WRE will not be competitive against the much smaller WRE 1.2.2 on held-out data, if we fine-tune WRE 1.2.2's weights on empirical data as well. The reason is that the lifted similarity gazetteer has to assign zero weight to many (in fact infinitely many) string pairs. But WRE 1.2.2 can just assign low weights to those. So when it comes to really noisy data (*i.e.* much garbled ASR text), the weighted similarity gazetteer would be forced to assign a zero weight to a string pair, while WRE 1.2.2 assigns a low but non-zero weight to it, which can still be compared against other weighted string pairs. This observation suggests that while richer formalism are often strictly more powerful than restrictive ones (*e.g.* WREs versus REs), it may be preferable to start from scratch to take advantage of the added expressiveness. We will make an analogy in §1.3.1.

Despite the added expressiveness when compared to unweighted REs, WREs are still subject to the Markov assumption. Specifically, our weighted regular expression cannot capture the tendency that certain products are more likely to be sold at certain shops: for example, 'MOM's homeopathetic flu remedies' is probably sold at 'MOM's organic market,' rather than 'Charles Village discount mart', because an item is likely to be sold in a shop that shares the same brand name; moreover, organic supermarkets may be more prone to sell homeopathetic remedies than a discount mart. In other words, the semantic similarity between item and shop names goes beyond superficial edit distance metrics.

The inability of our weighted regular expression (or equivalently, WFSTs) to capture semantic similarities between item and shop names ultimately comes from the Markov assumption of WFSTs — all information from the Shop segment will have been forgotten when we reach segment Product.

There are ways to patch up for a weighted regular expression's power limitation. One possibility is to divide skills by the shops: MOM's Organic Market has its skills (that only recognize their products) and Charles Village Discount Mart has its own, too. We can combine all these skills into a single 'buy' mega-skill. But that obviously will not scale well — the corresponding finite-state machine will have a size that grows exponentially in the number of shops. And the resultant skill will not generalize to a new shop (*i.e.*, unseen entity names will not be recognized).

1.2.2 The seq2seq approach

A seq2seq model (§2.1.3) explicitly parametrizes an autoregressive distribution over output strings, conditioned on an input string. Seq2seq models do not make a Markov assumption between segments of the output string. Therefore, a seq2seq model trained on input-output pairs like (*'buy cereals at Whole Foods'*, `BUY ...S_31 ...P_85 ...`) will likely do a decent job on filling these two slots, and can learn the tendency that store-branded items are more likely to be sold at some particular stores, all without writing any explicit rules. It can also generalize to speech act variations better than WFSTs given appropriate training data (*e.g.* suppose we also trained on input-output pairs that display alternative speech acts, such as (*'go buy cereals at Whole Foods'*, `BUY ...S_31 ...P_85 ...`)).

Unlike WFSTs, vanilla seq2seq models do not have a baked-in inductive bias of a monotonic alignment between input and output strings. There does not appear to be a straightforward way to constrain the format of output strings either: in our slot filling example, we require a valid output string to be a copy of the input string, with appropriate intent and argument slot markups. Therefore, more training data may be required to learn a vanilla seq2seq model that captures such behavior.[9][10]

A downside of seq2seq models in general is that there is no obvious way to reuse

[9] We note that there are extensions to the vanilla seq2seq models that address these issues (monotonic alignment and copying of fragments in the input string). We discuss these extensions in §2.1.3.

[10] We also note that there are *post hoc* constrained decoding methods that enforce lexical constraints during beam search (Hokamp and Liu, 2017; Post and Vilar, 2018; Hu et al., 2019). But these methods only guarantee *local* constraints, and may not be powerful enough to ensure output string wellformedness (*e.g.*, there can be only one product ID in a slot-filled string).

prebuilt modules. For example, suppose we need to develop a new skill that the intent and arguments of '*let me know when Cotton Candy grapes are in stock at Whole Foods*' as 'NOTIFY-IN-STOCK[let␣me␣know] ...P_103 ...S_31 ...,' we would have to collect data for this new task, and train a new seq2seq model for it. On the other hand, under the WFST approach, it is relatively easy to write a weighted regular expression (WRE 1.2.3) that reuses existing weighted regular expressions:

WRE 1.2.3: NotifyInStock

(Let␣me␣know␣when␣:notify-in-stock␣)

(.+◦Similar◦Product)(are␣in␣stock␣at␣:ϵ)(.+◦Similar◦Shop)

1.2.3 Common problem: cannot guarantee output well-formedness

Both WFSTs and seq2seq models suffer from a limited expressiveness problem: neither can guarantee that the output strings conform to given constraints, even when such constraints can be easily and efficiently checked (*i.e.*, wellformedness can be evaluated in polytime).

We consider constraints exhibited by the 'BOGO promotion' skill introduced in §1.1. We need to identify product and shop slots in an utterance for the BOGO skill (just like the 'buy' skill). However, unlike the buy skill, only certain products are eligible under the BOGO offer. Not all shops participate in the BOGO promotion, either. Finally, the promotion

is handled separately from the ordinarily 'buy' skill, and has different intent and slot codes.

Can we write a WRE for BOGO commands? As we discussed in §1.2.1, compact WREs cannot capture long term constraints of the BOGO offer (*i.e.*, only certain products in combination with certain shops are eligible). Therefore WREs cannot guarantee output strings are well-formed.

How about seq2seq models? Due to their autoregressive nature, seq2seq models cannot backtrack when they incrementally build up the output string. They also must consistently make optimal choices during the output-building process. For example, suppose a seq2seq model has partially built an output prefix '`Buy  `' from ASR-transcribed input string 'Buy two light bulbs at Best Buy Bangkok'. Should the next piece be a BOGO slot code (*e.g.*, `P_98_BOGO`), or a regular product code (*e.g.*, `P_98`)? This choice will imply whether the utterance is classified as a 'BOGO' skill. If the seq2seq model chooses `P_98_BOGO`, it can only choose a `PROMOTION_BOGO`; on the other hand, if `P_98` is instead chosen, then `PROMOTION_BOGO` will not be a valid intent, and something else (*e.g.*, `BUY`) must be used as the intent. Therefore, for a seq2seq model to confidently choose `P_98`, it must be confident that it *will not* get into a scenario, where such a choice is impossible.

But to make such a guarantee, we would need to consider all possible suffixes to our already-built prefix. For example, what if 'Best Buy Bangkok' is not a valid shop name, and the user actually meant to say 'Best Buy BOGO'? We will show that autoregressive models generally cannot consider *all* possible suffixes (§ 3), unless they are unreasonably slow (*i.e.*, evaluation takes superpolynomial runtime) or use unreasonably many parameters (*i.e.*,

have superpolynomial parameter size).

1.3 Thesis outline

In this thesis, we propose **neural regular expressions** (NREs) as an alternative paradigm. At a very high level, NREs generalize both seq2seq and graphical models, and get the best of two worlds: like weighted regular expressions, they have engineer-able topologies. And like seq2seq models, they are very expressive, being families of unbounded-treewidth graphs. They even exceed the expressiveness of seq2seq models, being able to model all output structures where wellformedness can be evaluated in polytime.

1.3.1 *Neuralization* of finite state machines

Here we give a very high-level overview of the formalism and operationalization of NREs, using digital assistant skills as our examples.

An NRE has a regular expression component that looks very similar to WREs. The only difference is that each symbol transduction is now associated with zero, one, or multiple **marks**, instead of a single real weight. And concatenating together all marks on a path, they form a **mark string** (denoted as a *space-delimited italic string*). We call such regular expression components **Marked Regular Expressions (MREs)**.

We again use a grocery shopping skill as an example. A marked version of RE 1.2.1 can

be written as:[11]

MRE 1.3.1: BuyGroceryMarked: a *Marked* version of RE 1.2.1

(Buy␣:buy␣/*buy*)

(.+◦SimilarMarked◦ProductMarked)

(at␣:at␣/ *location*)(.+◦SimilarMarked◦ShopMarked)

and Product redefined as an MRE as:

MRE 1.3.2: ProductMarked: a *Marked* version of RE 1.2.2

(cereals:P_85/*product_cereals food breakfast*)|(milk:P_28/*product_milk food dairy*)|...

ShopMarked is marked similarly as MRE 1.3.2. Finally, SimilarMarked is a marked version of WRE 1.2.2:

MRE 1.3.3: SimilarMarked: a *Marked* version of WRE 1.2.2

((a:e/*i-a o-e*)|(a:a/*i-a o-a*)|(a:c/*i-a o-c*)|(a:ϵ/*i-a* ϵ)|...|(z:s/*i-z o-s*)...)+

One possible mark string from NRE 1.3.1 (again, obtained by concatenating all mark symbols along the transduction path) is '*buy i-c o-c i-e o-e ... cereals food breakfast location i-w o-w i-h o-h ... whole_foods*'. Mark strings in MREs can be seen as latent variables: each accepting path $\mathbf{z}$ in an MRE τ will be associated with a (possibly non-unique) mark string. For specific $(\mathbf{x}, \mathbf{y})$ pairs, we are interested in paths in τ that transduce $\mathbf{x}$ into $\mathbf{y}$. Such paths

[11] Here we extend our transducing regular expression formalism again, to denote marks after '/'.

can be obtained by composing τ against the input and output strings (denoted by $\mathbf{x} \circ \tau \circ \mathbf{y}$). And an NRE τ defines the score of $(\mathbf{x}, \mathbf{y})$ as the sum of *mark string scores* on these paths:

$$\text{score of pair } (\mathbf{x}, \mathbf{y}) \text{ according to } \tau = \sum_{\mathbf{z}\,:\,\mathbf{z} \text{ is an accepting path of } \mathbf{x} \circ \tau \circ \mathbf{y}} G_{\theta}(\mathbf{z}), \tag{1.1}$$

where G_θ is a non-negative function that scores marks on $\mathbf{z}$. In equation (1.1) we define a weighted relation between input and output strings. Alternatively, we denote the weighted relation in equation (1.1) as NRE 1.3.1:

NRE 1.3.1: *Neuralization* of MRE 1.3.1

N[BuyGroceryMarked, G_θ]

We say G_θ **neuralizes** MRE 1.3.1 as NRE 1.3.1. Like WREs (*e.g.* WRE 1.2.1), an NRE defines a real-weighted relation between input and output strings. However, with an expressive enough family of G_θ, NREs are strictly more powerful than WREs: they do not have to be subject to the Markovian assumption. In other words, with an appropriate G_θ that knows to score a mark string $\mathbf{z}$ highly, if $\mathbf{z}$ has some possibly long-range mark pattern (*e.g.* $\mathbf{z}$ = '…**MOM's** *homeopathetic_flu_remedies* …**MOM's** *organic_market* …'), we will be able to capture the tendency that a store likely sells own-branded goods with NRE 1.3.1, unlike WRE 1.2.1.

Taking advantage of the expressiveness of NREs over WREs. Note that we do not need the partition-by-shop fix suggested in §1.2.1 to capture store-product correlation in

NREs: there is no need to make use of an exponentially large finite-state machine to keep track of long-distance interaction, given a powerful enough family of G_θ. More generally, just as huge 'ASR error' gazetteers are no longer necessary as we go from REs to WREs, huge WREs which are huge because of Markov assumptions can be made much smaller by using the more expressive NREs. Since NREs are strictly more powerful than WREs, we will be able to imbue prior knowledge from previously built weighted regular expressions into our NREs via careful regularization.

1.3.2 Combining NREs

The use of non-Markovian path scoring functions opens up many possibilities. For example, different subsequences of a path $\mathbf{z}$ (from different parts of a finite-state machine that it goes through) can be scored by different scoring functions. In this thesis, we achieve this goal, by combining different NREs that use their own scoring functions.

To reuse our grocery NRE running example, a practitioner may opt to separately neuralize components that recognize products and shops from our grocery NRE, in order to reuse those components as NREs in other skills. We write such a combination of different NREs as:

NRE 1.3.2: Alternative neuralization of MRE 1.3.1

N[

(Buy␣:buy␣/*buy*)

SimilarProductNeural

(at␣:at␣/ *location*)

SimilarShopNeural,

$G_\theta{}^3$].

where SimilarProductNeural is an NRE:

NRE 1.3.3: SimilarProductNeural

N[(. +∘SimilarMarked∘ProductMarked), $G_\theta{}^1$]

and SimilarShopNeural is an NRE as well:

NRE 1.3.4: SimilarShopNeural

N[(. +∘SimilarMarked∘ShopMarked), $G_\theta{}^2$]

Comparing NRE 1.3.1 with NRE 1.3.2, we see that the biggest difference is that whereas NRE 1.3.1 has MRE 1.3.1 neuralized as a whole, NRE 1.3.2 has the two MREs

(. +∘SimilarMarked∘ProductMarked)

and

(. +∘SimilarMarked∘ShopMarked)

neuralized *separately* in NREs 1.3.3 and 1.3.4, before concatenating them with other MRE components ((Buy␣:buy␣/*buy*) and (at␣:at␣/ *location*) in this case) and neuralizing again. NREs 1.3.3 and 1.3.4 can thus be reused in other composite NREs, and possibly trained

together with them, to share statistical strength with other contexts that require fuzzy matching of product and store names.

The set of NREs is closed under common finite state operations, such as concatenation (illustrated in NRE 1.3.2), union, and composition (§ 8) — in other words, NRE 1.3.2 can be written as

NRE 1.3.5: Alternative presentation of NRE 1.3.2

N[AnotherMRE, $G_\theta{}'$]

where AnotherMRE is an MRE (essentially MRE 1.3.1 with some inserted 'control' marks), and G'_θ is a path scoring function. The equivalence of NRE 1.3.2 and NRE 1.3.5 (and in general the closure of NREs under finite-state operations) allows us to express any weighted relation under NREs as a sum-product, which makes inference straightforward.

1.3.3 Model capacity and computability concerns of parametrization

NREs are powerful. NREs as we have described is an *extremely* expressive formalism. Our only requirements for the scoring function G_θ is that G_θ must be a polytime non-negative function (in mark string length), so it is possible to parametrize G_θ as neural network models, and that approximate inference is not prohibitively slow. This effectively makes NREs much more expressive than most bounded-treewidth factor graph-based

formalisms, since NREs can be used to define a family of factor graphs that have unbounded treewidths.

NREs are a strict generalization of both WREs and seq2seq models. Specifically, we formally show NREs are more powerful than either WREs or seq2seq models in the following theoretical sense:

> There exists a weighted relation that cannot be modeled by any polysize parametric families of WREs and seq2seq models — each parametric model M_i in family, which has parameter vector with size $O(\text{poly}(i))$, only has to model relations of strings shorter than i. But this weighted relation can be modeled by one finite-size NRE model.

In particular, neither polysize families of WREs nor seq2seq models can capture weighted relations which prefer output strings that conform to polytime checkable structural constraints (§1.2.3).

NREs may be *too* powerful. However, with great model expressiveness come great parameter estimation challenges. With an MRE τ, the weighted relation $N[\tau, G_\theta](\mathbf{x}, \epsilon)$ can be used to model *any* recursively enumerable language $L = \{\mathbf{x} : \mathbf{x} \in L\}$. Moreover, with unbounded treewidths, inference is not only intractable, but generally uncomputable and inapproximable. To see this, consider NRE 1.3.6

NRE 1.3.6

N[Program, G_{program}]

where Program is an MRE:

MRE 1.3.4

$((\epsilon{:}\epsilon/0)|(\epsilon{:}\epsilon/1))*$

We use ϵ to denote the empty string ''. Here we sketch how G_{program} can be defined in a way, such that $N[\textsc{Program}, G_{\text{program}}](\epsilon, \epsilon)$ is uncomputable (Proposition 7.3.1 is a more formal statement): we let $G_{\text{program}}(\mathbf{z}) = 1$ if $\mathbf{z}$ encodes a proof that ZFC is not consistent (in some formal language), and $G_{\text{program}}(\mathbf{z}) = 1/3^{|\mathbf{z}|+1}$ otherwise. G_{program} can be built as a finite-size 1-encoder-layer 4-decoder-layer Transformer network (§2.3.2).

Our construction of G_{program} implies if there is no proof that ZFC is inconsistent, then $N[\textsc{Program}, G_{\text{program}}](\epsilon, \epsilon) = 1$. Assuming the ZFC axioms as our axiomatic set, there indeed will not be a $\mathbf{z}$ such that $G_{\text{program}}(\mathbf{z}) > 1$; on the other hand, Gödel's second incompleteness theorem implies the we cannot prove there will not be such a proof either — that is, we cannot prove that $N[\textsc{Program}, G_{\text{program}}](\epsilon, \epsilon) = 1$. Therefore, there will be no provably correct algorithm (either randomized or deterministic) that exactly computes $N[\textsc{Program}, G_{\text{program}}](\epsilon, \epsilon)$ (and as we will later show in § 5, not even only approximately).

Even when we want to learn NREs 'simpler' than NRE 1.3.6, having a too expressive parametric family $\boldsymbol{\Theta}$ can still be problematic, since fundamental tasks such as likelihood-based model selection can be undecidable (§ 5): just as there exists $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ that leads to uncomputable weights, deciding whether a given $\boldsymbol{\theta}$ leads to uncomputable weights is itself undecidable.

We need to parametrize NREs in a way that limits their power. We certainly do not want to work with a parametric family where likelihood-based model selection is generally impossible. On the other hand, we are reluctant to give up the unbounded treewidth promise, which would significantly cripple expressiveness.

We make a parametrization choice, where likelihood-based model selection and parameter estimation are always feasible, while still ensuring the formalism strictly outperforms seq2seq models in expressiveness. Specifically, we parametrize *primitive* scoring functions — scoring functions that are not composed using other scoring functions — as *autoregressive models*: we define $G_\theta(\mathbf{z})$ to be of the form

$$G_\theta(\mathbf{z}) \triangleq \prod_{t=1}^{|\mathbf{z}|+1} p_\theta(z_t \mid z_{<t}), \tag{1.2}$$

where $z_{<t} \triangleq (z_0 \ldots z_{t-1})$, $z_0 \triangleq \textsc{bos}$, and $z_{|\mathbf{z}|+1} \triangleq \textsc{eos}$. In other words, a primitive scoring function defines a (normalized) probability distribution over mark strings.

In this thesis, we formally study the consequences of our parametrization choice, making connections between sequence modeling and computability/complexity theories. We would like to answer the following questions:

- Do autoregressive sequence models lose expressiveness, compared to the more general energy-based sequence models?
- Do we gain expressiveness over ordinary autoregressive models, by having string latent variables (*i.e.* mark strings)?

- Do NREs with autoregressive primitive scoring functions define computable weighted relations? Is computability of the relation weight closed under finite-state operations on NREs?

This thesis attempts to answer all these questions. In particular, it turns out that in the case of no-latent-variable sequence models, an autoregressive parametrization of the form equation (1.2) would be very restrictive — to ensure 'good' strings are always weighted higher than all 'bad' strings, where good/bad is decided by some polytime algorithm, autoregressive models need to have their parameter vectors grow *superpolynomially* in the maximum sequence length one wishes to model. This theoretical result has consequences beyond NREs: in general autoregressive models cannot check the goodness of strings, regardless of their parametrization, as long as they only make use of polytime compute.

As for our second question, we show that introducing string latent variables (in the manner of NREs) patches the aforementioned problem of autoregressive models. Furthermore, we show that with string latent variables, autoregressive sequence models have a very high capacity: they are able to model all languages $\in$ NP as distribution supports, exceeding that of no-latent-variable autoregressive sequence models (which can have some, but not all languages in P as support), and energy-based sequence models (which can have any language in P as support).

Finally, we have been able to show that NREs with autoregressive primitive scoring functions, and subsequent NREs built from these NREs, using finite-state operations, have computable partition functions. One implication is that likelihood-based model selection is

at least possible — even though the exact quantities of partition functions are likely still intractable, and must be approximated.

1.4 Thesis organization

In this thesis, we

- chart a taxonomy of the expressiveness of various sequence model families (§ 3, Lin et al. (2021)). In particular, we put forth complexity-theoretic proofs that **string latent-variable sequence models** are strictly more expressive than **energy-based sequence models**, which in turn are more expressive than **autoregressive sequence models**. Based on these findings, we
- introduce **residual energy-based sequence models**, a family of energy-based sequence models (§ 4, Lin et al. (2021)) whose sequence weights can be evaluated efficiently, and also perform competitively against autoregressive models. However, we
- show how unrestricted energy-based sequence models can suffer from **uncomputability**; and how such a problem is generally unfixable without knowledge of the true sequence distribution (§ 5, Lin and McCarthy (2022)). We then
- introduce **neural particle smoothing** (§ 6, Lin and Eisner (2018)), a family of approximate sampling methods that work with conditional latent variable models.

We then

- introduce **neural finite-state transducers** (§ 7, Lin et al. (2019)), which extend weighted finite state transducers with the introduction of mark strings, allowing scoring transduction paths in a finite state transducer with a neural network. Finally, we
- propose **neural regular expressions** (§ 8), a family of neural sequence models that are easy to engineer, allowing a user to design flexible weighted relations using Marked FSTs, and combine these weighted relations together with various operations.

Chapter 2

Background

We review popular sequence models in this chapter. We also define parametric sequence model families and weighted language classes that we will discuss in future chapters. In particular, we provide two different abstractions of sequence models. We first define and describe them as **weighted Turing machines**. This perspective allows us to derive complexity and computability results regarding different sequence model families. We then equivalently characterize them as **parametric sequence models**, which implies the power, and associated uncomputability of modern neural sequence models.

2.1 Common sequence model families

In this section we give a brief introduction to common sequence model families. Most pre-neural string transduction models are graph-based: speaking at a high level, the

transduction from input string $\mathbf{x}$ to output string $\mathbf{y}$ can be 'explained' by some graph. These graphs' structures explain how input symbols $\in \mathbf{x}$ correspond to output symbols $\in \mathbf{y}$. These models are generally realized as low-treewidth factor graph grammars (§2.1.1). In this work, we are especially interested in finite-state grammars (where symbols $\in \mathbf{x}$ are monotonically aligned with symbols $\in \mathbf{y}$ in factor graphs), namely weighted finite-state transducers. We therefore also give a focused review of this formalism (§2.1.2). As for neural string transduction models, a majority of them are based on the sequence-to-sequence transduction paradigm (§2.1.3).

2.1.1 Low-treewidth factor graph grammars

Graph-based models explicitly account for the interaction within clusters of symbols and/or discrete latent variables. In the case where both input and output strings have fixed and finite lengths, they can be described as **factor graphs** (Kschischang, Frey, and Loeliger, 2001). Recently, factor graph models have been generalized into a model family of unbounded sequence lengths: **factor graph grammars** (Chiang and Riley, 2020). Many popular sequence modeling formalisms, such as HMMs (Eddy, 1996), PCFGs (McAllester, Collins, and Pereira, 2004), and linear chain CRFs (Sutton and McCallum, 2012), which had had been popular before the resurgence of neural networks in the 2010s, have also been recognized as instantiations of factor graph grammars (Chiang and Riley, 2020, Theorem 10).

A factor graph grammar $\mathcal{G}$ is a hypergraph, which defines a set of derived graphs $\mathcal{D}(\mathcal{G})$.

A factor graph $D \in \mathcal{D}(\mathcal{G})$ is a (finite) bipartite graph, with two kinds of nodes: variables $\mathcal{V}$ and factors Ψ. Each variable $v \in \mathcal{V}$ has a finite set of possible values $\Omega(v)$. And each factor $\psi \in \Psi$ that is connected to variable nodes $v_1 \ldots v_k$ is associated with a function $G_\psi : \Omega(v_1) \times \ldots \times \Omega(v_k) \longrightarrow \mathbb{R}_{\geq 0}$. And we define a weight of D: $[D] \triangleq \sum_{\mathbf{z} \in \text{assignments of } \mathcal{V}} \prod_{\psi \in \Psi} G_\psi(\mathbf{z}_\psi)$, where $\mathbf{z}_\psi$ is the value sequence of variables that are connected to factor ψ. Finally, the weight of G is defined as the weight of all derived graphs' weights: $[\mathcal{G}] \triangleq \sum_{D \in \mathcal{D}(\mathcal{G})} [D]$.

Factor graphs can encode latent variable models: some variables $\in \mathcal{V}$ may not be observed, and factors that are functions of these variables are usually designed to reflect prior knowledge of how these latent variables influence each other, and also how they influence observed variables. G_ψ is also commonly engineered to ensure structured zeros: for example, to ensure that some variable assignment $\mathbf{z}$ is not possible, it suffices to engineer a factor ψ such that $G_\psi(\mathbf{z}_\psi) = 0$.

Factor graph grammars can model both strings, and transduction of string pairs: by defining a function f that maps every factor graph $D \in \mathcal{D}(\mathcal{G})$'s variable assignment $\mathbf{z}$ to a string $\mathbf{x}$, we can define factor graph grammar $\mathcal{G}$ which assigns positive weight to string $\mathbf{x}$ if and only if $\exists D \in \mathcal{D}(\mathcal{G}), [D] > 0$ where $\forall \mathbf{z} \in$ assignments of $\mathcal{V}, f(\mathbf{z}) = \mathbf{x}$. Likewise, by defining a function f that maps every factor graph $D \in \mathcal{D}(\mathcal{G})$'s variable assignment $\mathbf{z}$ to a string pair $(\mathbf{x}, \mathbf{y})$, we can similarly define a factor graph grammar which assigns positive weight to string pair $(\mathbf{x}, \mathbf{y})$ if and only if $\exists D \in \mathcal{D}(\mathcal{G}'), [D] > 0$ where $\forall \mathbf{z} \in$ assignments of $\mathcal{V}, f'(\mathbf{z}) = (\mathbf{x}, \mathbf{y})$.

We can thus define distributions over strings, by treating the weights of factor graph D whose variable assignments correspond to a single string $\mathbf{x}$ as the unnormalized probability

$\tilde{p}(\mathbf{x})$, and the weight of factor graph grammar $\mathcal{G}$, which generates only such factor graphs, as the partition function Z. We can define distributions over string transductions in a similar manner. Since we only know algorithms that compute both $[D]$ and $[\mathcal{G}]$ in time exponential in the graph treewidth in the worst case, compromises have to be made to limit the expressivity of the factors, to make marginalization and renormalization tractable via dynamic programming (Chandrasekaran, Srebro, and Harsha, 2008; Chiang and Riley, 2020). In sequence models, one very common compromise is the Markov assumption. For string transduction tasks, the assumption is that a factor cannot model interaction between symbols on $\mathbf{y}$ that are too far apart: a factor that is a function of y_i can typically only look at $\mathbf{y}_{i-k,i-k+1\ldots i+k}$ for some *very* small $k \in \mathbb{N}$.

Most human languages *do* exhibit complicated long term dependency (Hulst, 2010), rendering the Markov assumption unrealistic. Nonetheless, the Markov assumption allows for tractable inference and parameter estimation methods. Moreover, if a factor graph grammar $\mathcal{G}$ makes the Markov assumption, it is often easy to infer quantities other than conditional probability $p(\mathbf{y} \mid \mathbf{x})$ about $\tilde{p}$, because of the existence of efficient dynamic programming algorithms. For example, given graphical model $D \in \mathcal{D}(\mathcal{G})$, one can ask 'what is the most likely $\mathbf{y}$ given observed $\mathbf{x}$ under D, under the condition that $\mathbf{y}$ has the suffix '`d e f`'?' In contrast, seq2seq models (§2.1.3) that are trained to maximize $p(\mathbf{y} \mid \mathbf{x})$ will have to undergo expensive renormalization (*i.e.*, the partition function would be a weight sum of strings that have suffix '`d e f`'), to be able to answer such questions (i.e. compute $p(\mathbf{y} \mid \mathbf{x}, \text{`}\mathbf{y} \text{ ends with } \texttt{d e f}\text{'})$.

2.1.2 Weighted finite-state transducers

Weighted finite state transducers (WFSTs) are a commonly used subclass of factor graph grammars. They have been used for decades to analyze, align, and transduce strings in language and speech processing (Roche and Schabes, 1997; Mohri, Pereira, and Riley, 2008). They form a family of efficient, interpretable models with well-studied theory.

WFSTs are generalizations of (unweighted) finite-state transducers, which in turn generalize finite-state acceptors (FSAs):

Definition 2.1.1. *A finite-state acceptor M is an 5-tuple $M = (\Sigma, Q, q_{init}, E, F)$, where*

- *Σ is an finite alphabet,*
- *Q is a finite set of states,*
- *$q_{init} \in Q$ is the initial state,*
- *$E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is a finite set of transitions,*
- *$F \subseteq Q$ is a set of final states.*

An FSA is a directed graph. And FSA M **accepts** string $\mathbf{x}$ if there is a path from q_{init} to some state $\in F$, such that the concatenation of symbols on this path is $\mathbf{x}$.

Compared to FSAs, WFSTs:

- can transduce input strings into output strings; and

- can have different *weights* for different transductions.

Formally, weights in a WFST come from a set $\mathbb{K}$ which with a commutative operator (usually denoted as $\oplus$) and an associative operator (usually denoted as $\otimes$), forms a **semiring** (Kuich and Salomaa, 2012):

Definition 2.1.2. *A semiring K is a 5-tuple $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$, where*

- $(\mathbb{K}, \oplus, \overline{0})$ *is a commutative monoid with identity element* $\overline{0}$,
- $(\mathbb{K}, \otimes, \overline{1})$ *is a monoid with identity element* $\overline{1}$,
- $\otimes$ *distributes over* $\oplus$, *and*
- $\forall a \in \mathbb{K}, \overline{0} \otimes a = a \otimes \overline{0} = \overline{0}$.

Finally, we give a definition of WFSTs adopted from Mohri, Pereira, and Riley (2008):

Definition 2.1.3. *A weighted finite-state transducer T over a semiring $K = (\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$ is an 8-tuple $\tau = (\Sigma, \Delta, Q, q_{init}, E, F, \lambda, \rho)$, where*

- Σ *and* Δ *are input and output alphabets,*
- Q *is a finite set of states,*
- $q_{init} \in Q$ *is the initial state,*
- $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Delta \cup \{\epsilon\}) \times \mathbb{K} \times Q$ *is a finite set of transitions,*
- $F \subseteq Q$ *is a set of final states,*

- $\lambda \in \mathbb{K}, \rho : F \longrightarrow \mathbb{K}$ *are initial and final weights respectively.*

A **transition** $a = (p[a], i[a], o[a], w[a], n[a]) \in E$ is an arc from state $p[a]$ to state $n[a]$, with input and output labels $i[a]$ and $o[a]$, and weight $w[a]$. A **path** in E^* is a sequence of consecutive transitions $a_1 \dots a_n$ where $n[a_i] = p[a_{i+1}]$, $i \in [1 \dots n-1]$. An **accepting path** $\mathbf{a}$ is a path where $p[a_1] = q_{\text{init}}$ and $n[a_{|\mathbf{a}|}] \in F$. We also define the **path weight** of accepting path $\mathbf{a}$ to be $[\![\mathbf{a}]\!] \triangleq \lambda \otimes w[a_1] \otimes \cdots \otimes w[a_{|\mathbf{a}|}] \otimes \rho(n[a_{|\mathbf{a}|}])$. Finally, we define the **machine weight** of T: $[\![T]\!] \triangleq \bigoplus_{\mathbf{a}\,:\,\mathbf{a} \text{ is an accepting path of } T} [\![\mathbf{a}]\!]$.

Computing $[\![T]\!]$ exactly or approximately is feasible for production-level speech recognition systems (Mohri, 2009). However, WFSTs have the Markov property — since they lack memory. The weight of an accepting path $\mathbf{a}$: $[\![\mathbf{a}]\!]$ is simply the product of the transition weights.

2.1.3 Neural autoregressive sequence models

Factor graph grammars cannot capture arbitrarily long dependency involving arbitrarily many symbols, if we require inference to be tractable. However, neural autoregressive sequence models — the prevailing sequence-to-sequence modeling paradigm — are not subject to such a restriction.

These models — which we dub **standard autoregressive models** — are generative models: the probability of a string $\mathbf{x}$ is parametrized as $p(\mathbf{x}) = \prod_t p_\theta(x_t \mid \mathbf{x}_{<t})$. A closely related family — commonly known as **seq2seq** models (Sutskever, Vinyals, and Le, 2014) —

parametrize the probability of string $\mathbf{y}$ given $\mathbf{x}$ as $p(\mathbf{y} \mid \mathbf{x}) = \prod_t p_\theta(y_t \mid \mathbf{y}_{<t}, \mathbf{x})$.[1] Standard autoregressive models parametrize local conditional distributions $p_\theta(\cdot \mid \mathbf{x}_{<t})$ over the next symbol that follow $\mathbf{x}_{<t}$, where $\mathbf{x}_{<t}$ is a valid prefix of length $t-1$. To generate string $\mathbf{x}$, one samples from from conditional distributions $p_\theta(\cdot \mid \epsilon), p(\cdot \mid x_1), p(\cdot \mid x_1 x_2) \ldots$ iteratively, to obtain a growing sequence of output symbols $\mathbf{x}_{1\ldots t}$, until $x_{t+1} = \textsc{eos}$. Here EOS is a special symbol that indicates the transduction is done. In practice, beam search may be used in place of sampling as an effort to find more probable $\mathbf{x}$'s. The generation process for seq2seq models is similar: we sample from conditional distributions $p_\theta(\cdot \mid \epsilon, \mathbf{x}), p_\theta(\cdot \mid y_1, \mathbf{x}), p_\theta(\cdot \mid y_1 y_2, \mathbf{x}) \ldots$ iteratively, until $y_{t+1} = \textsc{eos}$.

Modern neural autoregressive sequence models typically parametrize p_θ with high-parameter-count neural networks. Furthermore, they usually parametrize local conditional distributions $p_\theta(\cdot \mid \mathbf{x}_{<t})$ (or $p_\theta(\cdot \mid \mathbf{y}_{<t}, \mathbf{x})$) in a way that these local probabilities are fast to evaluate. Therefore, not only sampling from the distributions they define is easy (using the generation method we described above), but we can also evaluate $p(\mathbf{x}) = \prod_t p_\theta(x_t \mid \mathbf{x}_{<t})$ for an arbitrary $\mathbf{x} \in \Sigma^*$; and likewise $p(\mathbf{y} \mid \mathbf{x})$. Moreover, log-likelihood gradients with regard to $\boldsymbol{\theta}$ are easy to evaluate, either: $\nabla_\theta \log p(\mathbf{x}) = \sum_t \nabla \log p_\theta(x_t \mid \mathbf{x}_{<t})$.

The ability to evaluate a string probability and its gradients with regard to $\boldsymbol{\theta}$ makes training neural autoregressive sequence models straightforward. Since the normalizing constant $Z = \sum_{\mathbf{x}} p(\mathbf{x}) = 1$ by definition for any value of the model parameters $\boldsymbol{\theta}$, we do not

[1] There are non-autoregressive seq2seq models (Gu et al., 2018; Lee, Mansimov, and Cho, 2018; Saharia et al., 2020; Ghazvininejad et al., 2020). They explore model families that are capable of *non-autoregressive* decoding that exploits parallelism. These models give up expressiveness in exchange for better speed. Therefore they still fall under our definition of *autoregressive models*, since they can be parametrized as autoregressive ones (*i.e.* decode sequentially without using parallelism).

need to compute $\nabla_\theta Z$ during MLE training, which is generally required for WFSTs (§2.1.2).

Standard autoregressive models and seq2seq models have both achieved stellar empirical results in many applications (Oord et al., 2016; Child et al., 2019; Zellers et al., 2019; Brown et al., 2020). In particular, thanks to the expressiveness of neural networks that parametrize local conditional distributions $p_\theta(\cdot \mid \mathbf{x}_{<t})$ and $p_\theta(\cdot \mid \mathbf{y}_{<t}, \mathbf{x})$, very large neural autoregressive sequence models can model human languages so well that untrained humans cannot reliably distinguish between human- and computer-generated content (Clark et al., 2021). One may attribute their success in part to the ability to capture long distance dependency — that is, unlike WFSTs, neural autoregressive sequence models in general do not have the Markov property. However this does not imply that neural autoregressive models can model all string distributions, or even approximate them 'well enough', given a reasonable amount of parameters: speaking very loosely, an autoregressive sequence model cannot 'correct' bad estimates of local conditional probability at time step t: $p_\theta(\cdot \mid \mathbf{x}_{<t})$ in subsequent time steps $t' > t$. We will make our argument rigorous in § 3. But we need to first discuss the computational aspects of weighted languages in §2.2.

2.2 Weighted languages

A sequence model that takes a string as input and outputs a non-negative number is essentially a **weighted language**. Formally, let **alphabet** V be a finite set of symbols, an (unweighted) language $\mathcal{X} \subseteq V^*$ is a countable set of strings. And a weighted language

$\tilde{p} : V^* \longrightarrow \mathbb{R}_{\geq 0}$ is a function that assigns positive scores to string $\mathbf{x}$ if and only if $\mathbf{x} \in \mathcal{X}$.

2.2.1 Normalizable weighted languages

We say a weighted language $\tilde{p}$ is **normalizable** when $Z_{\tilde{p}} \triangleq \sum_{\mathbf{x} \in V^*} \tilde{p}(\mathbf{x}) \in \mathbb{R}_{>0}$. The sum over all possible strings is finite and positive. $Z_{\tilde{p}}$ is also called the **partition function** of $\tilde{p}$.[2] We can then normalize $\tilde{p}$ into a **distribution** p over V^* such that $p(\mathbf{x}) = \tilde{p}(\mathbf{x})/Z_{\tilde{p}}$, and thereby $\sum_{\mathbf{x} \in V^*} p(\mathbf{x}) = 1$.

Let $\hat{\mathbf{x}} \preceq \mathbf{x}$ mean that $\hat{\mathbf{x}}$ is a prefix of $\mathbf{x} \in V^*$ (not necessarily a strict prefix). If $\tilde{p}$ is normalizable, then $Z(\hat{\mathbf{x}}) \triangleq \sum_{\mathbf{x} \in V^* : \hat{\mathbf{x}} \preceq \mathbf{x}} \tilde{p}(\mathbf{x})$ is $\leq Z$ for any $\hat{\mathbf{x}} \in V^*$, yielding a marginal **prefix probability** $Z(\hat{\mathbf{x}})/Z$. If the prefix $\hat{\mathbf{x}}$ has positive prefix probability, then it admits a **local conditional probability** $p(x \mid \hat{\mathbf{x}}) \triangleq Z(\hat{\mathbf{x}}\, x)/Z(\hat{\mathbf{x}})$ for each symbol $x \in V$, where the denominator is interpreted as a **local normalizing constant**. This is the conditional probability that if a random string starts with the prefix $\hat{\mathbf{x}}$, the next symbol is x. There is also a probability $p(\textsc{eos} \mid \hat{\mathbf{x}}) \triangleq 1 - \sum_{x \in V} p(x \mid \hat{\mathbf{x}}) = \tilde{p}(\hat{\mathbf{x}})/Z(\hat{\mathbf{x}}) \geq 0$ that the string ends immediately after $\hat{\mathbf{x}}$; the special symbol $\textsc{eos} \notin V$ represents "end of string."

[2] We use the convention that $Z_{\tilde{p}}$ is the partition function of $\tilde{p}$, and $Z_{\tilde{q}}$ of $\tilde{q}$, etc. The subscript is omitted in unambiguous cases.

2.2.2 Computable weighted languages

In this work, we put an emphasis on **computable weighted languages**. We say a weighted language $\tilde{p}$ is **computable**[3] if there exists a Turing machine that outputs a rational number $\tilde{p}(\mathbf{x}) \in \mathbb{Q}^+$, or $R(\mathbf{x}, \mathbf{y}) \in \mathbb{Q}^+$ on input $\mathbf{x}$ or $(\mathbf{x}, \mathbf{y})$ in finite time. A Turing machine is traditionally described as a 5-tuple $(Q, \Sigma, \delta, q_{\text{init}}, F)$ (Sipser, 2013), where Q is a set of states, Σ is an alphabet of symbols that can be read off and written onto the tape, $\delta : Q \times \Sigma \longrightarrow Q \times \Sigma \times \{\mathtt{L}, \mathtt{R}\}$ is a state transition function, q_{init} is the initial state, and F is a set of accepting states. This classical 5-tuple definition of Turing machines can be extended in many possible ways to associate a computable number with every accepted string. For example, we can make use of an additional tape to record the output number. However, in this thesis we adopt a Turing machine extension that does not change its machinery, but defines the string weight as a function of both number of computation steps, and the state sequences.

A **weighted Turing machine** W is defined as a 7-tuple

$$(Q, \Sigma, \delta, q_{\text{init}}, F, \texttt{update\_num}, \texttt{update\_denom}),$$

where the first 5 components $(Q, \Sigma, \delta, q_{\text{init}}, F)$ are defined just as in standard Turing machines. The last two additions are weight-update functions: both `update_num` and `update_denom` have signature $Q \longrightarrow \mathcal{A}$, where $\mathcal{A} \triangleq \{\textsc{same}, \textsc{carry}\}$. At the end of time step i, assuming

[3] Note that the string weights under our computable weighted languages form the set of rational numbers, not the set of *computable numbers*. That is, we cannot have string weights that are irrational but computable (*e.g.*, π). It may be possible to extend our definition of computable weighted languages, to allow general *computable* string weights. We leave the extension as future work.

the current state is q, we define

$$\textsc{num}_i = \begin{cases} \textsc{num}_{i-1} & \text{if } \texttt{update\_num}(q) = \textsc{same} \\ 2^i + \textsc{num}_{i-1} & \text{if } \texttt{update\_num}(q) = \textsc{carry} \end{cases}$$

and similarly,

$$\textsc{denom}_i = \begin{cases} \textsc{denom}_{i-1} & \text{if } \texttt{update\_denom}(q) = \textsc{same} \\ 2^i + \textsc{denom}_{i-1} & \text{if } \texttt{update\_denom}(q) = \textsc{carry}. \end{cases}$$

We (quite arbitrarily) define that $\forall q \in F \cup \{q_{\text{init}}\}$, $\texttt{update\_num}(q) = \texttt{update\_denom}(q) = \textsc{same}$. Finally, upon arriving at a halting state $\in F$ in r time steps, we say $\textsc{num}_r/\textsc{denom}_r$ is the weight of an input $\mathbf{x} \in \mathbb{B}^*$ under W, if $\textsc{num}_r/\textsc{denom}_r$ is a rational number.[4]

2.2.3 Efficiently computable (EC) weighted languages

Efficiently computable weighted languages (EC; Lin et al., 2021) are computable weighted languages, where the weight of string $\mathbf{x} \in \mathcal{X}$: $\tilde{p}(\mathbf{x})$ can be computed in $O(\text{poly}(|\mathbf{x}|))$. Most weighted languages defined by (fixed-size) neural sequence models fall into this class.

[4]We say an input string $\mathbf{x}$ has an undefined weight if $\mathbf{x}$ is not accepted by m. When $\textsc{num}_r/\textsc{denom}_r$ is not a rational number (because $\textsc{denom}_r = 0$), we also say the weight of input string $\mathbf{x}$ is undefined.

2.2.3.1 Non-uniform computation

In the machine learning approach to sequence modeling, we usually do not manually design the weighted Turing machine behind a weighted language $\tilde{p}$. Rather, we design a model M with *parameters* $\boldsymbol{\theta}$. M is a Turing machine that reads $\boldsymbol{\theta}$ and outputs the description of a specialized Turing machine $\tilde{p}_{\boldsymbol{\theta}} \triangleq M(\boldsymbol{\theta})$ that can score strings $\mathbf{x}$ and hence defines a weighted language. Without loss of generality, we will express $\boldsymbol{\theta}$ as a string in $\mathbb{B}^*$ (where $\mathbb{B} \triangleq \{0, 1\}$). For each $\boldsymbol{\theta}$, we obtain a potentially different weighted language.

Strings vary in length, and accurate modeling of longer strings may sometimes require more complex computations with more parameters. For example, when V is a natural language alphabet, a recurrent neural network may require more hidden units to model sentences of the language rather than individual words, and even more units to model whole documents. To accommodate this, we allow an *infinite sequence* of parameter vectors, $\boldsymbol{\Theta} = \{\boldsymbol{\theta}_n \in \mathbb{B}^* : n \in \mathbb{N}\}$, which yields an infinite sequence of Turing machines $\{\tilde{p}_n : n \in \mathbb{N}\}$ via $\tilde{p}_n \triangleq M(\boldsymbol{\theta}_n)$. We then define $\tilde{p}_{\boldsymbol{\Theta}}(\mathbf{x}) \triangleq \tilde{p}_{|\mathbf{x}|}(\mathbf{x})$, so a string of length n is scored by the $\tilde{p}_n$ machine. This is known as **non-uniform computation** (Arora and Barak, 2009, Chapter 6). Of course, it is legal (and common) for all of the $\boldsymbol{\theta}_n$ to be equal, or empty, but if desired, we can obtain more power by allowing the number of parameters to grow with n if needed.

We can now consider *how rapidly* the parametric and runtime complexity may grow.

- If $|\boldsymbol{\theta}_n|$ is permitted to be exponential in n, then one can fit *any* weighted language $\tilde{p}$ (even an uncomputable one). Simply use $\boldsymbol{\theta}_n$ to encode a trie with $O(|V|^{n+1})$ nodes that

maps $\mathbf{x} \mapsto \tilde{p}(\mathbf{x})$ for any $|\mathbf{x}|$ of length n, and design M such that the Turing machine $\tilde{p}_n = M(\boldsymbol{\theta}_n)$ has a (large) state transition table that mirrors the structure of this trie. The resulting collection of Turing machines $\{\tilde{p}_n \mid n \in \mathbb{N}\}$ can then compute $\tilde{p}(\mathbf{x})$ exactly for any $\mathbf{x}$, with only linear runtime $O(|\mathbf{x}|)$ (which is used to traverse the trie).

- Separately, if unbounded runtime is permitted for M, then one can exactly fit *any computable* weighted language $\tilde{p}$. Simply have M, when run on $\boldsymbol{\theta}_n$, *compute* and return the large trie-structured $\tilde{p}_n$ that was mentioned above. In this case, M need not even use the parameters $\boldsymbol{\theta}_n$, except to determine n.

- Finally, if unbounded runtime is permitted for $\tilde{p}_n$, then again one can exactly fit *any computable* weighted language $\tilde{p}$. In this case, M trivially returns $\tilde{p}_n = \tilde{p}$ for all n.

- However, if the parameters $\boldsymbol{\Theta}$ are "compact" in the sense that $|\boldsymbol{\theta}_n|$ grows only as $O(\text{poly}(n))$, and also $\tilde{p}_n = M(\boldsymbol{\theta}_n)$ is constructed by M in time $O(\text{poly}(n))$, and $\tilde{p}_n$ scores any $\mathbf{x}$ of length n in time $O(\text{poly}(n))$, then we say that the resulting weighted language $\tilde{p}$ is **efficiently computable with compact parameters** (ECCP).[5] We refer to M paired with a parameter space of possible compact values for $\boldsymbol{\Theta}$ as an **ECCP** weighted language.

ECCP weighted languages is a loose expressiveness upper bound of practical parametric sequence model families, where sequence probabilities can be evaluated reasonably fast,

[5] Since we require M to run in polytime, it can only look at a polynomial-sized portion of $\boldsymbol{\theta}_n$. Hence it is not really crucial for the parameters $\boldsymbol{\theta}_n^{\text{p}}$ to be compact, but we nonetheless include this intuitive condition, without loss of generality.

but training compute and dataset are both unbounded. The construction and execution of the neural network $\tilde{p}_n$ may perform a polynomial amount of total computation to score the string $\mathbf{x}$. This computation may involve parameters that were precomputed using any amount of effort (*e.g.*, training on data) or even obtained from an oracle (they need not be computable). However, the exponentially many strings of length n must share a polynomial-size parameter vector $\boldsymbol{\theta}_n$, which prevents the solution given in the first bullet point above.

In practice one takes $\boldsymbol{\theta}_n = \boldsymbol{\theta}$ for all n and obtains $\boldsymbol{\theta} \in \mathbb{R}^d$ by training. However, we do not consider whether such parameters are easy to estimate or even computable. We simply ask, for a given target language $\tilde{p}$, whether there *exists* a polynomially growing sequence $\boldsymbol{\Theta}$ of "good" parameter vectors for any parametric model M. When not, there can be no scheme for estimating arbitrarily long finite prefixes of such a sequence. So for any polynomial f, any training scheme that purports to return a trained model of size $f(n)$ that works "well" for strings of length $\leq n$ must fail for large enough n — even if unlimited data, computation, and oracles are allowed at training time.

2.2.4 Computable locally normalizable weighted languages (LN)

Many parameter estimation techniques and inference methods specifically work with local conditional probabilities $p(x \mid \hat{\mathbf{x}})$ (§2.2.1). Thus, weighted languages where such quantities

are computable,[6] and especially efficiently computable, are worth more discussion.

If for weighted language $\tilde{p}$, local conditional probabilities $p(x \mid \hat{\mathbf{x}})$ for all $x \in V$, and all valid prefixes $\hat{\mathbf{x}}$ can be computed by a single Turing machine in finite time, we say $\tilde{p}$ is **locally normalizable (**LN**)**. They are required to be **consistent**: the probability that a string is infinitely long under such distributions is zero. Equivalently, given any $\epsilon > 0$, we can approximate Z with a finite sum of string weights:

Proposition 2.2.1 (Consistency of LN distributions (Booth and Thompson, 1973; Chen et al., 2018b))**.** *Let $p \in$ LN be a locally normalized distribution over strings. All strings of a given length or longer have their probabilities bounded. That is, for all positive real numbers ϵ, there is a length n at which all strings $\mathbf{x}$ of length at least n have $p(\mathbf{x}) < \epsilon$.*

Most fixed-size autoregressive sequence models (§2.3.1) fall into the class of LN. In fact, many of these models compute such quantities can be computed in time $O(\text{poly}(|\hat{\mathbf{x}}|))$ (given the parameters).[7] We say that the resulting distributions are **efficiently locally normalizable**, or **ELN**.

We may again generalize ELNs to allow the use of non-uniform computation, through the use of compact parameters (in the manner of §2.2.3.1). For any weighted language

[6] We will further discuss weighted languages where these quantities are *not* computable in §3.1.

[7] An autoregressive model architecture generally defines $p(\mathbf{x})$ as an efficiently computable (§2.2.3) product of local conditional probabilities. However, the parametrization usually ensures only that $\sum_{x \in V} p_{\boldsymbol{\theta}}(x \mid \hat{\mathbf{x}}) = 1$ for all prefixes $\hat{\mathbf{x}}$. Some parameter settings may give rise to **inconsistent** distributions where $Z \triangleq \sum_{\mathbf{x} \in V^*} p_{\boldsymbol{\theta}}(\mathbf{x}) < 1$ because the generative process terminates with probability < 1 (Chen et al., 2018b). In this case, the factors $p_{\boldsymbol{\theta}}(x \mid \hat{\mathbf{x}})$ defined by the autoregressive model are not actually the conditional probabilities of the weighted language (as defined by §2.2.4). It is true that training $\boldsymbol{\theta}$ with a likelihood objective does encourage finding a weighted language whose generative process always terminates (hence $Z = 1$), since this is the behavior observed in the training corpus (Chi and Geman, 1998; Chen et al., 2018b; Welleck et al., 2020). Our definitions of ELN(CP) models require the *actual* conditional probabilities to be efficiently computable. Autoregressive models that do not sum to 1, whose normalized probabilities can be uncomputable, are not ruled out by our theorems that concern ELN(CP).

$\tilde{p}$, the Turing machine M^{q} **efficiently locally normalizes $\tilde{p}$ with compact parameters** $\boldsymbol{\Theta}^{\mathrm{q}} = \{\boldsymbol{\theta}^{\mathrm{q}}_n \mid n \in \mathbb{N}\}$ if

- the parameter size $|\boldsymbol{\theta}^{\mathrm{q}}_n|$ grows only as $O(\mathrm{poly}(n))$
- $M^{\mathrm{q}}(\boldsymbol{\theta}^{\mathrm{q}}_n)$ returns a Turing machine q_n (similar to $\tilde{p}_n$ in §2.2.3.1) in time $O(\mathrm{poly}(n))$
- $\tilde{p}$ is normalizable (so p exists)
- q_n maps $\hat{\mathbf{x}}x \mapsto p(x \mid \hat{\mathbf{x}})$ for all $x \in V \cup \{\textsc{eos}\}$ and all prefixes $\hat{\mathbf{x}} \in V^*$ with $|\hat{\mathbf{x}}| \leq n$ and $Z(\hat{\mathbf{x}}) > 0$
- q_n runs on those inputs $\hat{\mathbf{x}}x$ in time $O(\mathrm{poly}(n))$

If there is M^{q} that efficiently locally normalizes a weighted language $\tilde{p}$ with compact parameters $\boldsymbol{\Theta}^{\mathrm{q}}$, we say $\tilde{p}$ is **efficiently locally normalizable with compact parameters**, or **ELNCP**. Note that this is a property of the weighted language itself.

ELNCP weighted languages can be more powerful than ELN ones (*i.e.*, ELNCP $\supseteq$ ELN) because their parameters may be set by an oracle. In fact, some ELNCP weighted languages cannot be captured by LN weighted languages that do not have access to compact parameters (see also Figure 3.1):

Lemma 2.2.2. *The set* $\{\tilde{p} : \tilde{p} \in \mathrm{EC}, \tilde{p} \in \mathrm{ELNCP}, \tilde{p} \notin \mathrm{LN}\}$ *is not empty.*

We will prove Lemma 3.2.1 in § 3. It is also obvious that an ELNCP model is also ECCP:

Lemma 2.2.3. *An ELNCP model $\tilde{p}$ is also ECCP. Likewise, an ELN model is also EC.*

Proof. Let $\tilde{p}$ be an ELNCP language. This implies that $\tilde{p}$ is normalizable, so let $p(\mathbf{x}) \triangleq \tilde{p}(\mathbf{x}) / Z$ as usual. Specifically, let M^{q} efficiently locally normalize $\tilde{p}$ with compact parameters $\boldsymbol{\Theta}^{\mathrm{q}} = \{\boldsymbol{\theta}^{\mathrm{q}}_n \mid n \in \mathbb{N}\}$. It is simple to define a Turing machine M^{r} that maps each parameter string $\boldsymbol{\theta}^{\mathrm{q}}_n$ to a Turing machine r_n, where $r_n(\mathbf{x})$ simply computes $(\prod_{t=1}^{n} q_n(x_t \mid \mathbf{x}_{<t})) \cdot q_n(\$ \mid \mathbf{x})$. Then for all $\mathbf{x}$ of length n, $r_n(\mathbf{x}) = (\prod_{t=1}^{n} p(x_t \mid \mathbf{x}_{<t})) \cdot p(\$ \mid \mathbf{x})$, by the definition of local normalization, and thus $r_n(\mathbf{x}) = p(\mathbf{x})$.

M^{r} can be constructed by incorporating the definition of M^{q}, so that $r_n = M^{\mathrm{r}}(\boldsymbol{\theta}^{\mathrm{q}}_n)$ can include $q_n = M^{\mathrm{q}}(\boldsymbol{\theta}^{\mathrm{q}}_n)$ as a subroutine. This allows r_n to query q_n for local conditional probabilities and multiply them together.

- Since M^{q} runs in polytime, it is straightforward for this construction to ensure that M^{r} runs in polytime as well.
- Since $q_n(\cdot \mid \hat{\mathbf{x}}) \in O(\mathrm{poly}(n))$, this construction can ensure that r_n runs in polytime as well.
- We were given that $|\boldsymbol{\theta}^{\mathrm{q}}_n| \in O(\mathrm{poly}(n))$ (compact parameters).

Since p is the weighted language defined by $(M^{\mathrm{r}}, \boldsymbol{\Theta}^{\mathrm{q}})$, and M^{r} and $\boldsymbol{\Theta}^{\mathrm{q}}$ have the properties just discussed, we see that p is efficiently computable with compact parameters (ECCP). Therefore $\tilde{p}(\mathbf{x}) = Zp(\mathbf{x})$ is also ECCP.

In the case where $\tilde{p}$ is more strongly known to be ELN (the parameters $\boldsymbol{\Theta}^{\mathrm{q}}$ are not needed), a simplification of this argument shows that it is EC. □

If we define ELNCP *models* analogously to ECCP models, Lemma 2.2.3 means that locally normalized models do not provide any extra power. Their distributions can always

be captured by globally normalized models (of an appropriate architecture that we used in the proof). But we will see in § 3 that the converse is likely not true: provided that NP $\not\subseteq$ P/poly, there are efficiently computable weighted languages that cannot be efficiently locally normalized, even with the help of compact parameters. That is, they are EC (hence ECCP), yet they are not ELNCP (hence not ELN, see also Figure 3.1).

2.2.5 Complexity classes and known results

Some weighted languages we have introduced in this chapter generalize decision problem classes. In this section we describe their connection, and name some well-known complexity class results. We make use of these results to derive the relation between weighted language classes in § 3.

2.2.5.1 P, P/poly, and NP/poly

The phrase "efficiently computable with compact parameters" means that without access to those parameters, the ECCP weighted language may no longer be efficiently computable. Indeed, it need not be computable at all, if the parameter vectors store the outputs of some uncomputable function.

Our definitions above of EC and ECCP weighted languages are weighted generalizations of complexity classes P and P/poly, respectively,[8] and their supports are always unweighted languages in P and P/poly, respectively. An unweighted language L is in P iff there is a

[8] Namely the nonnegative functions in FP and FP/poly.

deterministic Turing machine that decides in $O(\text{poly}(|\mathbf{x}|))$ time whether $\mathbf{x} \in L$. And an unweighted language L' is in P/poly iff[9] there exist Turing machines $\{M_n : n \in \mathbb{N}\}$ such that M_n decides in $O(\text{poly}(n))$ time whether $\mathbf{x}$ of length n is in L', where each M_n can be constructed in $O(\text{poly}(n))$ time as $M(\boldsymbol{\theta}_n)$, for some Turing machine M and some sequence of polynomially-sized **advice strings** $\boldsymbol{\Theta} = \{\boldsymbol{\theta}_n \mid n \in \mathbb{N}\}$ with $|\boldsymbol{\theta}_n| \in O(\text{poly}(n))$. We define the language class NP/poly similarly to P/poly: the only difference is the family $\{M_n : n \in \mathbb{N}\}$ consists of *nondeterministic Turing machines*.

Naturally, P $\subseteq$ P/poly. But P/poly is larger than P: it contains all sparse languages, regardless of their hardness — even sparse undecidable languages — as well as many dense languages. The extra power of P/poly comes from its access to compact advice strings that do not have to be recursively enumerable, let alone efficient to find. This corresponds to statistical modeling, where the trained model has a computationally efficient architecture plus access to parameters that might have taken a long time to find.

2.2.5.2 NP-completeness and Sat

NP-complete decision problems have solutions that are efficient to validate but inefficient to find (assuming P $\neq$ NP). One of the most well-known NP-complete problems is the boolean

[9] Our presentation of P/poly is a variant of Arora and Barak (2009, §6), in which inputs $\mathbf{x}$ of length n are evaluated by a polytime function M that is given an advice string $\boldsymbol{\theta}_n$ as an auxiliary argument. This corresponds to a neural architecture M that can consult trained parameters $\boldsymbol{\theta}_n$ at runtime. We have replaced the standard call $M(\boldsymbol{\theta}_n, \mathbf{x})$ with the "curried" expression $M(\boldsymbol{\theta}_n)(\mathbf{x})$, which we still require to execute in polynomial total time. Here the intermediate result $M_n = M(\boldsymbol{\theta}_n)$ corresponds to a trained runtime model for inputs of length n. Our Turing machines M_n have size polynomial in n (because they are constructed by M in polynomial time). They correspond to the polynomial-sized boolean circuits M_n that are used to evaluate inputs of length n under the classical definition of P/poly (Ladner, 1975). We exposed these intermediate results M_n only to observe in §2.2.3.1 and §3.4.3 that if we had allowed the M_n to grow exponentially, they would have been able to encode the answers in tries.

satisfiability problem (SAT) (Cook, 1971). Given a boolean formula ϕ, SAT accepts ϕ iff ϕ can be satisfied by some value assignment. For example, the formula $(A_1 \vee \neg A_2 \vee A_3) \wedge (A_1 \vee \neg A_4)$ is in SAT, since there is a satisfying assignment $A_{1\ldots4} = 1101$. We denote the number of satisfying assignments to ϕ as $\#(\phi)$.

It is widely believed that no NP-complete languages are in P/poly. Otherwise we would have all of NP $\subseteq$ P/poly and the polynomial hierarchy would collapse at the second level (Karp and Lipton, 1980).

A capacity limitation of EC/ECCP weighted languages naturally follows from this belief:

Lemma 2.2.4. *For any $L \in$ P, there exists an EC weighted language with support L. For any $L \in$ P/poly, there exists an ECCP language with support L. But for any $L \in$ NP-complete, there exists no ECCP language with support L (assuming NP $\not\subseteq$ P/poly).*

This simple lemma relates our classes EC and ECCP of *weighted* languages to the complexity classes P and P/poly of their supports, which are *unweighted* formal languages (§ 2). It holds because computing a string's weight can be made as easy as determining whether that weight is nonzero (if we set the weights in a simple way), but is certainly no easier. We spell out the trivial proof to help the reader gain familiarity with the formalism.

Proof. Given L, define a weighted language $\tilde{p}$ with support L by $\tilde{p}(\mathbf{x}) = 1$ if $\mathbf{x} \in L$ and $\tilde{p}(\mathbf{x}) = 0$ otherwise.

If $L \in P$, then clearly $\tilde{p}$ is EC since the return value of 1 or 0 can be determined in polytime.

If $L \in$ P/poly, L can be described as a tuple $(M, \boldsymbol{\Theta})$ following our characterization in §2.2.5.1. It is easy to show that $\tilde{p}$ is ECCP, using the same polynomially-sized advice strings $\boldsymbol{\Theta}$. We simply construct $M^{\tilde{\mathrm{P}}}$ such that $M^{\tilde{\mathrm{P}}}(\boldsymbol{\theta}_n)$ returns 1 or 0 on input $\mathbf{x}$ according to whether $M(\boldsymbol{\theta}_n)$ accepts or rejects $\mathbf{x}$. Both $M^{\tilde{\mathrm{P}}}(\boldsymbol{\theta}_n)$ and $M^{\tilde{\mathrm{P}}}(\boldsymbol{\theta}_n)(\mathbf{x})$ are computed in time $O(\text{poly}(n))$ if $|\mathbf{x}| = n$. (The technical construction is that $M^{\tilde{\mathrm{P}}}$ simulates the operation of M on the input $\boldsymbol{\theta}_n$ to obtain the description of the Turing machine $M_n = M(\boldsymbol{\theta}_n)$, and then outputs a slightly modified version of this description that will write 1 or 0 on an output tape.)

For the second half of the lemma, we prove the contrapositive. Suppose $\tilde{p}$ is an ECCP weighted language with support L. $\tilde{p}$ can be characterized by a tuple $(M^{\tilde{\mathrm{P}}}, \boldsymbol{\Theta})$. It is easy to show that $L \in$ P/poly, using the same polynomially-sized advice strings $\boldsymbol{\Theta}$. We simply construct M such that $M(\boldsymbol{\theta}_n)$ accepts $\mathbf{x}$ iff $M^{\tilde{\mathrm{P}}}(\boldsymbol{\theta}_n)(\mathbf{x}) > 0$. Then by the assumption, $L \notin$ NP-complete. □

2.3 Parametric sequence models

We characterize computable weighted languages as automata in §2.2. Such a description makes it easy to discuss computability and complexity aspects of different weighted language families. However, most popular neural sequence models do not directly implement these automata: for example, RNNs and Transformers do not seem to have an intrinsic notion of halting states — RNNs and Transformers can map any rational-valued embedding sequences into a family of arbitrary length rational-valued embedding sequences. The lack of explicit

transition among finitely many states makes their correspondence with automaton-based abstractions (*i.e.*, weighted Turing machines which we introduced in §2.2.2) somewhat obscure.

2.3.1 Formal definition

To formally characterize the computational power of neural sequence models, we adopt the definition of **parametric sequence model families** from Pérez, Barceló, and Marinkovic (2021). We define a parametric sequence model family to be a set of **seq-to-seq** models $\mathbf{N} = \{N_{\boldsymbol{\theta}} : \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$, where $\boldsymbol{\Theta} \in \bigcup_{k \in \mathbb{N}} \mathbb{Q}^k$. We also define **dimension size** $d_{\boldsymbol{\theta}} \in \mathbb{N}$, **embedding function** $f_{\boldsymbol{\theta}} : \mathbb{B} \longrightarrow \mathbb{Q}^{d(\boldsymbol{\theta})}$, and **initial states** $\mathbf{s}_{\boldsymbol{\theta}} \in \mathbb{Q}^{d(\boldsymbol{\theta})}$ to be properties of $N_{\boldsymbol{\theta}}$.

Seq-to-seq model $N_{\boldsymbol{\theta}} : (\mathbb{Q}^d)^* \longrightarrow (\mathbb{Q}^d)^{\mathbb{N}}, N_{\boldsymbol{\theta}} \in \mathbf{N}$ maps an input embedding sequence $\{\mathbf{e}_t : \mathbf{e}_t \in \mathbb{Q}^{d(\boldsymbol{\theta})}, t \in [1 \dots T]\}, T \in \mathbb{N}$ to an infinite sequence $\{\mathbf{y}_k \in \mathbb{Q}^{d(\boldsymbol{\theta})} : k \in \mathbb{N}\}$. We define $N_{\boldsymbol{\theta}}$ to **accept** $\mathbf{x} = [x_1 \dots x_T] \in \mathbb{B}^*$ if and only if there exists $r \in \mathbb{N}$, such that $N_{\boldsymbol{\theta}}$ maps embedding sequence $[f_{\boldsymbol{\theta}}(x_1) \dots f_{\boldsymbol{\theta}}(x_T)]$ to an infinite sequence $\{\mathbf{y}_k \in \mathbb{Q}^{d(\boldsymbol{\theta})} : k \in \mathbb{N}\}$, where $g(\mathbf{y}_r) = 1$. And we say $N_{\boldsymbol{\theta}}$ **recognizes** language $L_{\boldsymbol{\theta}}$ if and only if there exists a polytime termination decision function $g_{\boldsymbol{\theta}} : \mathbb{Q}^{d(\boldsymbol{\theta})} \longrightarrow \mathbb{B}$ such that $N_{\boldsymbol{\theta}}$ *accepts* every string $\mathbf{x} \in L_{\boldsymbol{\theta}}$. Furthermore, we say $\mathbf{N}$ is **Turing-complete** if for every Turing machine M, there exists $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ such that $L_{\boldsymbol{\theta}}$ is the language of M. We further say $\mathbf{N}$ is **provably Turing-complete** if there exists an algorithm that takes Turing machine M as input, and outputs $\boldsymbol{\theta}$ such that $L_{\boldsymbol{\theta}}$ is the language of M.

Both RNNs and Transformers are provably Turing-complete (Siegelmann and Sontag, 1992; Pérez, Barceló, and Marinkovic, 2021).[10] Notably, both can simulate computations in a universal Turing machine, with finite-length parameter vectors.[11] Moreover, since there are Turing machines whose halting property cannot be proven, it follows that there are RNNs and Transformers whose behaviors cannot be predicted by any algorithms.

2.3.2 EC-complete parametric families

This thesis discusses *weighted* languages. Just as we extended the definition of Turing machines to weighted Turing machines in §2.2.2, here we also extend the definition of sequence model families to recognize weighted languages. We therefore define EC-**complete parametric families**, as a parametric sequence model counterpart of the weighted language class EC (which was defined in terms of weighted Turing machines).

Formally, a parametric family $\{N_{\boldsymbol{\theta}} : \boldsymbol{\theta} \in \boldsymbol{\Theta} \subseteq \mathbb{R}^d, d \in \mathbb{N}\}$ is EC-complete if given any $\tilde{p} \in$ EC (as a description of a weighted Turing machine), we can construct a parameter vector $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ such that there exist polytime functions $w_p : \mathbb{Q}^d \to \mathbb{N} \cup \{0\}$ and $w_q : \mathbb{Q}^d \to \mathbb{N} \cup \{0\}$, where whenever on input $\mathbf{x} = [x_1 \dots x_T]$, if $g(\mathbf{y}_r) = 1$ for some output embeddings $\mathbf{y}_r$ (that is, $N_{\boldsymbol{\theta}}$ accepts $\mathbf{x}$ in r time steps), ${}^{w_p(\mathbf{y}_r)}/_{w_q(\mathbf{y}_r)} = \tilde{p}(\mathbf{x})$.

Pérez, Barceló, and Marinkovic (2021) formally showed the Turing completeness of

[10] When some conditions are met: for example arbitrary precision rational weights. However such conditions usually do not hold for real-life machine learning models.

[11] Specifically, let L be the language of any universal Turing machine, and let $\boldsymbol{\Theta}_{\text{TRANSFORMER}}$ and $\boldsymbol{\Theta}_{\text{RNN}}$ be parameter families of Transformer and RNN models. There exist finitely long parameter vectors $\boldsymbol{\theta}_{\text{RNN}} \in \boldsymbol{\Theta}_{\text{RNN}}$ and $\boldsymbol{\theta}_{\text{TRANSFORMER}} \in \boldsymbol{\Theta}_{\text{TRANSFORMER}}$, where $L_{\boldsymbol{\theta}_{\text{RNN}}} = L_{\boldsymbol{\theta}_{\text{TRANSFORMER}}} = L$.

Transformers:

Theorem 2.3.1. *(Pérez, Barceló, and Marinkovic, 2021, Theorem 6) The class of Transformer networks with positional encodings is Turing complete. Moreover, Turing completeness holds even in the restricted setting in which the only non-constant values in positional embedding pos(n) of n, for $n \in \mathbb{N}$, are n, $1/n$, and $1/n^2$, and Transformer networks have a single encoder layer and three decoder layers.*

We show that with a slight modification to positional embeddings, we can extend Theorem 2.3.1 to show that the family of arbitrary precision, hard attention Transformers defined in Pérez, Barceló, and Marinkovic (2021, Section 3) is EC-complete:

Theorem 2.3.2. *The class of one-encoder-layer four-decoder-layer Transformer networks with positional encodings $(n, 1/n, 1/n^2, 2^n)$ is* EC*-complete.*

Proof. To model all weighted Turing machines defined in §2.2.2, we extend the Turing-complete one-encoder-layer three-decoder-layer Transformer network construction introduced by Pérez, Barceló, and Marinkovic (2021) with one additional layer. We also modify the original positional embedding function $\text{pos} : \mathbb{N} \longrightarrow \mathbb{Q}^d$ to include a new component. In this proof, we let

$$\text{pos}(i) = [0, \ldots, 0, 1, i, 1/i, 1/i^2, 2^i]$$

instead of $\text{pos}(i) = [0, \ldots, 0, 1, i, 1/i, 1/i^2]$ as in (Pérez, Barceló, and Marinkovic, 2021).

For the sake of clarity, our construction is a 'stack-on' construction: the encoder layer and the first 3 decoder layers are largely identical to the design of Pérez, Barceló, and Marinkovic (2021), with the only difference being necessary changes to accommodate our one additional positional embeddings component.[12] It may be possible to strengthen our results by showing that one-encoder-layer **three**-decoder-layer Transformer networks with the original positional embeddings — the parametrization family (Pérez, Barceló, and Marinkovic, 2021) showed to be Turing-complete — are EC-complete as well, with a more involved construction. We leave such an improvement as future work.

We claim our fourth layer of the decoder has output $\mathbf{y}_r = \mathbf{y}'_r + \mathbf{w}_r$, where

- $\mathbf{y}'_r$ is a zero-padded version of the original Transformer output embeddings from (Pérez, Barceló, and Marinkovic, 2021). $\mathbf{y}'_r$ is of the form

$$[[\![q^r]\!], [\![s^r]\!], m^{(r-1)}, 0, \dots, 0]$$

  where $[\![q^r]\!]$ denotes an one-hot vector of size $|Q|$ where the q^r-th component is 1 (again following the notation of (Pérez, Barceló, and Marinkovic, 2021)).

- And $\mathbf{w}_r$ is of the form

$$[\mathbf{0}_q, \mathbf{0}_s, 0, \text{NUM}_r, \text{DENOM}_r, 0, \dots, 0] \tag{2.1}$$

[12] Since we increase the output embeddings' dimension by 1, we also need to pad all matrices in the original construction by additional zero columns/rows, such that our new positional embeddings' new component has no effect on any computation in the encoder layer, and the first 3 decoder layers.

where $\text{num}_r \in \mathbb{N} \cup \{0\}$ and $\textsc{denom}_r \in \mathbb{N} \cup \{0\}$ are defined in §2.2.2.

Now we describe how $\mathbf{w}_r$ can be computed from $[\mathbf{y}'_0 \dots \mathbf{y}'_r]$ using the attention mechanism, with the help of our new positional embeddings. Specifically, we want to show that we can construct feedforward networks Q_4, K_4, and V_4 such that

$$\mathbf{y}_i = \text{Att}\big(Q_4(\mathbf{y}''_i), K_4(\mathbf{Y}''_i), V_4(\mathbf{Y}''_i)\big)$$

where $\mathbf{y}''_i = \mathbf{y}'_i + \text{pos}(i)$, $\mathbf{Y}' = [\mathbf{y}'_1 \dots \mathbf{y}'_i]$, and $\mathbf{Y}''_i = \mathbf{Y}' + [\text{pos}(1), \dots, \text{pos}(i)]$. We let $Q_4(\mathbf{y}'') = [0, \dots, 0, 1, 0, 0, 0, 0]$ be a constant vector, and $K_4(\mathbf{Y}''_i) = \mathbf{Y}''_i$. Finally, we let

$$\begin{aligned} V_4(\mathbf{Y}''_i) = [&\mathbf{0}_q, \mathbf{0}_s, 0, \\ &\mathbb{I}(\texttt{update\_num}(q^i) = \textsc{carry})2^i, \\ &\mathbb{I}(\texttt{update\_denom}(q^i) = \textsc{carry})2^i, 0, \dots, 0]. \end{aligned}$$

Q_4 is a constant function (such that it always attends to the unity component of $\mathbf{Y}''_i$), so it can be implemented as a single-layer feedforward network. K_4 is the identity function, which can also be implemented as a single-layer feedforward network. V_4 on the hand can be implemented as a fixed-size feedforward network, with the piecewise-linear sigmoidal function σ: in the case of $\textsc{denom}_i$, the network would first project q^i to $\mathbf{a} = [\mathbb{I}(\texttt{update\_denom}(q^i) = \textsc{carry}), \mathbb{I}(\texttt{update\_denom}(q^i) = \textsc{same})]$ (using the one-hot $[\![q^i]\!]$ segment from $\mathbf{y}'_i$), multiply it by $\mathbf{b} = [2^i, 0]$ (with the help of nonlinearity from σ), and put $\mathbf{a} \otimes \mathbf{b}$ at the position of $\textsc{denom}_i$ in equation (2.1). The component at $\textsc{num}_i$ in equation (2.1)

can be computed likewise.

Given any position $r \in \mathbb{N}$, we have

$$\begin{aligned}
\mathbf{y}_r = \mathrm{Att}(Q_4(\mathbf{y}''_r), K_4(\mathbf{Y}''_r), V_4(\mathbf{Y}''_r)) = [&\mathbf{0}_q, \\
&\mathbf{0}_s, \\
&0, \\
&\frac{1}{r}\sum_{i=1}^{r} \mathbb{I}(\texttt{update_num}(q^i) = \textsc{carry})2^i, \\
&\frac{1}{r}\sum_{i=1}^{r} \mathbb{I}(\texttt{update_denom}(q^i) = \textsc{carry})2^i, \\
&0, \\
&\dots, \\
&0].
\end{aligned}$$

Let $\texttt{extract\_avg\_num}(\mathbf{y}_r)$ be an affine transformation that extracts the $\big(|Q| + |\Sigma| + 2\big)^{\text{nd}}$ component from $\mathbf{y}_r$, and $\texttt{extract\_avg\_denom}(\mathbf{y}_r)$ be an affine transformation that extracts the $\big(|Q| + |\Sigma| + 3\big)^{\text{rd}}$ component from $\mathbf{y}_r$, we have

$$\begin{aligned}
\frac{\texttt{extract_avg_num}(\mathbf{y}_r)}{\texttt{extract_avg_denom}(\mathbf{y}_r)} &= \frac{\sum_{i=1}^{r} \mathbb{I}(\texttt{update_num}(q^i) = \textsc{carry})2^i}{\sum_{i=1}^{r} \mathbb{I}(\texttt{update_denom}(q^i) = \textsc{carry})2^i} \\
&= \frac{\textsc{num}_r}{\textsc{denom}_r}
\end{aligned}$$

which is the weight of input $\mathbf{x} = [x_1 \dots x_T]$ as we defined in §2.3.1. □

2.4 Realistic parametric sequence models

In §2.3 we discussed hard-attention Transformer models that can keep decoding symbols after they consumed every input symbol. This assumption does not hold for common Transformer language models, such as GPT-3 (Brown et al., 2020).

It turns out the constraint that no additional symbols can be decoded severely limits expressiveness. For example, Schwartz, Thomson, and Smith (2018) showed that a particular CNN model family is equivalent to weighted finite-state automata. Merrill et al. (2020) also showed saturated GRUs and RNNs can also be described by finite-state automata. More recently, Hao, Angluin, and Frank (2022) showed that hard-attention Transformers that do not decode additional symbols can only recognize languages within the circuit complexity class AC^0, which does not contain 'easy' languages such as Parity.

2.5 Brief summary

In this chapter, we have introduced several abstractions of sequence models. In particular, we have identified several classes of weighted languages, namely EC, LN, ELN, ECCP, and ELNCP.

In the next chapter, we will formally establish an expressiveness taxonomy of these weighted language classes.

Chapter 3

An expressiveness taxonomy of various weighted language classes

We have defined various weighted language families in §2.2. In particular, we have introduced EC and ELN as classes of weighted languages defined by common neural sequence model architectures §2.3. We have also introduced ECCP and ELNCP as non-uniform counterparts of EC and ELN. These weighted language classes can be seen as abstractions of neural sequence models that do arbitrary polytime computation, and are very powerful. Still, we ask: are there expressiveness differences among these classes? More specifically, in this chapter we answer the following questions:

Does local normalization hurt expressiveness? (§3.1) LN weighted languages introduced in §2.2.4 are an abstraction of autoregressive sequence models (§2.1.3). LN weighted languages are guaranteed to have computable local conditional distributions

given any possible prefix, which in turn makes inference and parameter estimation easier.

However, we do find that the parametrization assumption of LN weighted languages does reduce expressiveness (Theorem 3.1.1) — they cannot even model some normalizable weighted languages $\in$ EC — namely there is a string distribution where the unnormalized string probabilities are efficient to compute, but the string probabilities cannot be expressed as products of left-to-right autoregressive factors.

Just train larger models? (§3.2) The negative results stated by Theorem 3.1.1 might not deter machine learning practitioners from building bigger and bigger neural autoregressive sequence models. 'Why should we care when you said fixed-size autoregressive sequence models will not work for all strings? We will just build a bigger model that works on longer strings' they may say. While we can show that we do capture more distributions if we allow the parameter vector size to grow with the maximum length of strings we model (Lemma 3.2.1), we also show that making models larger is no panacea: we will exhibit a weighted language $\in$ EC that cannot be modeled by larger autoregressive models, unless the autoregressive models grow *superpolynomially* in string length (Theorem 3.2.2). Moreover, autoregressive models not only cannot fit these distributions exactly, they cannot even capture their supports (Theorem 3.2.3) or approximate distributions that honor rankings of string weights (Lemma 3.2.4). Finally, we show that we not only cannot keep the string weight rankings — we cannot even guarantee to approximate string weights from some

EC language within any multiplicative factor — if the parameter vector only grows polynomially in string length, and that we use only polytime in computing local conditional distributions, even with the help of randomized algorithms (Lemma 3.2.7 and Theorem 3.2.6).

Do latent variables help? (§3.3) Despite the negative results with autoregressive sequence models, they become much more powerful when they also model latent variables. Specifically, we show that the class ELNCP, which is an abstraction of parametrized autoregressive sequence models, can capture supports of all weighted languages ∈ ECCP, which is an abstraction of parametrized energy-based sequence models, as long as some symbols are latent variables (Theorems 3.3.1 and 3.3.2).

Model family	Compact parameters?	Efficient scoring?	Normalization possible?	Support can be …
ELN/ELNCP: Autoregressive models (§2.2.4)	✓	✓	✓	*some but not all* $L \in$ P
EC/ECCP: Energy-based models (§2.2.3)	✓	✓	✗	*all* $L \in$ P but *no* $L \in$ NPC
Lightly marginalized ELNCP: Latent-variable autoregressive models (§3.3)	✓	✗	✓	*all* $L \in$ NP
Lookup models (§3.4.3)	✗	✓	✓	*anything*

Table 3.1: A feature matrix of parametric model families discussed in this chapter.

The space of unweighted languages. Based on our answers to these questions, we identify three sequence model families that all more expressive than ELN (§3.4; also Table 3.1). We characterize these sequence families and connect them to existing models in the literature. Major findings of this chapter (in terms of the various supports of different weighted language classes) are summarized in Figure 3.1. We assume in this diagram that

NP $\not\subseteq$ P/poly. Each rectangular outline corresponds to a complexity class (named in its lower right corner) and encloses the languages whose decision problems fall into that class. Each ***bold-italic label*** (colored to match its shape outline) names a model family and encloses the languages that can be expressed as the *support* of some *weighted* language in that family. All induced partitions in the figure are non-empty sets: shape A properly encloses shape B if and only if language class A is a strict superset of language class B. As mentioned in Table 3.1,[1] standard autoregressive models (ELN models) have support languages that form a strict subset of P (Lemmas 2.2.3 and 2.2.4, Theorem 3.1.1, and §2.2.5.1). ELNCP models (§2.2.4) extend ELN models by allowing the parameter size to grow polynomially in string length, allowing them to capture both more languages inside P (Lemma 3.2.1) and languages outside P (including undecidable but sparse languages) that can be characterized autoregressively with the help of these compact parameters. All of those languages belong in the class P/poly. Theorem 3.2.3 establishes that energy-based (EC) and ECCP models go strictly further than ELN and ELNCP models, respectively (Theorem 3.2.3): they correspond to the *entire* classes P and P/poly (Lemma 2.2.4). However, even ECCP does not capture any NP-complete languages under our assumption NP $\not\subseteq$ P/poly. Allowing a polynomial number of latent symbols extends the power further still: lightly marginalized ELNCP or ECCP distributions cover exactly the languages $\in$ NP/poly (Theorem 3.3.2). Finally, if we were to drop the requirement that the parameters $\boldsymbol{\Theta}$ must be compact, we could store lookup tries to model any weighted language (§3.4.3).

[1] Also see Figure 3.1.

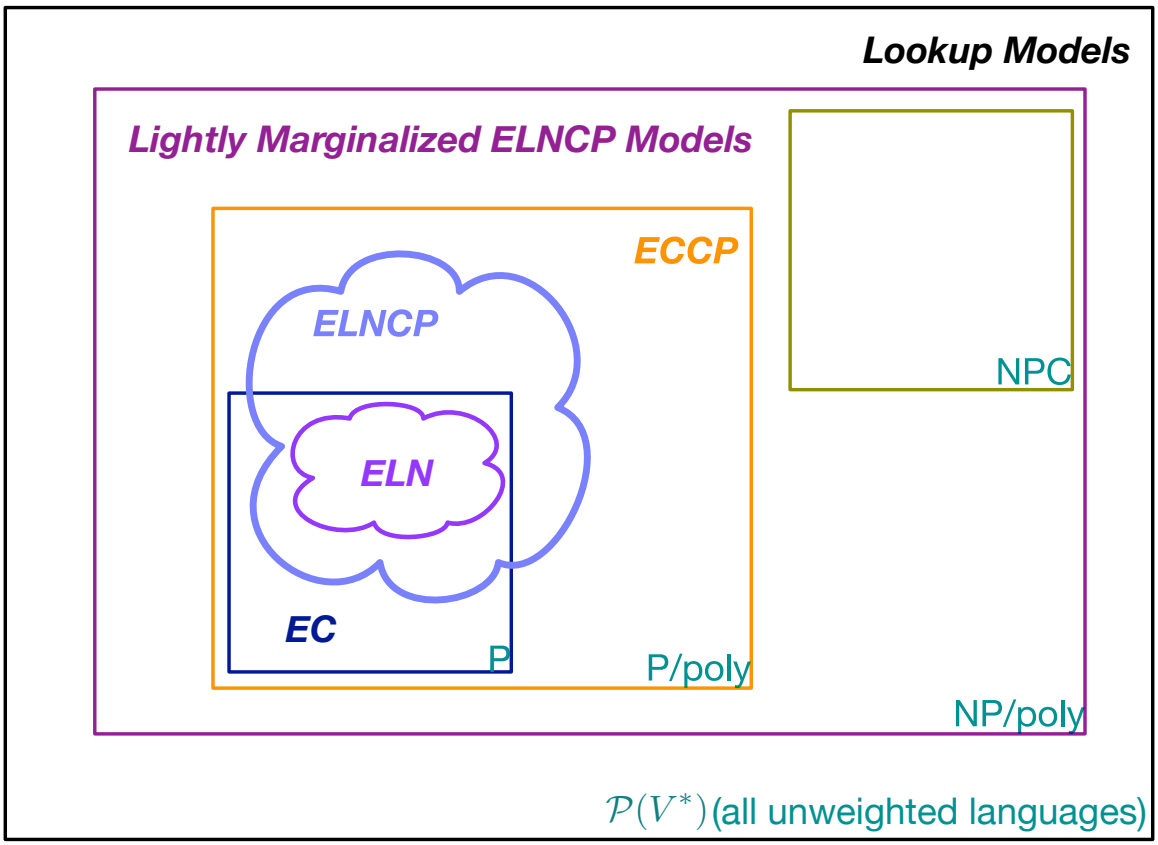


Figure 3.1: The space of unweighted languages.

3.1 Effects of local normalization

LN weighted languages require each autoregressive factor $p(\cdot \mid \mathbf{x}_{<t})$ to be computed in finite time. Therefore it may not be surprising that they cannot model weighted languages where such autoregressive factors are not computable. However, we can construct one such weighted language that is still $\in$ EC:

Theorem 3.1.1. *The set* $\{\tilde{p} : \tilde{p}$ *is normalizable,* $\tilde{p} \in \text{EC}, \tilde{p} \notin \text{LN}\}$ *is not empty.*

Proof. Given any unweighted language $L \subseteq \mathbb{B}^*$, we can define a normalizable weighted

language $\tilde{p}$ with support L by $\tilde{p}(\mathbf{x}) = 1/3^{|\mathbf{x}|+1}$ for $\mathbf{x} \in L$ and $\tilde{p}(\mathbf{x}) = 0$ otherwise. Moreover, if $L \in \mathrm{P}$, then $\tilde{p} \in \mathrm{EC}$.

For our purposes, we take L to consist of all strings of the form $\mathbf{x}^{(1)}\mathbf{x}^{(2)}$, for which there exists a deterministic Turing machine M such that $\mathbf{x}^{(1)} = \mathrm{enc}(M)$ (where enc is a prefix-free encoding function) and $\mathbf{x}^{(2)}$ encodes an accepting execution path of M on an empty input. (Such a path may be represented as a sequence of transitions of M that begins with an initial state and ends at an accepting state.) Note that any deterministic TM $\mathbf{x}^{(1)}$ can be paired with at most one accepting execution path $\mathbf{x}^{(2)}$, and cannot be paired with any $\mathbf{x}^{(2)}$ if it does not halt.

Clearly $L \in \mathrm{P}$: given $\mathbf{x} \in \mathbb{B}^*$, we can decide whether $\mathbf{x} \in L$ by first checking if $\mathbf{x}$ can be expressed as a concatenation of strings $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ of the required form. Then we build M from $\mathbf{x}^{(1)}$ and simulate it to check the transitions in $\mathbf{x}^{(2)}$ on M step-by-step. This can be done in $O(\mathrm{poly}(|\mathbf{x}|))$ total time. We conclude that the $\tilde{p}$ derived from L is EC.

Now, $Z(\mathbf{x}^{(1)}) > 0$ iff M halts on the empty input. But this undecidable problem could be decided if there were an LN weighted language that had support L, since then $Z(\mathbf{x}^{(1)}) / Z$ could be found as a product of local conditional probabilities, $\prod_{t=1}^{|\mathbf{x}^{(1)}|} p(x_t^{(1)} \mid \mathbf{x}_{<t}^{(1)})$, that could each be computed by a Turing machine. Therefore $\tilde{p}$ is not LN. □

3.2 Effects of non-uniform computation

Indeed, as we noted in §2.2.3.1, machine learning models of sequences arguably benefit from the power of non-uniform models of computation. Below is a summary of this section.

Non-uniform computation *does* give sequence models additional power (§3.2.1). Nonetheless, the limitations introduced by local normalization *cannot* be entirely mitigated by the additional power coming from non-uniform computation, if we require the model sizes to be reasonably small, and that inference from these models is reasonably fast, under the assumption of NP $\not\subseteq$ P/poly. More specifically, we will exhibit a weighted language that cannot be captured by ELNCP languages, regardless of:

- the amount of training data,
- the amount of compute used to train a sequence model,
- and the actual parametrization of the sequence model,

as long as it has a parameter vector size polynomial in the length of the strings being modeled, and computes local conditional probabilities in time polynomial of string lengths (*i.e.*, the sequence model $\in$ ELNCP).

3.2.1 ELNCP models are strictly more powerful than ELN models

ELNCP weighted languages do have more power than ELN weighted languages. In fact, the compact parameter vectors allow ELNCP weighted languages to capture distributions that no LN weighted languages can:

Lemma 3.2.1. *The set* $\{\tilde{p} : \tilde{p} \in \text{EC}, \tilde{p} \in \text{ELNCP}, \tilde{p} \notin \text{LN}\}$ *is not empty.*

Lemma 3.2.1 justifies why this region is drawn as non-empty in Figure 3.1. Note that Theorem 3.1.1 can be regarded as a corollary of Lemma 3.2.1.

Proof. The weighted language $\tilde{p}$ constructed in Theorem 3.1.1 is not necessarily ELNCP. To fix this, we modify the construction to obtain a weighted language $\tilde{p}'$ with *sparse* support L'. We will again be able to show that $\tilde{p}'$ is EC and not ELN. To show that $\tilde{p}'$ is also ELNCP, we will rely on the sparsity of L', meaning that $\text{prefixes}(L') \triangleq \{\hat{\mathbf{x}}' : (\exists \mathbf{x}' \in L')\ \hat{\mathbf{x}}' \preceq \mathbf{x}'\}$ contains at most $O(\text{poly}(n))$ strings $\hat{\mathbf{x}}'$ of length $\leq n+1$. Thus, we can use Θ_n^{q} to store all of those strings $\hat{\mathbf{x}}'$ in polynomial space, along with their $Z(\hat{\mathbf{x}}')$ values.[2] Notice that all strings $\hat{\mathbf{x}}' \notin \text{prefixes}(L')$ have $Z(\hat{\mathbf{x}}') = 0$, so they need not be stored. Now for any $\hat{\mathbf{x}}'$ of length $\leq n$, a Turing machine that consults $\boldsymbol{\theta}_n^{\text{q}}$ can compute $q(x \mid \hat{\mathbf{x}}') = Z_{\tilde{p}'}(\hat{\mathbf{x}}' x) \,/\, Z_{\tilde{p}'}(\hat{\mathbf{x}}')$ in time $O(\text{poly}(n))$ as desired, establishing that $\tilde{p}'$ is ELNCP.

[2] More precisely, the first b bits of $Z(\hat{\mathbf{x}}') \leq 1$ may be stored in Θ_{n+b}^{q}, when ELNCP is defined as explained in our "Remark on irrationality" below.

We may define $\tilde{p}'$ as follows. Let sparsify($\mathbf{x}$) be a version of $\mathbf{x}$ with many extra 0 symbols inserted: specifically, it inserts 2^t copies of 0 immediately before the t^{th} bit of $\mathbf{x}$, for all $1 \leq t \leq |\mathbf{x}|$. We construct $\tilde{p}'$ so that $\tilde{p}'(\text{sparsify}(\mathbf{x})) = \tilde{p}(\mathbf{x})$. Specifically, let $L' \triangleq \text{sparsify}(L)$. The inverse function $\text{sparsify}^{-1}(\mathbf{x}')$ is defined on exactly $\mathbf{x}' \in L'$, and is unique when defined. For all $\mathbf{x}' \in \mathbb{B}^*$, let $\tilde{p}'(\mathbf{x}') \triangleq \tilde{p}(\text{sparsify}^{-1}(\mathbf{x}'))$ if $\text{sparsify}^{-1}(\mathbf{x}')$ is defined, and $\tilde{p}'(\mathbf{x}') \triangleq 0$ otherwise. This can be computed in polytime, so $\tilde{p}'$ is EC. Also, its support L' is sparse as claimed, so $\tilde{p}'$ is ELNCP.

Finally, we claim $\tilde{p}'$ is not LN. A given deterministic Turing machine M halts on the empty input iff $\text{enc}(M) \in \text{prefixes}(L)$ iff $\text{sparsify}(\text{enc}(M)) \in \text{prefixes}(L')$ iff $Z'(\text{sparsify}(\text{enc}(M))) > 0$. But as in the proof of Theorem 3.1.1, this would be decidable if $\tilde{p}'$ were LN as defined in §2.2.4, since then we would have a Turing machine to compute the local conditional probabilities $p'(\hat{x}_t \mid \hat{\mathbf{x}}_{<t})$ for $\hat{\mathbf{x}} = \text{sparsify}(\text{enc}(M))$. □

3.2.2 ELNCP models cannot exactly capture all EC (or ECCP) distributions

Lemma 3.2.1 states that with the help of compact parameters, ELNCP models can have some languages ∈ P as their supports which cannot be supports of LN models (which contains all ELN languages). However, our proof of Lemma 3.2.1 depends on the observation that compact parameter vectors can encode any sparse language. Yet it is known that sparse NP-complete languages exist if and only if P = NP (Mahaney, 1982), which implies that

assuming P $\neq$ NP, we can no longer use the 'sparsification' technique in the proof of Lemma 3.2.1 to model certificates of any NP-complete language with ELNCP distributions, since none of them will be sparse. However, all such certificates are verifiable in polynomial time by definition (and hence model-able by EC distributions). Indeed, if we further assume NP $\not\subseteq$ P/poly (§2.2.5.2) — an assumption even stronger than P $\neq$ NP, yet still believed to be true by many — we can show there exists a distribution that is EC but not ELNCP.

We prove our claim by defining a certain weighted language $\tilde{p}$ and reducing Sat to computing certain local conditional probabilities of $\tilde{p}$ (as defined in §2.2.1). Each decision Sat(ϕ) (where ϕ ranges over formulas) corresponds to a particular local conditional probability, implying that there is no polytime scheme for computing all of these probabilities, even with polynomially sized advice strings (*i.e.*, parameters).

Without loss of generality, we consider only formulae ϕ such that the set of variables mentioned at least once in ϕ is $\{A_1, \ldots, A_j\}$ for some $j \in \mathbb{N}$; we use $|\phi|$ to denote the number of variables j in ϕ. We say that **a satisfies** ϕ if $\mathbf{a} \in \mathbb{B}^{|\phi|}$ and $(A_1 = a_1, \ldots, A_{|\phi|} = a_{|\phi|})$ is a satisfying assignment. Finally, let boldface $\boldsymbol{\phi} \in \mathbb{B}^*$ denote enc(ϕ) where enc is a prefix-free encoding function. We can now define the unweighted language $\mathcal{X} = \{\boldsymbol{\phi}\mathbf{a} \mid \phi$ is a formula and $\mathbf{a} \in \mathbb{B}^{|\phi|}$ and $\mathbf{a}$ satisfies $\phi\}$ over alphabet $\mathbb{B}$, which contains each possible Sat problem concatenated to each of its solutions.[3]

We now convert $\mathcal{X}$ to a weighted language $\tilde{p}$, defined by $\tilde{p}(\mathbf{x}) = \tilde{p}(\boldsymbol{\phi}, \mathbf{a}) = (\frac{1}{3})^{|\mathbf{x}|+1}$ for $\mathbf{x} \in \mathcal{X}$ (otherwise $\tilde{p}(\mathbf{x}) = 0$). $\tilde{p}$ is normalizable since Z is both finite ($Z = \sum_{\mathbf{x} \in \mathbb{B}^*} \tilde{p}(\mathbf{x}) \leq$

[3]For example, $\mathcal{X}$ contains the string $\boldsymbol{\phi}\mathbf{a}$ where $\boldsymbol{\phi} = \text{enc}((A_1 \vee \neg A_2 \vee A_3) \wedge (A_1 \vee \neg A_4))$ and $\mathbf{a} = \texttt{1101}$.

$\sum_{\mathbf{x}\in\mathbb{B}^*}(\frac{1}{3})^{|\mathbf{x}|+1} = 1$) and positive ($Z > 0$ because the example string in footnote 3 has weight > 0). The conditional distribution $p(\mathbf{a} \mid \boldsymbol{\phi})$ is uniform over the satisfying assignments $\mathbf{a}$ of $\boldsymbol{\phi}$, as they all have the same length $|\phi|$.

$\tilde{p}$ is efficiently computable, and so is $p = \tilde{p}/Z$.[4] Yet deciding whether the local conditional probabilities of $\tilde{p}$ are greater than 0 is NP-hard. In particular, in Theorem 3.2.2 we show that SAT can be reduced to deciding whether certain local probabilities are greater than 0, namely the ones that condition on prefixes $\hat{\mathbf{x}}$ that consist only of a formula: $\hat{\mathbf{x}} = \boldsymbol{\phi}$ for some ϕ. This implies, assuming NP $\not\subseteq$ P/poly, that no $(M^{\mathrm{q}}, \boldsymbol{\Theta}^{\mathrm{q}})$ can efficiently locally normalize $\tilde{p}$ with compact parameters. Granted, the restriction of $\tilde{p}$ to the finite set $\{\mathbf{x} \in \mathbb{B}^* : |\mathbf{x}| \leq n\}$ can be locally normalized by some polytime Turing machine q_n, using the same trie trick sketched in §2.2.3.1. But such tries have sizes growing exponentially in n, and it is not possible to produce a sequence of such machines, $\{q_n : n \in \mathbb{N}\}$, via a single master Turing machine M^{q} that runs in $O(\mathrm{poly}(n))$ on $\boldsymbol{\theta}^{\mathrm{q}}_n$. That is:

Theorem 3.2.2. *Assuming* NP $\not\subseteq$ P/poly*, there exists a normalizable weighted language* $\tilde{p} \in$ EC *that is not ELNCP.*

Proof. The unweighted language $\tilde{p}$ defined earlier in this section is efficiently computable via the following simple algorithm that outputs $\tilde{p}(\mathbf{x})$ given $\mathbf{x} \in \mathbb{B}^*$. If $\mathbf{x}$ has a prefix that encodes a formula ϕ, and the remainder of $\mathbf{x}$ is a satisfying assignment $\mathbf{a}$ to the variables of ϕ, then return $(\frac{1}{3})^{|\mathbf{x}|+1}$. Otherwise return 0. This algorithm can be made to run in polynomial

[4] Almost. This Z could be irrational, but at least it is computable to any desired precision. For any rational $\hat{Z} \approx Z$, we can say $\hat{p} = \tilde{p}/\hat{Z} \approx p$ is EC, via a Turing machine $M^{\hat{p}}$ that stores $\hat{Z}$.

time because whether an assignment satisfies a formula can be determined in polynomial time (a fact that is standardly used to establish that SAT ∈ NP).

Given a formula ϕ with variables $A_1, \ldots, A_j$, we define $\phi' = (\neg A_1 \wedge \neg A_2 \wedge \ldots \wedge \neg A_j \wedge \neg A_{j+1}) \vee (A_1 \wedge \text{Shift}(\phi))$, where $\text{Shift}(\phi)$ is a version of ϕ in which A_i has been renamed to A_{i+1} for all $1 \leq i \leq j$. It is obvious that ϕ' and p have the properties stated in the proof sketch. The strings in $\mathcal{X}$ that begin with $\boldsymbol{\phi}'$ are precisely the strings of the form $\boldsymbol{\phi}'\mathbf{a}'$ where $\mathbf{a}'$ is a satisfying assignment of ϕ' — which happen just when $\mathbf{a}' = 0^{j+1}$ or $\mathbf{a}' = 1\mathbf{a}$ where $\mathbf{a}$ is a satisfying assignment of ϕ. At least one string in $\mathcal{X}$ begins with $\boldsymbol{\phi}'$, namely $\boldsymbol{\phi}'0^{j+1}$, so $Z(\boldsymbol{\phi}') > 0$. Moreover, $Z(\boldsymbol{\phi}'1) > 0$ iff ϕ has any satisfying assignments. Therefore the local probability $p(1 \mid \boldsymbol{\phi}') = Z(\boldsymbol{\phi}'1) \,/\, Z(\boldsymbol{\phi}')$ is defined (see §2.2), and is > 0 iff $\text{SAT}(\phi)$.

Notice that the formal problem used in the proof is a version of SAT whose inputs are encoded using the same prefix-free encoding function enc that was used by our definition of $\mathcal{X}$ in §3.2.2. We must choose this encoding function to be concise in the sense that $\boldsymbol{\phi} \triangleq \text{enc}(\phi)$ can be converted to and from the conventional encoding of ϕ in polynomial time. This ensures that our version of SAT is $\leq_m^P$-interreducible with the conventional version and hence NP-complete. It also ensures that there is a polynomial function f such that $|\boldsymbol{\phi}'| \leq f(|\boldsymbol{\phi}|)$, as required by the proof sketch, since there is a polynomial-time function that maps $\boldsymbol{\phi} \longrightarrow \phi \longrightarrow \phi' \longrightarrow \boldsymbol{\phi}'$ and the output length of this function is bounded by its runtime. This is needed to show that our version of SAT is in P/poly.

Specifically, to show that the existence of $(M^{\mathrm{q}}, \boldsymbol{\Theta}^{\mathrm{q}})$ implies $\text{SAT} \in \text{P/poly}$, we use it to construct an appropriate pair $(M, \boldsymbol{\Theta})$ such that $(M(\boldsymbol{\theta}_n))(\boldsymbol{\phi}) = \text{SAT}(\phi)$ if $|\boldsymbol{\phi}| = n$. We define $\boldsymbol{\Theta}$

by $\boldsymbol{\theta}_n = \boldsymbol{\theta}^{\mathrm{q}}_{f(n)}$, and observe that $|\boldsymbol{\theta}_n| \in O(\mathrm{poly}(n))$ (thanks to compactness of the parameters $\boldsymbol{\Theta}^{\mathrm{q}}$ and the fact that f is polynomially bounded). Finally, define $M(\boldsymbol{\theta}_n)$ to be a Turing machine that maps its input $\boldsymbol{\phi}$ of length n to $\boldsymbol{\phi}'$ of length $\leq f(n)$, then calls $M^{\mathrm{q}}(\boldsymbol{\theta}_n) = M^{\mathrm{q}}(\boldsymbol{\theta}^{\mathrm{q}}_{f(n)})$ on $\boldsymbol{\phi}'1$ to obtain $p(1 \mid \boldsymbol{\phi}')$, and returns true or false according to whether $p(1 \mid \boldsymbol{\phi}') > 0$. Computing $\boldsymbol{\phi}'$ takes time polynomial in n (thanks to the properties of enc). Constructing $M^{\mathrm{q}}(\boldsymbol{\theta}_{f(n)})$ and calling it on $\boldsymbol{\phi}'$ each take time polynomial in n (thanks to the properties of f and M^{q}). □

Remark on conditional models. While we focus on modeling *joint* sequence probabilities in this work, we note that in many applications it often suffices to just model *conditional* probabilities $p(\cdot \mid \hat{\mathbf{x}})$ (Sutskever, Vinyals, and Le, 2014). Unfortunately, our proof of Theorem 3.2.2 above implies that ELNCPs do not make good conditional models either: specifically, there exists $\boldsymbol{\phi}$ such that deciding whether $p(1 \mid \boldsymbol{\phi}) > 0$ is NP-hard, and thus beyond ELNCP's capability.

Remark on irrationality. In our definitions of ECCP and ELNCP languages, we implicitly assumed that the Turing machines that return weights or probabilities would write them in full on the output tape (*e.g.*, in the manner of weighted Turing machines introduced in §2.2.2). Such a Turing machine can only return rational numbers.

But then our formulation of Theorem 3.2.2 allows another proof. We could construct $\tilde{p}$ such that the local conditional probabilities $p(x \mid \hat{\mathbf{x}}) \triangleq Z(\hat{\mathbf{x}}x)/Z(\hat{\mathbf{x}})$ are sometimes irrational. In this case, they cannot be output exactly by a Turing machine, implying that $\tilde{p}$ is not ELNCP. However, this proof exposes only a trivial weakness of ELNCPs, namely the fact

that they can only define distributions whose local marginal probabilities are rational.

We can correct this weakness by formulating ELNCP languages slightly differently. A real number is said to be **computable** if it can be output by a Turing machine to any desired precision. That Turing machine takes an extra input b which specifies the number of bits of precision of the output. Similarly, our definitions of ECCP and ELNCP can be modified so that their respective Turing machines $\tilde{p}_n$ and q_n take this form, are allowed to run in time $O(\text{poly}(n + b))$, and have access to the respective parameter vectors $\boldsymbol{\Theta}^{\text{p}}_{n+b}$ and $\boldsymbol{\Theta}^{\text{q}}_{n+b}$. Since some of our results concern the ability to distinguish zero from small values (arbitrarily small in the case of Lemma 3.2.1), our modified definitions also require $\tilde{p}_n$ and q_n to output a bit indicating whether the output is *exactly* zero. For simplicity, we suppressed these technical details from our exposition.

Relatedly, in §3.4.3, we claimed that lookup models can fit any weighted language up to length n. This is not strictly true if the weights can be irrational. A more precise statement is that for any weighted language $\tilde{p}$, there is a lookup model that maps $(\mathbf{x}, b)$ to the first b bits of $\tilde{p}(\mathbf{x})$. Indeed, this holds even when $\tilde{p}(\mathbf{x})$ is uncomputable.

Remark on computability. In §2.2.1 we claimed that any weighted language $\tilde{p}$ that has a finite and strictly positive Z can be normalized as $p(\mathbf{x}) = \tilde{p}(\mathbf{x})/Z$. However, Z may be uncomputable: that is, there is no algorithm that takes number of bits of precision b as input, and outputs an approximation of Z within b bits of precision. Therefore, even if $\tilde{p}$ is computable, p may have weights that are not merely irrational but even *uncomputable*. An example appears in the proof of Lemma 3.2.1. Weighted language classes (*e.g.* ELNCP) that

only model normalized languages will not be able to model such languages, simply because the partition function is uncomputable.

However, our proof of Theorem 3.2.2 does not rely on this issue, because the $\tilde{p}$ that it exhibits happens to have a computable Z. For any b, Z may be computed to b bits of precision as the explicit sum $\sum_{\mathbf{x}:|\mathbf{x}|\leq N} \tilde{p}(\mathbf{x})$ for a certain large N that depends on b.

Remark on RNNs. Our proof of Theorem 3.2.2 showed that our problematic language $\tilde{p}$ is efficiently computable (though not by any locally normalized architecture with compact parameters). Because this chapter is in part a response to popular neural architectures, we now show that $\tilde{p}$ can in fact be computed efficiently *by a recurrent neural network (RNN)* with compact parameters. Thus, this is an example where a simple globally normalized RNN parameterization is fundamentally more efficient (in runtime or parameters) than any locally normalized parameterization of any architecture (RNN, Transformer, etc.).

Since we showed that $\tilde{p}$ is efficiently computable, the existence of an RNN implementation is established in some sense by the ability of finite rational-weighted RNNs to simulate Turing machines (Siegelmann and Sontag, 1992), as well as an extension to Chen et al. (2018b, Thm. 11) to a family of RNNs, where each RNN instance also takes some formula encoding as input. However, it is straightforward to give a concrete construction, for each $n \in \mathbb{N}$, for a simple RNN that maps each string $\mathbf{x} \in \mathbb{B}^n$ to $\tilde{p}(\mathbf{x})$. Here $\tilde{p}(\mathbf{x})$ will be either $(\frac{1}{3})^{n+1}$ or 0, according to whether $\mathbf{x}$ has the form $\boldsymbol{\phi}\mathbf{a}$ where $\boldsymbol{\phi}$ encodes a 3-CNF-SAT formula[5]

[5] 3-CNF-SAT formulae are conjunctive normal form (CNF) SAT formulae, where each clause in a formula contains no more than 3 variables. Deciding whether a 3-CNF-SAT formula is satisfiable is NP-complete.

ϕ that is satisfied by $\mathbf{a}$.[6] The basic idea is that $\boldsymbol{\phi}$ has $j \leq n$ variables, so there are only $O(n^3)$ possible 3-CNF clauses. The RNN allocates one hidden unit to each of these. When reading $\boldsymbol{\phi}\mathbf{a}$, each clause encountered in $\boldsymbol{\phi}$ causes the corresponding hidden unit to turn on, and then each literal encountered in $\mathbf{a}$ turns off the hidden units for all clauses that would be satisfied by that literal. If any hidden units remain on after $\mathbf{x}$ has been fully read, then $\boldsymbol{\phi}$ was not satisfied by $\mathbf{a}$, and the RNN's final output unit should return 0. Otherwise it should return $(\frac{1}{3})^{n+1}$, which is constant for this RNN. To obtain digital behaviors such as turning hidden units on and off, it is most convenient to use ramp activation functions for the hidden units and the final output unit, rather than sigmoid activation functions. Note that our use of a separate RNN M_n^{RNN} for each input length n is an example of using more hidden units for larger problems, namely the non-uniform computation paradigm that we introduced in §2.2.3.1 in order to look at asymptotic behavior. The RNN's parameter sequence $\boldsymbol{\Theta}^{\text{RNN}} = \{\boldsymbol{\theta}_n^{\text{RNN}} \mid n \in \mathbb{N}\}$ is obviously compact, as $\boldsymbol{\theta}_n^{\text{RNN}}$ only has to store the input length n. With our alphabet $\mathbb{B}$ for $\tilde{p}$, $|\boldsymbol{\theta}_n^{\text{RNN}}| \in O(\log n)$.

3.2.3 ELNCP models cannot even capture all EC (or ECCP) supports or rankings

We can strengthen Theorem 3.2.2 as follows:

[6] The restriction to 3-CNF-SAT formulas is convenient, but makes this a slightly different definition of $\mathcal{X}$ and $\tilde{p}$ than we used in the proofs above. Those proofs can be adjusted to show that this $\tilde{p}$, too, cannot be efficiently locally normalized with compact parameters. The only change is that in the construction of Theorem 3.2.2, ϕ' must be converted to 3-CNF. The proof then obtains its contradiction by showing that 3-CNF-SAT $\in$ P/poly (which suffices since 3-CNF-SAT is also NP-complete).

Theorem 3.2.3. *Assuming* NP $\not\subseteq$ P/poly*, there exists an efficiently computable normalizable weighted language* $\tilde{p}$ *where there is no ELNCP* $\tilde{q}$ *such that* $\text{support}(\tilde{p}) = \text{support}(\tilde{q})$.

Proof. Observe that for any two weighted languages $\tilde{p}$ and $\tilde{q}$ with the same support, $\forall \hat{\mathbf{x}} \in V^*, Z_{\tilde{p}}(\hat{\mathbf{x}}) > 0 \iff Z_{\tilde{q}}(\hat{\mathbf{x}}) > 0$ (where $Z_{\tilde{p}}$ and $Z_{\tilde{q}}$ return the prefix probabilities of $\tilde{p}$ and $\tilde{q}$ respectively). Thus, for any $\hat{\mathbf{x}}$ with $Z_{\tilde{p}}(\hat{\mathbf{x}}) > 0$, $p(1 \mid \hat{\mathbf{x}}) \triangleq Z_{\tilde{p}}(\hat{\mathbf{x}}1)/Z_{\tilde{p}}(\hat{\mathbf{x}})$ and $q(1 \mid \hat{\mathbf{x}}) \triangleq Z_{\tilde{q}}(\hat{\mathbf{x}}1)/Z_{\tilde{q}}(\hat{\mathbf{x}})$ are well-defined and $p(1 \mid \hat{\mathbf{x}}) > 0 \iff q(1 \mid \hat{\mathbf{x}}) > 0$. If $\tilde{q}$ is ELNCP, then all such probabilities $q(1 \mid \hat{\mathbf{x}})$ can be computed in polytime with compact parameters, so it is likewise efficient to determine whether $p(1 \mid \hat{\mathbf{x}}) > 0$. But this cannot be the case when $\tilde{p}$ is the weighted language used in the proof of Theorem 3.2.2, since that would suffice to establish that SAT $\in$ P/poly, following the proof of that theorem. □

To put this another way, there exists an unweighted language in P (namely $\text{support}(\tilde{p})$) that is not the support of *any* ELNCP distribution.

If they have different support, normalizable languages also differ in their ranking of strings:

Lemma 3.2.4. *Let* $\tilde{p}, \tilde{q}$ *be normalizable weighted languages. If* $\forall \mathbf{x}_1 \in \mathbb{B}^*, \forall \mathbf{x}_2 \in \mathbb{B}^*, \tilde{p}(\mathbf{x}_1) < \tilde{p}(\mathbf{x}_2) \iff \tilde{q}(\mathbf{x}_1) < \tilde{q}(\mathbf{x}_2)$*, then* $\text{support}(\tilde{p}) = \text{support}(\tilde{q})$.

Proof. Suppose that the claim is false, *i.e.*, $\tilde{p}$ and $\tilde{q}$ have the same ranking of strings, but that the supports are different. Then the minimum-weight strings under $\tilde{p}$ must also be minimum-weight under $\tilde{q}$. WLOG, there exists $\mathbf{x} \in V^*$ with $\tilde{p}(\mathbf{x}) = 0$ and $\tilde{q}(\mathbf{x}) = c > 0$, because the supports are different. Then $c > 0$ is the minimum weight of strings in $\tilde{q}$. But this

is not possible for a normalizable language $\tilde{q}$, since it means that $Z_{\tilde{q}} \triangleq \sum_{\mathbf{x}' \in V^*} q(\mathbf{x}') \geq \sum_{\mathbf{x}' \in V^*} c$ diverges. □

Therefore, no ELNCP $\tilde{q}$ captures the string ranking of $\tilde{p}$ from Theorem 3.2.3. And for some $\tilde{p}$, any ELNCP $\tilde{q}$ misranks even string pairs of "similar" lengths:

Theorem 3.2.5. *Assuming* NP $\not\subseteq$ P/poly*, there exists an efficiently computable normalizable weighted language $\tilde{p}$ such that no ELNCP $\tilde{q}$ with* support$(\tilde{q}) \supseteq$ support$(\tilde{p})$ *has* $\tilde{p}(\mathbf{x}_1) < \tilde{p}(\mathbf{x}_2) \implies \tilde{q}(\mathbf{x}_1) < \tilde{q}(\mathbf{x}_2)$ *for all* $\mathbf{x}_1, \mathbf{x}_2 \in V^*$*. Indeed, any such $\tilde{q}$ has a counterexample where $\tilde{p}(\mathbf{x}_1) = 0$. Moreover, there is a polynomial $f_{\tilde{q}} : \mathbb{N} \longrightarrow \mathbb{N}$ such that a counterexample exists for* every $\mathbf{x}_1$ *such that $\tilde{p}(\mathbf{x}_1) = 0$ and $\tilde{q}(\mathbf{x}_1) > 0$, where the $\mathbf{x}_2$ in this counterexample always satisfies $|\mathbf{x}_2| \leq f_{\tilde{q}}(|\mathbf{x}_1|)$.*

Proof. Let $\tilde{p}$ be the weighted language from Theorem 3.2.3. Given an ELNCP $\tilde{q}$. By Theorem 3.2.3, support$(\tilde{q}) \neq$ support$(\tilde{p})$, so there must exist a string $\mathbf{x}_1$ that is in one support language but not the other. With the additional assumption that support$(\tilde{q}) \supseteq$ support$(\tilde{p})$, it must be that $\mathbf{x}_1 \in$ support$(\tilde{q})$, so $\tilde{p}(\mathbf{x}_1) = 0$ but $\tilde{q}(\mathbf{x}_1) > 0$.

Given *any* such $\mathbf{x}_1$ with $\tilde{p}(\mathbf{x}_1) = 0$ but $\tilde{q}(\mathbf{x}_1) > 0$, we must find a $\mathbf{x}_2$ of length $O(\text{poly}(|\mathbf{x}_1|))$ with $\tilde{p}(\mathbf{x}_2) > 0$ but $\tilde{q}(\mathbf{x}_2) \leq \tilde{q}(\mathbf{x}_1)$.

To ensure that $\tilde{p}(\mathbf{x}_2) > 0$, let us use the structure of $\tilde{p}$. For any j, we can construct a tautological formula ϕ over variables $A_1, \ldots A_j$, as $\phi = (A_1 \vee \neg A_1) \wedge \cdots \wedge (A_j \vee \neg A_j)$. It follows that $\tilde{p}(\boldsymbol{\phi}\mathbf{a}) > 0$ for any $\mathbf{a} \in \mathbb{B}^j$. We will take $\mathbf{x}_2 = \boldsymbol{\phi}\mathbf{a}$ for a particular choice of j and $\mathbf{a}$.

Specifically, we choose them to ensure that $\tilde{q}(\mathbf{x}_2) \leq \tilde{q}(\mathbf{x}_1)$. Since $\tilde{q}$ is ELNCP, it is normalizable and hence has a finite Z. Thus, $\sum_{\mathbf{a} \in \mathbb{B}^j} \tilde{q}(\boldsymbol{\phi}\mathbf{a}) \leq Z$. So there must exist some

$\mathbf{a} \in \mathbb{B}^j$ such that $\tilde{q}(\boldsymbol{\phi}\mathbf{a}) \leq Z/2^j$. We choose that $\mathbf{a}$, after choosing j large enough such that $Z/2^j \leq \tilde{q}(\mathbf{x}_1)$. Then $\tilde{q}(\mathbf{x}_2) = \tilde{q}(\boldsymbol{\phi}\mathbf{a}) \leq Z/2^j \leq \tilde{q}(\mathbf{x}_1)$.

To achieve the last claim of the theorem, we must also ensure that $|\mathbf{x}_2| \in O(\text{poly}(|\mathbf{x}_1|))$. Observe that $\tilde{q}(\mathbf{x}_1)$ can be computed in polytime (with access to compact parameters), by Lemma 2.2.3. But this means that the representation of $\tilde{q}(\mathbf{x}_1) > 0$ as a rational number must have $\leq g(|\mathbf{x}_1|)$ bits for some polynomial g. Then $\tilde{q}(\mathbf{x}_1) \geq 2^{-g(|\mathbf{x}_1|))}$, and it suffices to choose $j = \lceil g(|\mathbf{x}_1|) + \log_2 Z \rceil$ to ensure that $Z/2^j \leq 2^{-g|\mathbf{x}_1|} \leq \tilde{q}(\mathbf{x}_1)$ as required above.

But then $j \in O(\text{poly}(|\mathbf{x}_1|))$. Also, recall that the encoding function enc used in the construction of $\tilde{p}$ is guaranteed to have only polynomial blowup (see the proof of Theorem 3.2.3). Thus, $|\mathbf{x}_2| = |\boldsymbol{\phi}| + |\mathbf{a}| = |\text{enc}(\phi)| + j \in O(\text{poly}(j)) \subseteq O(\text{poly}(|\mathbf{x}_1|))$ as required by the theorem. □

Theorem 3.2.5 is relevant if one wishes to train a model $\tilde{q}$ to rerank strings that are proposed by another method (*e.g.*, beam search on $\tilde{q}$, or exact k-best decoding from a more tractable distribution). If the desired rankings are given by Theorem 3.2.5's $\tilde{p}$, any smoothed[7] ELNCP model $\tilde{q}$ will misrank some sets of candidate strings, even sets all of whose strings are "close" in length, by failing to rank an impossible string ($\mathbf{x}_1$ with $\tilde{p}(\mathbf{x}_1) = 0$) below a possible one ($\mathbf{x}_2$ with $\tilde{p}(\mathbf{x}_2) > 0$).

[7] Smoothing is used to avoid ever incorrectly predicting 0 (a "false negative") by ensuring $\text{support}(\tilde{q}) \supseteq \text{support}(\tilde{p})$. E.g., autoregressive language models often define $q(x \mid \hat{\mathbf{x}})$ using a softmax over $V \cup \{\text{EOS}\}$, ensuring that $q(\mathbf{x}) > 0$ for all $\mathbf{x} \in V^*$.

3.2.4 ELNCP models cannot even *approximate* EC (or ECCP) distributions

Theorem 3.2.3 implies that there exists $\tilde{p}$ whose local probabilities $p(x \mid \hat{\mathbf{x}})$ are not approximated by any ELNCP q to within any constant factor λ, since that would perfectly distinguish zeroes from non-zeroes and the resulting support sets would be equal.[8]

However, this demonstration hinges on the difficulty of multiplicative approximation of zeroes — whereas real-world distributions may lack zeroes. Below we further show that it is hard even to approximate the *non-zero* local conditional probabilities (even with the additional help of randomness).

Theorem 3.2.6. *Assuming* NP $\not\subseteq$ P/poly*, there exists an efficiently computable weighted language* $\tilde{p} : V^* \longrightarrow \mathbb{R}_{\geq 0}$ *such that there is no* $(M^{\mathrm{q}}, \boldsymbol{\Theta}^{\mathrm{q}})$ *where* $\boldsymbol{\Theta}^{\mathrm{q}} = \{\boldsymbol{\theta}^{\mathrm{q}}_n \mid n \in \mathbb{N}\}$ *that satisfies all of the following properties:*

- *the parameter size* $|\boldsymbol{\theta}^{\mathrm{q}}_n|$ *grows only as* $O(\mathrm{poly}(n))$
- $M^{\mathrm{q}}(\boldsymbol{\theta}^{\mathrm{q}}_n)$ *returns a* probabilistic *Turing machine* q_n *in time* $O(\mathrm{poly}(n))$
- *there exists* $\lambda \geq 1$ *such that for each* $x \in V \cup \{\$\}$ *and* $\hat{\mathbf{x}} \in V^*$ *with* $|\hat{\mathbf{x}}| \leq n$ *and* $p(x \mid \hat{\mathbf{x}}) > 0$*, the probabilistic computation* $q_n(\hat{\mathbf{x}}x)$ *has probability* $> 2/3$ *of approximating* $p(x \mid \hat{\mathbf{x}})$ *to within a factor of* λ *(that is,* $q_n(\hat{\mathbf{x}}x)/p(x \mid \hat{\mathbf{x}}) \in [1/\lambda, \lambda]$*)*

[8] Dropping the normalization requirement on the approximated local probabilities (so that $q \notin$ ELNCP, and possibly $\sum_{x \in V} q(x \mid \hat{\mathbf{x}}) \neq 1$) does not help. Otherwise, again, SAT could be solved in polynomial time (with the help of polysize advice strings) by using $q(1 \mid \boldsymbol{\phi}')$ to determine in the proof of Theorem 3.2.2 whether $p(1 \mid \boldsymbol{\phi}') > 0$.

- *q_n runs on those inputs $\hat{\mathbf{x}}x$ in time $O(\text{poly}(n))$*

Moreover, the statement above remains true

(a) when the approximation guarantee is only required to hold for prefixes $\hat{\mathbf{x}}$ where $\{\mathbf{x} : \hat{\mathbf{x}} \preceq \mathbf{x}\}$ is finite (so $p(x \mid \hat{\mathbf{x}})$ is computable by brute force)

(b) or, when $\text{support}(\tilde{p}) = V^*$

Lemma 3.2.7. *The first part of Theorem 3.2.6 (without the modifications (a) and (b)).*

We first prove the first part of Theorem 3.2.6 (which is restated in full below). In this case we will use a distribution $\tilde{p}$ that does not have support V^* (so it does not prove modification (b)).

Proof. We take $\tilde{p}$ to be the weighted language that was defined in §3.2.2, which was already shown to be efficiently computable. Suppose $(M^{\mathrm{q}}, \boldsymbol{\Theta}^{\mathrm{q}}, \lambda)$ is a counterexample to Lemma 3.2.7. Choose integer $k \geq 1$ in a manner (dependent only on λ) to be described at the end of the proof.

Suppose we would like to answer SAT where ϕ is a formula with variables $A_1, \ldots, A_j$. Define $\phi' = (\neg A_1 \wedge \neg A_2 \wedge \ldots \wedge \neg A_j \wedge \neg A_{j+1} \wedge \neg A_{j+k}) \vee (A_1 \wedge \text{Shift}(\phi))$. Note that ϕ' augments ϕ with k additional variables, namely A_1 and $A_{j+2,\ldots,j+k}$. For $k = 1$, this is the same construction as in the proof of Theorem 3.2.2. Let $n = |\boldsymbol{\phi}'|$ and note that n is polynomial in the size of ϕ (holding k constant).

The strings in $\mathcal{X} = \text{support}(\tilde{p})$ that begin with $\boldsymbol{\phi}'$ are precisely the strings of the form $\boldsymbol{\phi}'\mathbf{a}'$ where $\mathbf{a}'$ is a satisfying assignment of ϕ'. This is achieved precisely when $\mathbf{a}' = 0^{j+k}$ or $\mathbf{a}' = 1\mathbf{a}\vec{b}$ where $\mathbf{a}$ is a satisfying assignment of ϕ and $\vec{b} \in \mathbb{B}^{k-1}$.

By our definition of $\tilde{p}$, all strings in $\mathcal{X}$ that begin with $\boldsymbol{\phi}'$ have equal weight under $\tilde{p}$. Call this weight w.[9] Clearly $Z(\boldsymbol{\phi}'0) = w$, and $Z(\boldsymbol{\phi}'1) = w \cdot 2^{k-1} \cdot$(number of satisfying assignments of ϕ).

Recall that $p(0 \mid \boldsymbol{\phi}') = Z(\boldsymbol{\phi}'0)/(Z(\boldsymbol{\phi}'0) + Z(\boldsymbol{\phi}'1))$. Let us abbreviate this quantity by p. It follows from the previous paragraph that if ϕ is unsatisfiable, then $p = 1$, but if ϕ is satisfiable, then $p \leq 1/(1+2^{k-1})$. By hypothesis, p is approximated (with error probability $< 1/3$) by the possibly random quantity $(M^{\mathrm{q}}(\boldsymbol{\theta}^{\mathrm{q}}_{|\boldsymbol{\phi}'|}))(\boldsymbol{\phi}'0)$, which we abbreviate by q, to within a factor of λ. That is, $p \in [q/\lambda, \lambda q]$. By choosing k large enough[10] such that $[q/\lambda, \lambda q]$ cannot contain both 1 and $1/(1+2^{k-1})$, we can use q to determine whether $p = 1$ or $p \leq 1/(1+2^{k-1})$. This allows us to determine $\textsc{Sat}(\phi)$ in polynomial time with error probability $< 1/3$, since by hypothesis q is computable in polynomial time with compact parameters. This shows that $\textsc{Sat} \in \text{BPP/poly} = \text{P/poly}$ (Adleman, 1978), implying $\text{NP} \subseteq \text{P/poly}$, contrary to our assumption. (BPP/poly is similar to P/poly but allows M^{q} to be a bounded-error probabilistic Turing machine.) □

Based on Lemma 3.2.7, we then proceed to prove Theorem 3.2.6:

Theorem 3.2.6. *Assuming* NP $\not\subseteq$ P/poly, *there exists an efficiently computable weighted language* $\tilde{p} : V^* \longrightarrow \mathbb{R}_{\geq 0}$ *such that there is no* $(M^{\mathrm{q}}, \boldsymbol{\Theta}^{\mathrm{q}})$ *where* $\boldsymbol{\Theta}^{\mathrm{q}} = \{\boldsymbol{\theta}^{\mathrm{q}}_n \mid n \in \mathbb{N}\}$ *that*

[9] Specifically, each such string has length $n + j + k$, so $\tilde{p}$ gives it a weight of $w = (\frac{1}{3})^{n+j+k+1}$.

[10] It suffices to ensure that $1 + 2^{k-1} > \lambda^2$, so take any $k > 1 + \log_2(\lambda^2 - 1)$.

satisfies all of the following properties:

- *the parameter size* $|\boldsymbol{\theta}_n^{\mathrm{q}}|$ *grows only as* $O(\mathrm{poly}(n))$
- $M^{\mathrm{q}}(\boldsymbol{\theta}_n^{\mathrm{q}})$ *returns a* probabilistic *Turing machine* q_n *in time* $O(\mathrm{poly}(n))$
- *there exists* $\lambda \geq 1$ *such that for each* $x \in V \cup \{\$\}$ *and* $\hat{\mathbf{x}} \in V^*$ *with* $|\hat{\mathbf{x}}| \leq n$ *and* $p(x \mid \hat{\mathbf{x}}) > 0$*, the probabilistic computation* $q_n(\hat{\mathbf{x}}x)$ *has probability* $> 2/3$ *of approximating* $p(x \mid \hat{\mathbf{x}})$ *to within a factor of* λ *(that is,* $q_n(\hat{\mathbf{x}}x)/p(x \mid \hat{\mathbf{x}}) \in [1/\lambda, \lambda]$*)*
- q_n *runs on those inputs* $\hat{\mathbf{x}}x$ *in time* $O(\mathrm{poly}(n))$

Moreover, the statement above remains true

(a) when the approximation guarantee is only required to hold for prefixes $\hat{\mathbf{x}}$ *where* $\{\mathbf{x} : \hat{\mathbf{x}} \preceq \mathbf{x}\}$ *is finite (so* $p(x \mid \hat{\mathbf{x}})$ *is computable by brute force)*

(b) or, when $\mathrm{support}(\tilde{p}) = V^*$

Proof. It remains to show that the statement remains true with modification (a) and with modification (b). For (a), the proof of Lemma 3.2.7 suffices, since it reduces SAT to approximate local probability queries of the stated form. That is, the true local probabilities $p(x \mid \hat{\mathbf{x}})$ can be computed with finite summations, thanks to the structure of our example language $\tilde{p}$, which guarantees that the prefix $\hat{\mathbf{x}}$ can only continue with suffixes of a fixed length that is easily determined from $\hat{\mathbf{x}}$.

For modification (b), again let $V = \mathbb{B} = \{0, 1\}$. Choose some $\epsilon > 0$ (any choice will do),

and let

$$\tilde{p}_1(\mathbf{x}) = \begin{cases} (\frac{1}{3})^{|\mathbf{x}+1|} & \text{if } \mathbf{x} = \boldsymbol{\phi}\mathbf{a} \text{ where } \boldsymbol{\phi} = \text{enc}(\phi) \\ & \quad \text{and } \mathbf{a} \text{ satisfies } \phi \\ 0 & \text{otherwise} \end{cases}$$

$$\tilde{p}_2(\mathbf{x}) = (\tfrac{1}{9})^{|\mathbf{x}+1|} > 0$$

$$\tilde{p}(\mathbf{x}) = \tilde{p}_1(\mathbf{x}) + \epsilon \cdot \tilde{p}_2(\mathbf{x})$$

We use Z_1, Z_2, and Z respectively to denote normalizing constants of these three weighted languages. Note that $\tilde{p}_1$ is the weighted language that was previously used in the proofs of Theorem 3.2.2 and Lemma 3.2.7. Our new $\tilde{p}$ is intended to be very similar while satisfying the additional condition (b). It is easy to show that $\tilde{p}$ is efficiently computable, much as we showed for $\tilde{p}_1$ in Theorem 3.2.2. Also, $\tilde{p}$ is normalizable, since $Z = Z_1 + \epsilon \cdot Z_2$, where $Z_1 \leq (\frac{1}{3})/(1 - \frac{2}{3}) = 1$ and $Z_2 = (\frac{1}{9})/(1 - \frac{2}{9}) = \frac{1}{7}$ are both finite.

The proof proceeds as in Lemma 3.2.7, with ϕ' constructed from ϕ as before. Recall that ϕ has j variables, ϕ' has $j + k$ variables, and $|\boldsymbol{\phi}'| = n$. We may assume WLOG that the encoding function enc is such that an encoded formula always has at least as many bits as the number of variables in the formula, so $n \geq j + k$.

Notice that $Z_1(\boldsymbol{\phi}')$ sums over the satisfying assignments of ϕ', and there may be as few as one of these (if ϕ is unsatisfiable). By contrast, $Z_2(\boldsymbol{\phi}')$ sums over an infinite number of continuations with positive probability. The faster decay rate of $\frac{1}{9}$ in $\tilde{p}_2$ was chosen to keep

$Z_2(\boldsymbol{\phi}')$ small relative to $Z_1(\boldsymbol{\phi}')$ despite this. Specifically,

$$
\begin{aligned}
Z_1(\boldsymbol{\phi}'0) &= (\tfrac{1}{3})^{n+j+k+1} \\
Z_1(\boldsymbol{\phi}'1) &= (\tfrac{1}{3})^{n+j+k+1} \cdot 2^{k-1} \\
&\qquad \cdot (\text{\# of satisfying assignments of } \phi) \\
Z_2(\boldsymbol{\phi}'0) &= (\tfrac{1}{9})^{n} \cdot \tfrac{1}{9} \cdot (\tfrac{1}{9}/(1 - \tfrac{2}{9})) \\
&= \tfrac{1}{7} \cdot (\tfrac{1}{3})^{2(n+1)} \\
&< \tfrac{1}{7} \cdot Z_1(\boldsymbol{\phi}'0) \\
&\qquad (\text{because } 2(n+1) > n + j + k + 1) \\
Z_2(\boldsymbol{\phi}'1) &= Z_2(\boldsymbol{\phi}'0)
\end{aligned}
$$

As in the proof of Lemma 3.2.7, we will show that $p(0 \mid \boldsymbol{\phi}')$ is much larger when ϕ is unsatisfiable. Recall that $Z(\hat{\mathbf{x}}) = Z_1(\hat{\mathbf{x}}) + \epsilon \cdot Z_2(\hat{\mathbf{x}})$. When ϕ has zero satisfying assignments,

$$
\begin{aligned}
p(0 \mid \boldsymbol{\phi}') &= \frac{Z(\boldsymbol{\phi}'0)}{Z(\boldsymbol{\phi}'0) + Z(\boldsymbol{\phi}'1)} \\
&= \frac{Z(\boldsymbol{\phi}'0)}{Z_1(\boldsymbol{\phi}'0) + \epsilon \cdot Z_2(\boldsymbol{\phi}'0) + \epsilon \cdot Z_2(\boldsymbol{\phi}'1)} \\
&> \frac{Z(\boldsymbol{\phi}'0)}{Z_1(\boldsymbol{\phi}'0) + 2 \cdot \frac{\epsilon}{7} \cdot Z_1(\boldsymbol{\phi}'0)}
\end{aligned}
$$

whereas if ϕ has at least one satisfying assignment, then

$$\begin{aligned} p(0 \mid \boldsymbol{\phi}') &= \frac{Z(\boldsymbol{\phi}'0)}{Z(\boldsymbol{\phi}'0) + Z(\boldsymbol{\phi}'1)} \\ &< \frac{Z(\boldsymbol{\phi}'0)}{Z_1(\boldsymbol{\phi}'0) + Z_1(\boldsymbol{\phi}'1)} \\ &\leq \frac{Z(\boldsymbol{\phi}'0)}{Z_1(\boldsymbol{\phi}'0) + 2^{k-1} Z_1(\boldsymbol{\phi}'0)} \end{aligned}$$

This rewrites both probabilities in terms of $Z_\cdot(\boldsymbol{\phi}'0)$ quantities, which do not depend on the number of satisfying assignments. So now we can see that the first probability is at least $(1+2^{k-1}) \,/\, (1+\frac{2\epsilon}{7})$ times as large as the second probability. Choose k large enough[11] such that $[q/\lambda, \lambda q]$ cannot contain both probabilities, and complete the proof as in Lemma 3.2.7. □

3.3 Effects of (discrete) latent variables

Autoregressive models have $Z = 1$ for any setting of the parameters (or at least any setting that guarantees consistency: see footnote 7). Clearly $Z = 1$ ensures that Z is both finite and tractable. Can we find a model family that retains this convenience (unlike EBMs), while still being expressive enough to have any non-empty language in P as support?

Autoregressive *latent-variable* models form such a family. As in directed graphical models, the use of latent variables provides a natural way to model partial observations of an underlying stochastic sequence of events. We will model an observed string $\mathbf{x}$ of length n as a function of a latent string $\mathbf{z}$ of length $O(\text{poly}(n))$, and let the weighted language

[11] It suffices to ensure that $(1 + 2^{k-1})/(1 + \frac{2\epsilon}{7}) > \lambda^2$, so take any $k > 1 + \log_2(\lambda^2 \cdot (1 + \frac{2\epsilon}{7}) - 1)$.

of $\mathbf{z}$ be $\in$ ELNCP. As in EBMs, the probability $p(\mathbf{x})$ can be computationally intractable, allowing these models to break the expressivity bottleneck of ordinary autoregressive models. However, the intractability no longer comes from exponentially many summands in the denominator Z (which is a constant), but rather from exponentially many summands in the numerator — namely, the summation over all latent $\mathbf{z}$ that could have produced $\mathbf{x}$. Notice that as a result, even unnormalized string weights are now hard to compute, although once computed they are already normalized.

Formally, we define **marginalized** weighted languages. We say that $\tilde{p}$ is a **marginalization** of the weighted language $\tilde{r}$ if it can be expressed as $\tilde{p}(\mathbf{x}) = \sum_{\mathbf{z}:\mu(\mathbf{z})=\mathbf{x}} \tilde{r}(\mathbf{z})$, where $\mu : \mathcal{Z} \longrightarrow V^*$ is some function (the **marginalization operator**). We say it is a **light marginalization** if $|\mathbf{z}| \in O(\text{poly}(|\mu(\mathbf{z})|))$ and μ runs in time $O(\text{poly}(|\mathbf{z}|))$.Typically $\mu(\mathbf{z})$ extracts a subsequence of $\mathbf{z}$; it can be regarded as keeping the observed symbols while throwing away a polynomially bounded number of latent symbols.

Light marginalizations of ELN distributions are a reasonable formalization of latent-variable autoregressive models. They are more powerful than ELN distributions, and even include some distributions that (by Lemma 2.2.4) are not even ELNCP or ECCP:

Theorem 3.3.1. *There exists a light marginalization p of an ELN distribution, such that* support(p) *is an* NP*-complete language.*

Proof. We will construct p such that support(p) is the NP-complete language SAT of all satisfiable boolean formulas. The idea is to construct an ELN distribution r that can autoregressively generate any assignment $\mathbf{a}$ followed by any formula ϕ that is satisfied by

$\mathbf{a}$. Thus, if we delete the $\mathbf{a}$ prefixes, the support consists of exactly the satisfiable formulas ϕ (or more precisely, their encodings $\boldsymbol{\phi}$).

To be more precise, we will have support(r) be the language

$$L = \{\mathbf{a}\#\boldsymbol{\phi} \mid \mathbf{a} \in \mathbb{B}^* \text{ and } \phi \text{ is a formula satisfied by } \mathbf{a}\}.$$

This is defined similarly to the support language L in §3.2.2, but with the order of $\boldsymbol{\phi}$ and $\mathbf{a}$ crucially swapped: r will now generate the "solution" $\mathbf{a}$ *before* the "problem" $\boldsymbol{\phi}$. The alphabet V of this language contains at least the symbols $\{0, 1, \#\}$, where # is a separator symbol, and any other symbols needed to encode ϕ as $\boldsymbol{\phi}$. The marginalization operator μ maps $\mathbf{a}\#\boldsymbol{\phi}$ to $\boldsymbol{\phi}$.

Let $j = |\mathbf{a}|$. As in §3.2.2, we will require ϕ to use all of the variables $A_1, \ldots, A_j$ (and only those variables), implying that $|\boldsymbol{\phi}| \geq j$. This ensures that marginalizing over the $j + 1$ latent symbols is only light marginalization since $j + 1 + |\boldsymbol{\phi}| \in O(\text{poly}(|\boldsymbol{\phi}|))$. For convenience, we will also require ϕ to be a CNF formula. These requirements shrink support(p) but do not affect its NP-completeness.

The remaining challenge is to construct an autoregressive distribution r whose support is L. We can think of this distribution as describing an efficient procedure for randomly generating a string from left to right so that the procedure generates the t^{th} symbol in time $O(\text{poly}(t))$, terminates with probability 1,[12] has positive probability of producing any

[12]Phase 1 almost surely terminates after a finite number of bits. Phase 2 almost surely terminates after a finite number of clauses, and each clause almost surely terminates after a finite number of literals. "Almost surely" means "with probability 1."

string in L, and has zero probability of producing any string not in L. Below we give such a procedure.[13]

1. First, the procedure generates $\mathbf{a}\#$ as a sequence of random symbols from $\{0, 1, \#\}$, making a uniform draw at each step. It stops immediately after generating # for the first time. The string generated before # is called $\mathbf{a}$ and we let $j = |\mathbf{a}|$. For example, $\mathbf{a} = 010$ and $j = 3$.

2. Second, the procedure must generate the encoding $\boldsymbol{\phi}$ of a random CNF formula ϕ that is satisfied by $\mathbf{a}$, such as $(A_2 \vee \neg A_3 \vee \neg A_2 \vee A_2) \wedge (\neg A_1)$ in our example. This involves generating a random sequence of 0 or more satisfied clauses connected by $\wedge$. At each step, the procedure decides whether to generate a new clause or end the formula. The probability of generating a new clause is ordinarily ½. However, this probability is 1 if the previous clauses do not yet mention all the variables $A_1, \ldots, A_j$.

   How does it generate each satisfied clause? This involves generating a sequence of literals connected by $\vee$, at least one of which must be true. At each step of this subroutine, it uniformly chooses an integer $i \in [1, j]$, and then flips a fair coin to decide whether to add the literal A_i or $\neg A_i$ to the current clause. If the clause is now satisfied by $\mathbf{a}$ (*i.e.*, at least one of the literals is true), it then flips another fair coin to decide whether to end the clause.

r is ELN because there exists a Turing machine that computes from input $\hat{\mathbf{x}}x$ — in time

[13] Our presentation here makes use of an infinite alphabet that includes symbols such as A_i and $\neg A_i$ for all $i \in \mathbb{N}_{>0}$, as well as symbols such as $0, 1, \wedge, \vee$. We implicitly invoke some prefix-free encoding scheme to translate each symbol into a fixed string over the finite alphabet V.

$O(\text{poly}(|\hat{\mathbf{x}}|))$ — the probability that the next symbol generated after the prefix $\hat{\mathbf{x}}$ would be x, under the above procedure. As discussed in footnote 7, that probability equals $r(x \mid \hat{\mathbf{x}})$ — which is what our Turing machine is required to return — because the above procedure almost surely terminates (footnote 12), ensuring that r is a consistent probability distribution over V^* (that is, $\sum_{\mathbf{x} \in V^*} r(\mathbf{x}) = 1$). □

Our proof of Theorem 3.3.1 relies on special structure of a certain NP-complete language (SAT) and does not evidently generalize to all languages in NP.

However, light marginalizations of *ELNCP* distributions are more powerful still,[14] and can have *any* language ∈ NP or even NP/poly (§2.2.5.1) as support:

Theorem 3.3.2. *The following statements are equivalent for any nonempty $L \subseteq V^*$:*

(a) $L \in$ NP/poly.

(b) L is the support of a light marginalization of an ELNCP distribution.

(c) L is the support of a light marginalization of an ECCP weighted language.

Proof. (b) implies (c) since any ELNCP distribution is an ECCP weighted language (Lemma 2.2.3). (c) implies (a) by Lemma 3.3.3 below. Finally, (a) implies (b) by Lemma 3.3.4 below. □

Lemma 3.3.3. *For any ECCP weighted language $\tilde{r}$, if $\tilde{p}$ is a light marginalization of $\tilde{r}$, then* $\text{support}(\tilde{p}) \in$ NP/poly.

[14] The capacity established by Theorem 3.3.2 does not need the full power of marginalization. We could similarly define light *maximizations* of ELNCP distributions, $\tilde{p}(\mathbf{x}) = \max_{\mathbf{z}\,:\,\mu(\mathbf{z})=\mathbf{x}} \tilde{r}(\mathbf{z})$. Replacing sum by max does not change the support.

Notice that this lemma concerns the class NP/poly, not P/poly (see §2.2.5.1). The proof is straightforward.

Proof. Suppose $\tilde{r}$ is ECCP via $(M^{\tilde{r}}, \boldsymbol{\theta}^{\tilde{r}})$, and μ is the marginalization operator such that $\tilde{p}(\mathbf{x}) = \sum_{\mathbf{z}:\mu(\mathbf{z})=\mathbf{x}} \tilde{r}(\mathbf{z})$. By the light marginalization assumption, there is a polynomial f such that $|\mathbf{z}| \leq f(|\mu(\mathbf{z})|)$.

To prove support$(\tilde{p}) \in$ NP/poly, we must show that there exists $(M, \boldsymbol{\Theta})$ such that for all $n \geq 0$, a *nondeterministic* Turing machine M_n can be constructed as $M(\boldsymbol{\theta}_n)$ in time $O(\text{poly}(n))$, which can in turn decide in time $O(\text{poly}(n))$ whether $\tilde{p}(\mathbf{x}) > 0$ for any $\mathbf{x}$ with $|\mathbf{x}| = n$.

Deciding $\tilde{p}(\mathbf{x}) > 0$ means deciding whether $(\exists \mathbf{z} \in V^*)\ \mu(\mathbf{z}) = \mathbf{x}$ and $\tilde{r}(\mathbf{z}) > 0$. But if $|\mathbf{x}| = n$, the first condition $\mu(\mathbf{z}) = \mathbf{x}$ implies $|\mathbf{z}| \leq f(|\mu(\mathbf{z})|) = f(|\mathbf{x}|) = f(n)$. Thus, we need M_n to nondeterministically check only the $\mathbf{z}$ of length up to $f(n)$ to see whether $\mu(\mathbf{z}) = \mathbf{x}$ and $\tilde{r}(\mathbf{z}) > 0$.

How can M_n check a string $\mathbf{z}$ of length m? It can decide the first condition $\mu(\mathbf{z}) = \mathbf{x}$ in time $O(\text{poly}(m))$, since the marginalization operator μ is a polytime function. To decide the second condition $\tilde{r}(\mathbf{z}) > 0$, it must construct the (deterministic) Turing machine $M^{\tilde{r}}(\boldsymbol{\theta}^{\tilde{r}}_m)$ and then apply it to $\mathbf{z}$ to obtain $\tilde{r}(\mathbf{z})$: since $\tilde{r}$ is ECCP, both steps take time $O(\text{poly}(m)) = O(\text{poly}(f(n))) \subseteq O(\text{poly}(n))$ as required.

However, this means that $M_n = M(\boldsymbol{\theta}_n)$ must have access to the parameter vectors $\boldsymbol{\theta}^{\tilde{r}}_m$ for all $m \leq f(n)$. We therefore make $\boldsymbol{\theta}_n$ include this collection of parameter vectors. Each $|\boldsymbol{\theta}^{\tilde{r}}_m| \in O(\text{poly}(m)) \subseteq O(\text{poly}(n))$ since $\tilde{r}$ is ECCP. So $|\boldsymbol{\theta}_n| \in O(\text{poly}(n))$ as required. □

Lemma 3.3.4. *For any* $L \in$ NP/poly, *there exists a light marginalization* p *of an ELNCP distribution, such that* support$(p) = L$.

Lemma 3.3.4 resembles Theorem 3.3.1, but it constructs distributions for *all* $L \in$ NP/poly, not just for *one particular* $L \in$ NPC. The proof is similar but more complicated. In both cases, the goal is to demonstrate how an ELNCP distribution r can define a left-to-right stochastic string generation process such that the suffix of the generated string must be in L and can be any element of L.

Our string generation process in this case is inspired by rejection sampling, a widely used method for sampling from an energy-based model with support L. The standard scheme is to first sample a string $\mathbf{x}$ from a tractable distribution q such that support$(q) \supseteq L$, then accept the sample with an appropriate probability, which is 0 if $\mathbf{x} \notin L$. The process is repeated until a sample is finally accepted. There is no guarantee that this standard scheme will terminate in polynomial time, however. Fortunately, in our setting, we are not trying to match our sampling distribution p to a given energy-based model, but simply match its support to a given language L. We make use of the polysize parameter vectors of ELNCP languages to store certain 'fallback strings' that are guaranteed to be in the desired language L. Wherever ordinary rejection sampling would reject a string and try generating another, we switch to generating a stored fallback string of an appropriate length. This scheme places all of the rejected probability mass on the small set of fallback strings (in contrast to rejection sampling, which in effect throws away this mass and renormalizes). The advantage is that it does not iterate indefinitely. At a high level, r is a distribution over

strings $\mathbf{z}$ that record traces of this generative story we describe above.

Proof. WLOG we assume L uses the alphabet $V = \{0, 1, \#\}$. In the case where L is finite, the result is trivial. We simply define $r(\mathbf{x}) = 1/|L|$ for $\mathbf{x} \in L$ and $r(\mathbf{x}) = 0$ otherwise. We then take $p = r$ (a trivial marginalization). It is easy to show that r is ELN, and therefore ELNCP as desired, by constructing an appropriate Turing machine that maps $\hat{\mathbf{x}}x$ to $r(x \mid \hat{\mathbf{x}})$ in time $O(|\hat{\mathbf{x}}x|)$, for any $\hat{\mathbf{x}}$ that is a prefix of some string in L and any $x \in V \cup \{\text{EOS}\}$. The finite state table of the Turing machine includes states that correspond to all possible strings $\hat{\mathbf{x}}x$, with transitions arranged in a trie. It reads the input string $\hat{\mathbf{x}}x$ from left to right to reach the state corresponding to $\hat{\mathbf{x}}x$. If it detects the end of the input while in that state, it writes $r(x \mid \hat{\mathbf{x}})$ on the output tape.

Now we consider the case where L is infinite. For each $j \in \mathbb{N}_{\geq 0}$, let the 'fallback string' $\mathbf{x}^{(j)}$ be some string in L of length $\geq j$. For definiteness, let us take it to be the shortest such string, breaking ties lexicographically. At least one such string does exist because L is infinite, so $\mathbf{x}^{(j)}$ is well-defined.

Also, since $L \in$ NP/poly (§2.2.5.1), let $(M, \boldsymbol{\Theta})$ be an ordered pair and f be a polynomial such that $M_j = M(\theta_j)$ nondeterministically accepts $\mathbf{a}$ within $\leq f(j)$ steps iff $\mathbf{a} \in L$.

As in the proof of Theorem 3.3.1, we now describe a procedure for randomly generating a string $\mathbf{z}$ from left to right. $\mathbf{z}$ will have the form $\mathbf{a}\#\mathbf{b}\#c\mathbf{d}$, where $\mathbf{d} \in L$ and the latent substring $\mathbf{a}\#\mathbf{b}\#c$ will be removed by the marginalization operator μ.

1. First we generate a random string $\mathbf{a} \in \mathbb{B}^*$ followed by #, just as in the proof of Theorem 3.3.1. Again let $j = |\mathbf{a}|$.

2. Next, we must consider whether $\mathbf{a} \in L$. We generate a random computation path $\mathbf{b}$ of M_j on input $\mathbf{a}$ until it either accepts (in which case we then generate `#1` to record acceptance of $\mathbf{a}$) or has run for $f(j)$ steps without accepting (in which case we then generate `#0` to record rejection).

3. In the former case ($c = 1$) we finish by deterministically generating $\mathbf{d} \triangleq \mathbf{a} \in L$. In the latter case ($c = 0$), $\mathbf{a} \notin L$, so we fall back and finish by deterministically generating $\mathbf{d} \triangleq \mathbf{x}^{(j)} \in L$.

Let $r(\mathbf{z})$ be the probability that the above procedure generates $\mathbf{z}$. support(r) is then the set of strings that can be generated by the above procedure. The marginalized language $\mu(\text{support}(r))$ keeps just the $\mathbf{d}$ parts of those strings. It consists of all strings $\mathbf{a}$ that are accepted by at least one path $\mathbf{b}$ of $M_{|\mathbf{a}|}$ (which are exactly the strings in L) together with the fallback strings (which form a subset of L). Thus, $\mu(\text{support}(r)) = L$ as desired.

We wish to show that r is ELNCP. In other words, some Turing machine M^{q} efficiently locally normalizes r with compact parameters $\boldsymbol{\Theta}^{\mathrm{q}}$, as defined in §2.2.4. The parameters will be used to store information about the infinite set of fallback strings.

In particular, for each n, $\boldsymbol{\theta}^{\mathrm{q}}_n$ must have enough information to construct a Turing machine $q_n = M^{\mathrm{q}}(\boldsymbol{\theta}^{\mathrm{q}}_n)$ such that $q_n(\hat{\mathbf{z}}z)$ returns $r(z \mid \hat{\mathbf{z}})$ for all $z \in V \cup \{\textsc{eos}\}$ and all $\hat{\mathbf{z}}$ with $|\hat{\mathbf{z}}| \leq n$ and $Z(\hat{\mathbf{z}}) > 0$. Here $Z(\hat{\mathbf{z}}) > 0$ means that $\hat{\mathbf{z}}$ is a prefix of a string $\mathbf{z} = \mathbf{a}\#\mathbf{b}\#c\mathbf{d}$ that could be generated by the above procedure. The computation $q_n(\hat{\mathbf{z}}z)$ proceeds by simulating the sequence of choices in the above procedure that would be required to generate $\hat{\mathbf{z}}$, and then returning the probability that the procedure would generate symbol z next. That

probability equals $r(z \mid \hat{\mathbf{z}})$ as desired because the above procedure almost surely terminates (as explained at the end of the proof of Theorem 3.3.1).

In general, the computation $q_n(\hat{\mathbf{z}}z)$ may have to construct $M_j = M(\theta_j)$ and simulate it on $\mathbf{a}$ (for $j = |\mathbf{a}|$) if z falls in the $\mathbf{b}\#c$ portion of $\hat{\mathbf{z}}$, and it may have to look up a character of the fallback string $\mathbf{x}^{(j)}\textsc{eos}$ if z falls in the $\mathbf{d}$ portion of $\hat{\mathbf{z}}$ or terminates that portion with $z = \textsc{eos}$. Fortunately $j < n$, and fortunately if the computation looks up the t^{th} character of $\mathbf{x}^{(j)}\textsc{eos}$ then $t < n$. Thus, constructing and simulating M_j can be done in time $O(\text{poly}(j)) \subseteq O(\text{poly}(n))$, and looking up the t^{th} character of $\mathbf{x}^{(j)}\textsc{eos}$ can be achieved with access to the first n characters of each of $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}$, which can be stored by $\boldsymbol{\theta}_n^{\mathrm{q}}$ in space $O(n^2)$. It follows that M^{q} can construct and apply q_n in polynomial time with access to compact parameters $\boldsymbol{\Theta}^{\mathrm{q}}$, so r is ELNCP.

□

3.4 Three expressive parametrizations of sequence model families

Theoretical results from §§3.1–3.3 imply there are three weighted language classes whose expressiveness surpass that of ELNCP — which again is the abstraction of autoregressive parametric sequence model families (with the possibility of the use of a longer parameter vector on harder datasets). In this section, we connect these three weighted language classes to existing sequence model families (that we identify), and discuss the trade-offs

they have to make, for an expressiveness greater than autoregressive sequence models.

3.4.1 Energy-based models (EBMs)

Energy-based models (LeCun et al., 2006) of discrete sequences (Rosenfeld, Chen, and Zhu, 2001; Sandbank, 2008; Huang et al., 2018) traditionally refer to the EC models of §2.2.3. Only the unnormalized probabilities $\tilde{p}_\theta(\mathbf{x})$ are required to be efficiently computable. Lemmas 2.2.3 and 2.2.4 showed that this model family contains all ELN languages and can achieve any support in P. While EBMs are known for their flexible model-specifying mechanisms, in Corollary 5.6.2.1 we formally show that a capacity gap exists between EBMs and autoregressive models (and therefore autoregressive approximations of EBMs (Khalifa, Elsahar, and Dymetman, 2021) in general will be imperfect.) Specifically, Theorem 3.2.2 shows that it also contains languages that are not ELN or even ELNCP: intuitively, the reason is that the sums $Z(\hat{\mathbf{x}})$ needed to compute the local normalizing constants (see §2.2) can be intractable.

If we generalize energy-based sequence models to include all ECCP models — that is, we allow non-uniform computation with compact parameters — then Lemmas 2.2.3 and 2.2.4 guarantee that they can capture all ELNCP languages and furthermore all languages in P/poly (though still not NP-complete languages).

3.4.2 Autoregressive latent-variable sequence models

Theorems 3.3.1 and 3.3.2 make use of unrestricted latent-variable autoregressive models. There exist more practical restricted families of such models that admit tractable computation of $p(\mathbf{x})$ (Lafferty, McCallum, and Pereira, 2001; Rastogi, Cotterell, and Eisner, 2016; Wu, Shapiro, and Cotterell, 2018; Buys and Blunsom, 2018). Such models are EC (and indeed, typically ELN) — but this limits their expressivity, by Theorem 3.2.2. Both Lin et al. (2019) and Buys and Blunsom (2018) observed that such models yield worse empirical results than models that do not have tractable exact inference methods. The tractability requirement is dropped in "self-talk" or "chain-of-thought" (blixt, 2020; Gontier et al., 2020; Shwartz et al., 2020; Wei et al., 2022), where a neural autoregressive language model generates an analysis of the prefix $\hat{\mathbf{x}}$ via latent intermediate symbols before predicting the next output symbol.[15]

We remark that for autoregressive models, the *position* of the latent variables is significant. Marginalizing out latent variables at the *end* of the string adds no power. More precisely, if an ELNCP distribution is over strings $\mathbf{z}$ of the form $\mathbf{x}\#\mathbf{y}$, then its marginalization via $\mu(\mathbf{x}\#\mathbf{y}) = \mathbf{x}$ can be expressed more simply as an ELNCP language. Thus, by Theorem 3.2.3, marginalizations of such distributions cannot have arbitrary NP languages as support. Our proofs of Theorems 3.3.1 and 3.3.2 instead use latent strings of the form $\mathbf{y}\#\mathbf{x}$, where all latent variables precede all observed ones (as in Kingma and Welling, 2014). (This simple

[15] Here the marginal distribution of the next observed symbol can require superpolynomial time to compute (if #P $\neq$ FP, which follows from NP $\not\subseteq$ P/poly). Theorem 3.2.2 could likewise be evaded by *other* autoregressive approaches that invest superpolynomial computation in predicting the next symbol (Graves, 2016). Each autoregressive step might explicitly invoke lookahead or reasoning algorithms, just as feed-forward network layers can invoke optimizers or solvers (Amos and Kolter, 2017; Wang et al., 2019).

design can always be used without loss of generality.) Trying to reorder those latent strings as $\mathbf{x}\#\mathbf{y}$ while preserving their weights would have yielded a non-ELNCP distribution $p(\mathbf{x}\#\mathbf{y})$ (because if it were ELNCP, then $p(\mathbf{x})$ would be ELNCP also, and we know from Lemma 2.2.4 that it cannot be for any distribution whose support is an NP-complete language).

How about lightly marginalizing ECCP languages instead of ELNCP ones? This cannot model any additional *unweighted* languages, by Theorem 3.3.2. But it may be able to model more probability distributions. One can easily construct a light marginalization p of an ECCP distribution such that $\#(\phi) = c_n \cdot p(\boldsymbol{\phi})$, where $\#(\phi)$ is the number of satisfying assignments of ϕ and the constant c_n depends only on $n = |\boldsymbol{\phi}|$. We conjecture that this is not possible with lightly marginalized ELNCP distributions.

3.4.3 Lookup models

§2.2.3.1 noted that with exponential growth in stored parameters, it is possible to fit any weighted language up to length n, with local probabilities computed in only $O(n)$ time by lookup. Of course this rapidly becomes impractical as n increases, even if the amount of training data increases accordingly. However, there has been some recent movement toward storage-heavy models. Such models are typically semiparametric: they use a parametric neural model, such as an autoregressive model, together with an external knowledge base of text strings or factoids that are not memorized in the layer weights. The neural model generates queries against the knowledge base and combines their results. Examples include kNNLMs (Khandelwal et al., 2020), semiparametric LMs (Yogatama, Masson d'Autume, and

Kong, 2021), and retrieval-augmented generation models (Lewis et al., 2020). The knowledge base grows linearly with the training data rather than compressing the data into a smaller parameter vector. It is in fact a copy of the training data, indexed to allow fast lookup (Indyk and Motwani, 1998). (Preparing the index is much cheaper than neural network training.) Access to the large knowledge base may reduce the amount of computation needed to find the local conditional probabilities, much as in the trie construction of §2.2.3.1.

3.5 Related work

Chen et al. (2018b) show that it is hard to map *RNN parameters* to properties of the resulting autoregressive weighted language, such as consistency ($Z = 1$). We focus on cases where the RNN parameters are already known to be consistent, so the RNN efficiently maps a *string* $\hat{\mathbf{x}}$ to its local conditional distribution $p(\cdot \mid \hat{\mathbf{x}})$. Our point is that for some weighted languages, this is not possible (even allowing polynomially larger RNNs for longer strings), so consistent RNNs and their ilk cannot be used to describe such languages.

In a Bayes network — which is really just an autoregressive model of fixed-length strings — approximate marginal inference is NP-hard (Roth, 1996). Assuming NP $\not\subseteq$ P/poly and the grid-minor hypothesis, Chandrasekaran, Srebro, and Harsha (2008, Theorem 5.6) further showed that for any infinite sequence of graphs $G_1, G_2, \ldots$ where G_n has treewidth n, there is no sequence of algorithms $M_1, M_2, \ldots$ such that M_n performs approximate marginal inference in time $O(\text{poly}(n))$ on graphical models of structure G_n. This remarkable negative

result says that in *any* graph sequence of unbounded treewidth, approximating the normalizing constant for G_n given arbitrary parameters is hard (not $O(\text{poly}(n))$), even with advice strings. Our negative result (Theorem 3.2.6) focuses on *one particular* infinite weighted language, showing that approximating local conditional probabilities given an arbitrary length-n prefix is hard in the same way. (So this language cannot be captured by an RNN, even with advice strings.)

While we only show superpolynomiality results under the assumption NP $\not\subseteq$ P/poly, both Chandrasekaran, Srebro, and Harsha (2008) and Kwisthout, Bodlaender, and Gaag (2010) were able to further sharpen their hardness results under the stronger exponential time hypothesis (ETH) (Impagliazzo and Paturi, 1999). We leave possible sharpening of our results with the consideration of ETH as future work.

3.6 Conclusion

Autoregressive models are suited to those probability distributions whose prefix probabilities are efficiently computable. This efficiency is convenient for training and sampling. But unless we sacrifice it and allow runtime or parameter size to grow superpolynomially in input length, autoregressive models are less expressive than models whose prefix probabilities expensively marginalize over suffixes or latent variables.

All model families we have discussed in this chapter can be seen as making compromises between different desiderata (Table 3.1). Natural follow-up questions include *'Are there*

model families that win on all fronts?' 'What are other modeling desiderata?'

While *some* languages $\in$ P cannot be supports of ELNCPs, we do not know if the same can be said for *most* languages $\in$ P. This problem seems to be closely related to the average complexity of NP-complete languages, where most questions remain open (Levin, 1986; Bogdanov and Trevisan, 2006).

Chapter 4

Residual energy-based sequence models

In this chapter, we describe a particular design of energy-based sequence models: **residual energy-based sequence models** (REBMs). While in § 3 we argue that energy-based sequence models are much more expressive than autoregressive sequence models (Corollary 5.6.2.1 and Theorem 3.2.2), they are in general intractable, unless we limit their expressiveness, and/or the maximum string length that we seek to model.[1] We empirically demonstrate that with both restrictions imposed, energy-based sequence models do provide marginal, yet statistically significant improvement over autoregressive baselines over several datasets, and on two different neural architectures.

[1] We discuss these restrictions in further detail in § 5.

4.1 Design of the REBM architecture

Residual energy-based models (**REBMs**) (Bakhtin et al., 2021)[2] are a simple hybrid architecture:

$$p_\theta(\mathbf{x}) \propto \tilde{p}_\theta(\mathbf{x}) \triangleq p_0(\mathbf{x}) \cdot \exp g_\theta(\mathbf{x})$$

This simply multiplies our previous weight by a new factor $p_0(\mathbf{x})$. The *base model* $p_0 : \mathcal{X} \longrightarrow (0, 1]$ is a locally normalized neural sequence model (ELN model) that was pretrained on the same distribution. $g_\theta : V^* \longrightarrow \mathbb{R}$ is a learnable function (with parameters θ) that is used to adjust p_0, yielding a weighted language $\tilde{p}_\theta$ with the same support $\mathcal{X}$.

4.1.1 Modeling finite subsets of infinite languages

The experiments of this chapter are conducted on datasets where we only observe strings that are finitely long. Given a possibly infinite language $\mathcal{X}$, we use the notation $\mathcal{X}_{\leq T} = \{\mathbf{x} : \mathbf{x} \in \mathcal{X}, |\mathbf{x}| \leq T\}$ for the subset of strings that are most T symbols long. Our learned sequence model will have a support $\subseteq \mathcal{X}_{\leq T}$. Specific values of T for datasets used in our experiments are listed in §4.3.1.

[2]We independently conceived of and implemented the REBM idea proposed in Bakhtin et al. (2021). Details of neural architecture choice, model parameter sizes, training regimen, and evaluation differ between our work and theirs, which also reported positive empirical results (on different datasets). We regard the two independent positive findings as a strong indication that the REBM design is effective.

4.1.2 Design of base models p_0

p_0 can be any distribution over $\mathcal{X}_{\leq T}$[3] provided that we can sample from it, and evaluate $p_0(\mathbf{x}), \forall \mathbf{x} \in \mathcal{X}_{\leq T}$, both in $O(\text{poly}(|\mathbf{x}|))$. In this work, we experiment with two designs of p_0: GRU- and Transformer-based locally normalized language models. GRU-based models are used in WikiText and Yelp experiments.

The GRU-based p_0's are parametrized with 2-layer GRUs with 500 hidden units, and word embeddings of dimension size 500.

As for Transformer-based p_0's, we make use of Grover models (Zellers et al., 2019), which effectively are GPT-2 models (Radford et al., 2019) trained on the aforementioned RealNews dataset. In this work, we experiment with the 'base' variant of public available weights, which are 12-layered Transformers, with 12 heads, and 768 hidden units.

4.1.3 Design of g_θ

We formulate $g_\theta(\mathbf{x})$ as a summation of scores at positions $1 \dots |\mathbf{x}|$, passed through an activation function f:

$$g_\theta(\mathbf{x}) = f\left(\sum_{i=1}^{|\mathbf{x}|} g_t(\mathbf{x}; \theta)\right). \tag{4.1}$$

[3]Note that since p_0 does not have support over $\mathcal{X}$, it has to assign $p(\text{EOS} \mid \mathbf{x}_{1\dots T}) = 1$, which is generally not an issue.

To ensure learnability, we experiment with two variants of f, both of which have constrained ranges:

- **tanh**: $f(x) = 2 \cdot \tanh(x)$
- **softplus**: $f(x) = -\log(1 + \exp(x + s))$

The former one is bounded between $(-2, 2)$, while the second one has range $(-\infty, 0)$. The offset term s in the softplus activation function determines initial values of Z_θ. In this chapter we set $s = 20$.

The architecture design of $g_t(\mathbf{x}; \boldsymbol{\theta})$ follows their base model counterparts: we implement $g_t(\mathbf{x}; \boldsymbol{\theta})$ as Bi-GRU for GRU base models; and bi-directional Transformer discriminators for Transformer ones. In both cases, $g_t(\mathbf{x}; \boldsymbol{\theta})$ is the score of x_t in context.

4.1.4 Training procedure

Maximum-likelihood parameter estimation (MLE) can be expensive in an EBM because the likelihood formula involves the expensive summation $Z = \sum_{\mathbf{x} \in V^*} \tilde{p}_\theta(\mathbf{x})$, even when Z is computable. We therefore opt to train our model using the ranking variant of noise contrastive estimation (NCE) (Ma and Collins, 2018), which does not require samples from p_0 and has a simple form for residual LMs. Using p_0 as a *noise distribution*, NCE training requires minimizing the following single-sequence loss, in expectation over the

true distribution p, where $\mathbf{x}$ is drawn:

$$\mathcal{L}_{\text{NCE}}(\boldsymbol{\theta}, \mathbf{x}, p_0, K) = -\log \frac{\frac{\tilde{p}_\theta}{p_0}(\mathbf{x})}{\sum_{k=0}^{K} \frac{\tilde{p}_\theta}{p_0}(\mathbf{x}^{(k)})}, \tag{4.2}$$

where $\mathbf{x}^{(0)} \triangleq \mathbf{x}$, $\frac{\tilde{p}_\theta}{p_0}(\mathbf{x}) \triangleq \frac{\tilde{p}_\theta(\mathbf{x})}{p_0(\mathbf{x})}$, and $\mathbf{x}^{(1)} \dots \mathbf{x}^{(K)} \sim p_0$. Since $\tilde{p}_\theta(\mathbf{x}) = p_0(\mathbf{x}) \cdot \exp g_\theta(\mathbf{x})$, we have $\frac{\tilde{p}_\theta}{p_0}(\mathbf{x}) = \exp g_\theta(\mathbf{x})$. The NCE minimization objective (4.2) now reduces to the simple form

$$\mathcal{L}_{\text{NCE}}(\boldsymbol{\theta}, \mathbf{x}, p_0, K) = -g_\theta(\mathbf{x}) + \log(\exp g_\theta(\mathbf{x}) + \sum_{k=1}^{K} \exp g_\theta(\mathbf{x}^{(k)})). \tag{4.3}$$

Notice that minimizing the expected loss with stochastic gradient descent methods $\mathcal{L}_{\text{NCE}}$ defined in equation (4.3) requires only evaluating sequence probabilities under g_θ, and tuning its parameters, but not the base model p_0. We can just draw the noise samples from p_0. This way we do not need to backpropagate through parameters of the base model p_0, which can speed up training considerably when p_0 is backed by a huge network. In fact, the training of g_θ can be completely agnostic to the design of p_0, allowing for the application of finetuning any locally normalized p_0.

Given the same discriminator g_θ, the difference of KL-divergence between the true model p and residual language models $\tilde{p}'_\theta(\mathbf{x}) = p'_0(\mathbf{x}) \cdot \exp g_\theta(\mathbf{x})$, and the KL-divergence between the true model and $\tilde{p}''_\theta(\mathbf{x}) = p''_0(\mathbf{x}) \cdot \exp g_\theta(\mathbf{x})$, defined with base models p'_0 and p''_0 respectively, can be written as

$$\text{KL}[p||p'_\theta] - \text{KL}[p||p''_\theta] = \text{KL}[p||p'_0] - \text{KL}[p||p''_0] + \log \frac{Z'}{Z''}, \tag{4.4}$$

where $Z' = \mathbb{E}_{\mathbf{x}\sim p_0'}[\exp g_\theta(\mathbf{x})]$, and Z'' is similarly defined with p_0''. As a direct result of equation (4.4), we can see that finding p_0'' where $\mathrm{KL}[p||p_0''] < \mathrm{KL}[p||p_0']$ implies improvement in $\mathrm{KL}[p||p_\theta'']$ over $\mathrm{KL}[p||p_\theta']$, under some conditions:

Proposition 4.1.1. *If* $\exists k > 0$ *such that* $\frac{\mathbb{E}_{\mathbf{x}\sim p_0'}[\exp g_\theta(\mathbf{x})]}{\mathbb{E}_{\mathbf{x}\sim p_0''}[\exp g_\theta(\mathbf{x})]} > \exp(-k)$ *and* $\mathrm{KL}[p||p_0'] - \mathrm{KL}[p||p_0''] > k$ *then* $\mathrm{KL}[p||p_\theta'] > \mathrm{KL}[p||p_\theta'']$.

Proof.

$$
\begin{aligned}
&\mathrm{KL}[p||p_\theta'] - \mathrm{KL}[p||p_\theta''] \\
&= \mathop{\mathbb{E}}_{\mathbf{x}\sim p}[\log p_\theta''(\mathbf{x}) - \log p_\theta'(\mathbf{x})] \\
&= \mathop{\mathbb{E}}_{\mathbf{x}\sim p}[\log \frac{p_0''(\mathbf{x}) \exp g_\theta(\mathbf{x})}{\sum_{\mathbf{x}'\in\mathcal{X}_{\leq T}} p_0''(\mathbf{x}) \exp g_\theta(\mathbf{x})} \\
&- \log \frac{p_0'(\mathbf{x}) \exp g_\theta(\mathbf{x})}{\sum_{\mathbf{x}'\in\mathcal{X}_{\leq T}} p_0'(\mathbf{x}) \exp g_\theta(\mathbf{x})}] \\
&= \mathop{\mathbb{E}}_{\mathbf{x}\sim p}[\log \frac{p_0''(\mathbf{x}) \exp g_\theta(\mathbf{x})}{\mathbb{E}_{\mathbf{x}'\sim p_0''}[\exp g_\theta(\mathbf{x})]} \\
&- \log \frac{p_0'(\mathbf{x}) \exp g_\theta(\mathbf{x})}{\mathbb{E}_{\mathbf{x}'\sim p_0'}[\exp g_\theta(\mathbf{x})]}] \\
&= \mathop{\mathbb{E}}_{\mathbf{x}\sim p}[\log p_0''(\mathbf{x}) - \log p_0'(\mathbf{x})] \\
&+ \mathop{\mathbb{E}}_{\mathbf{x}\sim p}[\log \mathop{\mathbb{E}}_{\mathbf{x}'\sim p_0'}[\exp g_\theta(\mathbf{x})] - \log \mathop{\mathbb{E}}_{\mathbf{x}'\sim p_0''}[\exp g_\theta(\mathbf{x})]] \\
&= \mathrm{KL}[p||p_0'] - \mathrm{KL}[p||p_0''] \\
&+ \log \frac{\mathbb{E}_{\mathbf{x}'\sim p_0'}[\exp g_\theta(\mathbf{x})]}{\mathbb{E}_{\mathbf{x}'\sim p_0''}[\exp g_\theta(\mathbf{x})]}.
\end{aligned} \tag{4.5}
$$

Plugging assumptions $\frac{\mathbb{E}_{\mathbf{x}\sim p_0'}[\exp g_\theta(\mathbf{x})]}{\mathbb{E}_{\mathbf{x}\sim p_0''}[\exp g_\theta(\mathbf{x})]} > \exp(-k)$ and $\mathrm{KL}[p||p_0'] - \mathrm{KL}[p||p_0''] > k$ into equa-

tion (4.5), $\mathrm{KL}[p||p'_\theta] - \mathrm{KL}[p||p''_\theta] > 0$. □

Proposition 4.1.1 suggests a training heuristics where we first train the base model p_0, then finetune g_θ: under a roughly uniform g_θ (*e.g.* when θ is newly initialized), $\mathbb{E}_{\mathbf{x}\sim p'_0}[\exp g_\theta] / \mathbb{E}_{\mathbf{x}\sim p''_0}[\exp g_\theta] \approx \exp(0)$; so improvements on the inclusive KL-divergence of base model $\mathrm{KL}[p||p_0]$ will mostly translate to improvement in $\mathrm{KL}[p||\tilde{p}_\theta]$. Optimizing the base model (*i.e.* finding p''_0 such that $\mathrm{KL}[p||p''_0] < \mathrm{KL}[p||p'_0]$) is much easier than directly minimizing $\mathrm{KL}[p||p'_\theta]$: the former can be done by minimizing empirical cross entropy, which is computationally efficient, while the latter involves an intractable partition function $\sum_{\mathbf{x}\in\mathcal{X}_{\leq T}} \tilde{p}'_\theta(\mathbf{x})$.

Pseudocode for fine-tuning g_θ is listed in Algorithm 1.

Algorithm 1: Pseudocode for training g_θ

Input:

- Training/validation corpora $\mathcal{D}_{\{\text{train,dev}\}}$
- base model $p_0 : \mathcal{X}_{\leq T} \longrightarrow [0, 1]$
- initial parameter vector $\theta_0 \in \mathbb{B}^d$
- noise sample size $K \in \mathbb{N}$

Output: unnormalized residual language model $\tilde{q}_\theta : \mathcal{X}_{\leq T} \longrightarrow [0, 1]$

$\theta \leftarrow \theta_0$;

`/* ` $\mathcal{L}_{\text{NCE}}$ `is defined in equation (4.3) */`

while $\sum_{\mathbf{x}\in\mathcal{D}_{dev}} \mathcal{L}_{NCE}(\theta, \mathbf{x}, p_0, K)$ *is still decreasing* **do**

  foreach $\mathbf{x} \in$ *shuffle*$(\mathcal{D}_{train})$ **do**

    $\nabla_\theta \mathcal{L}_{\text{NCE}} = \nabla_\theta \mathcal{L}_{\text{NCE}}(\theta, \mathbf{x}, p_0, K)$;

    $\theta \leftarrow$ update-gradient$(\theta, \nabla_\theta \mathcal{L}_{\text{NCE}})$;

  end

end

return $\mathbf{x} \mapsto p_0(\mathbf{x}) + \exp g_\theta(\mathbf{x})$;

4.1.5 Computing normalized probabilities

The unnormalized probability $\tilde{p}_\theta(\mathbf{x})$ (in equation (4.1)) can be evaluated easily, and should suffice for (re)ranking purposes (*e.g.* for ASR and MT applications). However, the *normalized* probability $q_\theta(\mathbf{x}) \triangleq \frac{\tilde{p}_\theta(\mathbf{x})}{\sum_{\mathbf{x}} \tilde{p}_\theta(\mathbf{x})}$ does require computing the partition function Z_θ. An unbiased importance sampling estimate of $\sum_{\mathbf{x} \in \mathcal{X}_{\leq T}} \tilde{p}_\theta(\mathbf{x})$ is

$$\begin{aligned} Z_\theta &= \sum_{\mathbf{x} \in \mathcal{X}_{\leq T}} \tilde{p}_\theta(\mathbf{x}) \\ &= \sum_{\mathbf{x} \in \mathcal{X}_{\leq T}} p_0(\mathbf{x}) \exp g_\theta(\mathbf{x}) \\ &= \mathop{\mathbb{E}}_{\mathbf{x} \sim p_0} [\exp g_\theta(\mathbf{x})] \\ &\approx \sum_{m=1}^{M} \frac{\exp g_\theta(\mathbf{x}^{(m)})}{M}, \end{aligned} \tag{4.6}$$

where $\mathbf{x}^{(1)} \ldots \mathbf{x}^{(M)} \sim p_0$.

4.2 Comparison between REBMs and autoregressive models

We evaluate the effectiveness of REBMs on two different neural architectures (GRU- and Transformer-based) and 3 datasets: WikiText (Merity et al., 2017), Yelp (*Yelp Open Dataset*), and RealNews (Zellers et al., 2019), on the task of modeling sequence probabilities. An

Experiment (Architecture)	Model	Best configuration (according to devset)	log likelihood improvement (95% CI)	perplexity improvement
RealNews (Transformer)	p_θ	4-layer, tanh	$(-0.18, -0.13), \mu = -0.15$	.03%
RealNews (Transformer)	p_0'	N/A	N/A	.00%
WikiText (GRU)	p_θ	1-layer/500, softplus	$(-1.85, -1.54), \mu = -1.69$	1.44%
WikiText (GRU)	p_0'	N/A	N/A	.50%
Yelp (GRU)	p_θ	2-layer/500, softplus	$(-1.89, -1.67), \mu = -1.80$	1.82%
Yelp (GRU)	p_0'	N/A	N/A	.49%

Table 4.1: Residual energy-based model $\tilde{p}_\theta$ improvements over autoregressive base models p_0. The perplexity numbers are per-token, and log likelihood improvements are per sequence (in nats).

REBM $\tilde{p}_\theta$ has two components, g_θ and p_0, and we would like to see how $\tilde{p}_\theta$ competes against p_0 itself. We do not further tune p_0 while training p_θ. As a fair comparison, we also see how p_0' compares against p_0, where p_0' is simply a version of p_0 that has been trained as many additional epochs as were used to train p_θ.

p_0 models are pretrained on moderately large corpora (in GRU cases) or a very large corpus (in the Transformer case).[4] We compare residual energy-based models $\tilde{p}_\theta$ to further-fine-tuned base models p_0', on conservatively estimated (*i.e.*, the low end of 95% confidence interval) token perplexity and bootstrap-sampled log likelihood improvements. The results are in Table 4.1.[5] Residual energy-based models show consistent perplexity improvement compared to p_0' that are trained on the same data using the same maximum numbers of iterations. Although the improvement in log-likelihood of p_θ over p_0 is modest (especially for RealNews experiments, where p_0 is a very strong baseline), we verify that these

[4] In the Transformer case we simply take p_0 to be the Grover (Zellers et al., 2019) pretrained language model, which is based on the GPT-2 (Radford et al., 2019) architecture and performs competitively on news article generation.

[5] We only report each dataset's best model (according to validation data) in this table. See §4.3 for experimental details.

improvements are all statistically significant ($p < 0.05$) using bootstrapped test datasets.

We experiment with different designs of the discriminator g_θ, evaluating the effectiveness of bounding g_θ (§4.1.3) and varying its number of parameters. We find that in Transformer-based experiments, bounding g_θ considerably helps with performance; but the opposite happens for GRU-based models. We speculate that this is due to the base models' performance: the Transformer base models have high parameter count and were trained on a lot of data; and the true distribution p likely is relatively similar to p_0, and benefits from a small hypothesis space. On the other hand our GRU-based p_0 has neither the capacity, nor the huge amount of training data. As a result, the unbounded variant g_θ (and q_θ) may end up learning a better approximation of p.

4.3 Experimental details

4.3.1 Datasets

Residual language model experiments are conducted on these datasets:

- **Segmented WikiText**: we take the standard WikiText-2 corpus (Merity et al., 2017), and segment it into sequences at new line breaks, followed by BPE tokenization. We discard all empty lines, and any line that starts with the '=' token (which signifies a header line). In effect, we obtain sequences that are mostly entire paragraphs. We also only keep lines that are shorter than 800 tokens. Because of our preprocessing,

Segmented WikiText loses much interparagraph context information, and doesn't have the 'simple' header sequences that were in the original WikiText corpus, and is much harder to language-model.

- **Yelp**: the Yelp dataset (*Yelp Open Dataset*) contains business reviews. Each sequence is a review. As in Segmented WikiText, we keep only reviews shorter than 800 tokens.

- **RealNews**: we make use of the standard RealNews corpus comes from (Zellers et al., 2019), which contains news articles that are up to 1,024 tokens long.

In all experiments we tokenize with BPE tokenizers derived from the GPT-2 language models: the GRU models use Huggingface's implementation[6] and the Transformers use Grover's[7]. Number of sequences in preprocessed datasets are listed in Table 4.2.

	Train	Dev	Test
RealNews	3,855	1,533	6,158
WikiText	18,519	878	2,183
Yelp	10,951	9,964	994

Table 4.2: Number of sequences in preprocessed datasets (for training and tuning the discriminators g_θ, and evaluation).

4.3.2 Pretraining base models p_0

We use a pretrained Grover model as the base model in RealNews experiments. For GRU-based experiments, we train base models on WikiText and Yelp datasets using separate

[6] https://github.com/huggingface/transformers
[7] https://github.com/rowanz/grover

training and validation splits than those of the discriminator g_θ (Table 4.3). The base models are periodically (every $1,000$ iterations) evaluated on the validation split for early stopping (with patience equal to 10 epochs). The base models q_θ achieve 113.98 for Segmented WikiText, and 110.89 in test set perplexity, respectively. Note that these base models are further fine-tuned on additional datasets in our comparison against residual language models.

	Train	Dev
WikiText	$17,556$	$1,841$
Yelp	$9,954$	$1,000$

Table 4.3: Number of sequences in preprocessed datasets (for training and tuning the base model q). Note that we do not train our own base models for RealNews, but use one of the pretrained models provided by (Zellers et al., 2019).

4.3.3 Metrics

We evaluate the relative performance of residual language models against autoregressive models (*i.e.* fine-tuned base models) on two metrics, log likelihood and perplexity improvement, which are approximated as follows:

- **Approximate** log **likelihood improvement**: since p, p_θ and q_0 are all distributions over $\mathcal{X}_{\leq T}$, we can quantitatively evaluate their difference in log likelihood. We measure

the difference between $\mathrm{KL}[p||p_\theta]$ and $\mathrm{KL}[p||p_0]$:[8]

$$\begin{aligned}
&\mathrm{KL}[p||p_\theta] - \mathrm{KL}[p||p_0] \\
&= \mathop{\mathbb{E}}_{\mathbf{x}\sim p}[\log p_\theta(\mathbf{x}) - \log p_0(\mathbf{x})] \\
&= \mathop{\mathbb{E}}_{\mathbf{x}\sim p}[\log \tilde{p}_\theta(\mathbf{x}) - \log p_0(\mathbf{x})] - \log Z_\theta \\
&= \mathop{\mathbb{E}}_{\mathbf{x}\sim p}[g_\theta(\mathbf{x})] - \log Z_\theta \\
&\approx \frac{\sum_{\mathbf{x}\in\mathcal{D}_{\text{test}}} g_\theta(\mathbf{x})}{|\mathcal{D}_{\text{test}}|} - \log \hat{Z}_{\theta M},
\end{aligned} \tag{4.7}$$

where $\hat{Z}_{\theta M}$ is estimated using equation (4.6). A negative value of log likelihood difference indicates that $\tilde{q}_\theta$ approximates p better than p_0 in terms of KL-divergence.

- **Perplexity improvement**: perplexity is a common language modeling metric. Following (Rosenfeld, Chen, and Zhu, 2001), we compute

$$\begin{aligned}
&\text{perplexity improvement of } p_\theta \\
&= \frac{\exp \frac{|\mathcal{D}| \log \hat{Z}_{\theta M}}{w(\mathcal{D}_{\text{test}})}}{\exp \frac{\sum_{\mathbf{x}\in\mathcal{D}_{\text{test}}} g_\theta(\mathbf{x})}{w(\mathcal{D}_{\text{test}})}},
\end{aligned} \tag{4.8}$$

where $w(\mathcal{D})$ is the total token count of dataset $\mathcal{D}$, and $|\mathcal{D}|$ is the number of sequences of $\mathcal{D}$. $\hat{Z}_{\theta M}$ is ecomputed §4.1.5

Both evaluation metrics involve estimating the partition function with $\hat{Z}_{\theta M}$. For the per-

[8]Note that p_0 here is the base model component of $\tilde{p}_\theta$. While comparing between residual language models and autoregressive models, we also finetune p_0 on additional data to get a new model q_0', which has different parameters than p_0.

plexity improvement metric, we obtain 32 estimates of $\hat{Z}_{\theta M}$, which are normally distributed, and compute equation (4.8) using $\hat{Z}_{\theta M}$ the conservative end of a 95% confidence level. We set $M = 512$ in this work. To account for variance in our test datasets, we further make use of bootstrapping estimation for log likelihood improvement: we bootstrap-sample $1,000$ subsamples for each test dataset, and compute equation (4.7) for each datapoint in the Cartesian product ($1,000 \times 32$ in total). We then report results at the 2.5% and 97.5% percentiles.

4.3.4 Hyperparameters

Transformer experiments. We train our models on 64 GPUs across 8 nodes, with a total batch size of $64 \times 8 \times 2 = 1,024$, and with 1 noise sequence ($K = 1$ in §4.1.4) per batch. We use an initial learning rate of $5e-5$. The rest of the hyperparameters largely follow settings in (Zellers et al., 2019). Optimization is done with the Grover implementation of AdaFactor.

GRU experiments. We train our models on 8 GPUs on a single node, with a total batch size of $8 \times 2 = 16$, and with 25 noise sequences ($K = 25$ in §4.1.4) per batch. We have an initial learning rate of $1e-4$. Upon no improvement on validation data, we half the learning rate, with patience = 1. The model parameters are l_2 regularized with a coefficient of $1e-5$. We also apply dropout regularization with $p = 0.5$. Optimization is done with PyTorch-supplied Adam.

4.3.5 Configurations

We study the effects of these configurations:

- **Bounding** g_θ: we note in §4.1.3 that with the strong hypothesis that the base model p_0 has bounded error, g_θ will have a bounded range, and leads to a much smaller hypothesis space. In this work we experiment with both bounded and unbounded g_θ's, with ranges $(-\infty, 0)$ and $(-2, 2)$ respectively. More details can be found in §4.1.3.

- **Model capability of** g_θ: we hypothesize that the expressiveness of g_θ does not need to be as rich as the parametrization of p_0, since g_θ essentially only has to tell whether the sequence $\mathbf{x}$ comes from p or p_0. For the GRU + WikiText experiments, we experiment with $\{1, 2\}$-layer GRU models of g_θ. For 1-layer models, we additionally experiment with a setup that has only 250 hidden units. For the Transformers/RealNews dataset, we experiment with $\{12, 4\}$-layer Transformer models.

4.3.6 Log likelihood improvements under different configurations

We also see in Table 4.4 that using tanh as the activation function f does better than softplus for Transformers; but performs very poorly for GRUs. We also observe degeneracy problems. We speculate that our Transformer-based base models q_θ have already learned a good approximation of the true distribution; and limiting the model capacity of g_θ in

Model Size	Activation	log likelihood improvement	
		95% CI	μ
RealNews (Transformers)			
12-layer	softplus	$(-0.13, 0.08)$	-0.09
12-layer	tanh	$(-0.14, -0.10)$	-0.12
4-layer	softplus	$(-0.15, 2.62)$	-0.02
4-layer	tanh	$(-0.18, -0.13)$	-0.16
WikiText (GRUs)			
2-layer / 500	tanh	$(-0.00, 0.00)$	-0.00
2-layer / 500	softplus	$(-1.32, -0.85)$	-1.18
1-layer / 500	tanh	$(-0.79, -0.64)$	-0.71
1-layer / 500	softplus	$(-1.85, -1.54)$	-1.69
1-layer / 250	tanh	$(-0.02, 0.02)$	-0.00
1-layer / 250	softplus	$(-1.85, -1.46)$	-1.67
Yelp (GRUs)			
2-layer / 500	tanh	$(-0.03, 0.01)$	-0.02
2-layer / 500	softplus	$(-1.89, -1.67)$	-1.80
1-layer / 500	tanh	$(-0.65, -0.57)$	-0.61
1-layer / 500	softplus	$(-2.62, -2.03)$	-2.43
1-layer / 250	tanh	$(-0.00, 0.00)$	-0.00
1-layer / 250	softplus	$(-2.25, -1.99)$	-2.13

Table 4.4: Comparison of different configurations.

exchange of smaller variance results in a favorable trade-off, and vice versa for GRUs. Regarding discriminator capability: we see that performance is not sensitive to model size. Our best Transformers run actually is from the smaller-model runs. And the 1-layer 500-unit GRU models achieve best performance. Overall, results in Table 4.4 suggests that performance is sensitive to the choice of model configuration.

4.4 Conclusion

In this chapter, we have experimented with a particular family of energy-based sequence models — REBMs — and showed their effectiveness over two different neural architectures and 3 datasets. Our improvements over the autoregressive baseline models are statistically significant, yet marginal.

A major limitation of our REBMs is that they define *finite*-support distributions over strings with maximum length $T \in \mathbb{N}$ (§4.1.1). With a predefined T that is large enough we may handle paragraph- or document-level language modeling well. However for some applications it is difficult to set T beforehand: for example, a path in a finite-state machine can go through a cycle arbitrarily many times. If we were to model a path distribution using an REBM, we effectively disallow cycles.

So, can we relax the constraint that the REBM distribution has finite support? In § 5, we will see that the answer is unfortunately no, unless we are willing to cripple model expressiveness in significant ways.

Chapter 5

Uncomputability and inapproximability of the partition functions of EC languages

One of the conclusions of § 3 is that energy-based sequence models are strictly more expressive than autoregressive sequence models. As Theorems 3.1.1, 3.2.2, 3.2.3 and 3.2.6 and Lemmas 3.2.4 and 3.2.7 imply, there are distributions which cannot be captured by any autoregressive sequence models, while being model-able by energy-based sequence models.

Because of expressiveness concerns, it may be desirable to parametrize energy-based sequence models as neural networks, to ensure that they can capture interaction between symbols over an arbitrarily long distance (*e.g.*, as required to model the formal language we construct in the proof of Theorem 3.2.2).

CHAPTER 5. UNCOMPUTABILITY AND INAPPROXIMABILITY OF THE PARTITION FUNCTIONS OF EC LANGUAGES

Contemporary neural networks have been shown to be very powerful. In particular, some popular parametric sequence model families (§2.3), such as RNNs (Siegelmann and Sontag, 1995) and Transformers (Bhattamishra, Patel, and Goyal, 2020; Pérez, Barceló, and Marinkovic, 2021), have been formally shown to recognize recursively enumerable languages: given *any* Turing machine M, there is a parameter vector $\boldsymbol{\theta} \in \boldsymbol{\Theta}$, where $\boldsymbol{\Theta}$ is a parametric sequence model family (§2.3), such that the parametrized sequence model $N_{\boldsymbol{\theta}}$ recognizes the same language as M does.[1] In other words, these sequence model families are **Turing-complete**.

It is therefore intuitive that energy-based sequence models, backed by such powerful neural networks, can model any weighted language $\in$ EC. That is, these parametric sequence model families are EC-complete (§2.3.2). And indeed, we have shown that a particular class of Transformers are EC-complete (Theorem 2.3.2). It would seem assuring to work with such model families: assuming there exists an algorithm that computes true string probabilities in polytime,[2] the model family $\boldsymbol{\Theta}$ is well-specified — that is, there exists $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ such that $N_{\boldsymbol{\theta}}$ defines the true weighted language. Moreover, one may assume that given we can sample from $p_{\boldsymbol{\theta}_M}$, we would be able to use consistent estimators to find $\boldsymbol{\theta}' \in \boldsymbol{\Theta}$, where $\boldsymbol{\theta}' \approx \boldsymbol{\theta}_M$.

Unfortunately, we find that with such an expressive distribution family $\boldsymbol{\Theta}$ (*e.g.*, when $\boldsymbol{\Theta}$ is EC-complete), whether the identifiability assumption holds — required for most consistent

[1] We refer the interested reader to Pérez, Barceló, and Marinkovic (2021) for technical details.

[2] By definition, string weights $\tilde{p}(\boldsymbol{\omega})$ in an EC weighted language can be computed in $O(\text{poly}(|\boldsymbol{\omega}|))$. But those are not necessarily normalized probabilities $p(\boldsymbol{\omega})$. Therefore, the algorithm that computes $p(\boldsymbol{\omega})$ may need the normalizing constant $Z = \sum \tilde{p}(\boldsymbol{\omega})$ memorized, in order to compute $p(\boldsymbol{\omega})$ in $O(\text{poly}(|\boldsymbol{\omega}|))$.

estimators (Lehmann and Casella, 2006) — itself is **undecidable** (Turing, 1937). Moreover, likelihood-based model selection[3] on *any* held-out data is also undecidable (*i.e.*, there is no algorithm that decides which model is better in finite time). Even worse, likelihood-based model selection is not possible for certain subsets $\Theta' \subseteq \Theta$ either — even when Θ' itself is not necessarily expressive (*e.g.*, Θ' correspond to a finite set of fixed-size Transformer EBMs, which cannot parametrize all EBMs that require more parameters). We construct one such 'pathological' distribution $\theta' \in \Theta' \subsetneq \Theta$ as a Transformer EBM.

These negative results stem from the uncomputability and inapproximability of Z (first introduced in §2.2.1). A main technical contribution of this chapter is that there is no algorithm (either deterministic or randomized) that can approximate Z well, even when we limit ourselves to fairly restrictive parametric sequence model families. Since statistical procedures are also algorithms, common randomized estimates are not useful in this scenario, either — however, we will see that for less expressive model families, such uncomputability issues do not arise. Our negative results are summarized in Table 5.1. To ensure that we can compare the goodness of different parameter vectors with confidence,[4] we have no choice but to resort to less expressive string distributions (§5.6).

[3] We note that the term *model selection* has some subtle ambiguation in the literature: some authors (Bishop and Nasrabadi, 2006, Ch. 1) use it to refer to the *decision* problem that which system is the best. On the other hand, some authors (Nishii, 1988) use it to refer to the computation of model selection *criteria* of different systems, where the best system is often implicitly assumed to be the system with the highest criterion value. In this chapter, we adopt the first meaning — namely we use the term 'model selection' to refer to the decision problem. Specifically, we opt for one of the simpler model selection variants: *likelihood-based model selection* in this chapter. In §5.5.2 we will further discuss why the model selection criteria may not be useful, even when they are well-defined.

[4] Of course, likelihood-based model selection (and computing $Z_{\tilde{p}}$) are not always needed — for example, simply deciding whether $\tilde{p}(\mathbf{x}_1) > \tilde{p}(\mathbf{x}_2)$ for two given strings $\mathbf{x}_{\{1,2\}}$. In such case the uncomputability issues we discuss are not a concern.

	exact	asymptotic
deterministic	✗ (Theorem 5.2.1)	✓ (§5.4; but no useful guarantee in finite time)
randomized	✗ (Theorem 5.3.2)	?; but ✗ for rejection sampling (Theorem 5.4.1) and ✗ for importance sampling (Theorem 5.4.2) when paired with autoregressive proposal distributions

Table 5.1: Summary of negative results: neither deterministic or randomized algorithms can estimate EBM partition functions accurately. On the other hand, popular sampling schemes such as rejection and importance sampling require their autoregressive proposal distributions to be uncomputable.

The chapter is structured as follows. In §5.1 we formally define computable estimates. In §§5.2–5.4 we describe our main technical results: there exist pathological EBM sequence models that have uncomputable partition functions, which cannot be approximated well under stochastic estimates, and do not have asymptotic estimates that have any good guarantees. In §5.5 we that argue our negative results make likelihood-based model selection impossible for expressive model families, and discuss why common estimation methods fail. Finally, we discuss three 'palliative' parametrization choices in §5.6, which all guarantee computable partition functions, at the cost of expressiveness.

5.1 Estimators

A main result of this work is that partition functions in an EC-complete family are uncomputable. Moreover, randomness does not help estimation; and correct asymptotic estimators are not useful. We define these estimators here in order to discuss the power of

different estimators concretely.

Let Θ be a parametric family. The function $f : \Theta \to \mathbb{Q}$ is an **exact estimator** if there exists a weighted deterministic Turing machine that takes $\boldsymbol{\theta}$ as input and, before halting in finite time, outputs $f(\boldsymbol{\theta}) \in \mathbb{Q}$.

Many estimation procedures are *anytime algorithms*: they do not have a predetermined runtime, and they can be stopped at any time before completion to produce a valid output. Output quality of an anytime algorithm may improve with increased runtime. Moreover, many of these algorithms have asymptotic guarantees, in that their outputs are optimal in some sense (*e.g.*, consistency) in the limit. We capture these algorithms with **asymptotic estimators**: a function $f(\boldsymbol{\theta})$ is an asymptotic estimator if there exists a weighted Turing machine that takes both $\boldsymbol{\theta}$ and an index i as input, then outputs a value $f_{\tilde{p},i} \in \mathbb{Q}$ in finite time, such that the outputs converge toward $f(\boldsymbol{\theta})$ as i increases toward infinity. (An example is $\hat{Z}_{\text{asym}}$ introduced at §5.4.)

We now extend both exact and asymptotic estimators to the stochastic case, where we compute the estimates using randomized algorithms instead of deterministic ones. As is conventional for randomized algorithms, we assume the use of probabilistic Turing machines. These have access to an infinitely long **random tape**. The tape describes an infinite binary sequence from tossing a fair two-sided coin infinitely many times. We call the random tape distribution p_τ.[5] We define a **randomized exact estimator** $\mathbf{f}$ as a

[5]Formally speaking, we define a probability space $(\Omega, \mathcal{A}, \mathbb{P})$ where $\Omega = \mathbb{B}^{\mathbb{N}}$ is our sample space, $\mathcal{A} = \{A_{\mathbf{b}} : A_{\mathbf{b}} \text{ is the set of all sequences } \in \Omega \text{ that have prefix } \mathbf{b} \in \mathbb{B}^*\}$ is our event space, and $\mathbb{P}(A) = 2^{-n}$ where n is the length of the longest shared prefix of A, is our probability function (Stroock, 2014).

weighted Turing machine with two input tapes — the random tape $\tau \in \mathbb{B}^{\mathbb{N}}$, and an input tape that has θ — and outputs $f(\theta, \tau)$ in finite time. Likewise, we say $\mathbf{f}_{\theta,i}$ is a **randomized asymptotic estimator** if there exists a function $f(\theta) \in \mathbb{R}$ and a weighted Turing machine that takes $(\theta, i), \tau$ on two input tapes, so that for all random Boolean tapes $\tau \in \mathbb{B}^{\mathbb{N}}$, we converge with $\lim_{i\to\infty} f_{\theta,i,\tau} = f(\theta)$. Many Monte Carlo estimators can be seen as randomized asymptotic estimators, including rejection sampling and importance sampling estimators.

5.2 Expressiveness and uncomputability: pathological EBMs

5.2.1 Expressive sequence distributions

To illustrate the uncomputability issues of energy-based models, we construct families of string distributions that are **expressive**: they require the full power of an EC-complete sequence model family.

We define $\mathcal{G}_k = \{\tilde{p}_{M,k} : k \in \mathbb{Q}_{>0}\}$ to be a set of weighted languages, where $\tilde{p}_{M,k}$ is parametrized by deterministic Turing machine M that takes an empty string as input ('input-free'). Let $L_M \subseteq \mathbb{B}^*$ be a prefix-free Boolean language of computation sequences of a

Turing machine M — that is, encodings of the trace of M's operations. We define

$$\tilde{p}_{M,k}(\mathbf{x}) = \begin{cases} 1/3^{|\mathbf{x}|+1} + k & \text{if } \mathbf{x} \in L_M, \text{and } \mathbf{x} \text{ encodes a valid accepting} \\ & \text{trace of } M. \\ 1/3^{|\mathbf{x}|+1} & \text{otherwise.} \end{cases} \tag{5.1}$$

The weight of any string $\mathbf{x}$, where $|\mathbf{x}| = n$, under $\tilde{p}_{M,k}$ can be computed in time $O\big(\text{poly}(n)\big)$, by verifying whether $\mathbf{x}$ is an accepting execution trace of M, from the initial state to an halting state. That is, $\tilde{p} \in \text{EC}$ (§2.2.3). We also know that for any (deterministic) machine M, the language's partition function $Z_{\tilde{p}_{M,k}}$ exists, and it must equal either 1 or $1 + k$, since there is either one halting trace ($Z_{\tilde{p}_{M,k}} = 1 + k$), or none ($Z_{\tilde{p}_{M,k}} = 1$). Therefore, each $\tilde{p} \in \mathcal{G}_k$ defines a string distribution.

Since for all $k \in \mathbb{Q}_{>0}$, $\mathcal{G}_k \subset \text{EC}$, all weighted languages $\tilde{p}_{M,k}$ have an equivalent parameter vector $\boldsymbol{\theta}_{M,k}$ in any EC-complete family $\boldsymbol{\Theta}$. Also, since each $\mathcal{G}_k$ is a bijection between the set of all input-free Turing machines and a subset of $\boldsymbol{\Theta}$, it follows that there is no exact estimator (§5.1) of the partition function of any EC-complete family (*e.g.*, Transformer EBMs (Theorem 2.3.2)), by a reduction from HALT:[6]

Theorem 5.2.1. *Let $\boldsymbol{\Theta}$ be a parametric* EC*-complete sequence model family. And let $\boldsymbol{\Theta}_k \subset \boldsymbol{\Theta}$ be bijective with $\mathcal{G}_k$. There is no $k \in \mathbb{Q}_{>0}$ for which there exists an exact estimator $\hat{Z}_k$ that takes as input $\boldsymbol{\theta} \in \boldsymbol{\Theta}_k$ as input, and computes $\hat{Z}_k(\boldsymbol{\theta}) = Z_{\tilde{p}_{\boldsymbol{\theta}}}$ in finite time.*

[6]Speaking loosely, HALT is the task of recognizing (deciding) whether a given program on an ideal computer will properly terminate in finite time or not (Sipser, 2013; Arora and Barak, 2009).

Proof. We can reduce HALT to computing our partition function. For the sake of contradiction, let us assume that for some $k \in \mathbb{Q}_{>0}$, the exact estimator $\hat{Z}_k$ exists. Our reduction from HALT of input-free Turing machines is as follows: Given any deterministic input-free Turing machine M, we build a *weighted* deterministic Turing machine: M' (§2.2.2). M' takes as input $\mathbf{x} \in \mathbb{B}^*$, and outputs $\tilde{p}_{M,k}(\mathbf{x})$ as defined in equation (5.1).[7]

M' always returns a weight for $\mathbf{x}$ in polytime. By the assumptions of EC-complete families (§2.3.2), we can build a parameter vector $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ such that $\tilde{p}_{M'} = \tilde{p}_{\boldsymbol{\theta}}$. Since from the definitions of $\mathcal{G}_k$ we know $\tilde{p}_{M'} = \tilde{p}_{M,k}$, we have $\boldsymbol{\theta} \in \boldsymbol{\Theta}_k$.

We have thus completed our reduction: if $\hat{Z}_k$ — which by definition is a computable function — existed, we could decide whether any given input-less deterministic Turing machine M halts, by first constructing the weighted Turing machine M', then the corresponding $\boldsymbol{\theta}$. By our assumption, $\hat{Z}_k(\tilde{p}_{\boldsymbol{\theta}}) = k + 1$ if and only if $\exists \mathbf{x} \in \mathbb{B}^*$ that is an accepting path of M', which is true if and only if M halts for some finite steps. Since whether M halts is undecidable, we have arrived at a contradiction (Turing, 1937; Sipser, 2013). Therefore for all $k \in \mathbb{Q}$, the algorithm $\hat{Z}_k$ does not exist. □

We note that the uncomputability of partition functions has also been observed in other expressive model families, where Turing completeness also arises (Kempe, Champarnaud, and Eisner, 2004; Chiang and Riley, 2020).

[7] Specifically, M' returns weight $1/3^{|\mathbf{x}|+1} + k$ when $\mathbf{x}$ encodes an accepting trace of M'. Otherwise, M' outputs weight $1/3^{|\mathbf{x}|+1}$.

5.2.2 An EBM whose partition function is uncomputable

Theorem 5.2.1 states there is no estimator that 'works' for a subset of parameter vectors. While every $\mathcal{G}_k$ is much smaller than its superset EC, $\mathcal{G}_k$ is still (countably) infinite. Here under the assumption that ZFC is consistent, we construct *one* weighted language $\tilde{b} \in \mathcal{G}_1$ (for simplicity; it holds for arbitrary k), where no provably correct (under ZFC) algorithm can compute $Z_{\tilde{b}}$:

Theorem 5.2.2. *Assuming ZFC is a consistent axiomatic system, there exists an* EC *weighted language* $\tilde{b} \in \mathcal{G}_1$ *such that (a)* $Z_{\tilde{b}} = 1$*, but (b) ZFC* $\nvdash \left(Z_{\tilde{b}} = 1\right)$.

Proof. Our proof hinges on Gödel's second incompleteness theorem: no consistent axiomatic system which includes Peano arithmetic (*e.g.*, ZFC) can prove its own consistency. We need a Turing machine M_b that enumerates all provable propositions under ZF, and halts if and only if M_b proves the proposition $1 = 0$. One such M_b with $7,918$ states has been built by Yedidia and Aaronson (2016).[8]

M_b is an input-free deterministic Turing machine. We construct a weighted Turing machine M' from M_b, in the manner of the proof of Theorem 5.2.1. We let M' return weight $1 + {}^1\!/_3{}^{|\mathbf{x}|+1}$, in the case M_b halts with trace $\mathbf{x}$. M' returns a weight in polytime, and therefore defines a weighted language $\tilde{b} \in \mathcal{G}_1 \subset \text{EC}$. We know from the definitions of $\mathcal{G}_1$ that $Z_{\tilde{b}} \in \{1, 2\}$. Furthermore, under our assumption that ZFC is consistent, we know $Z_{\tilde{b}} \neq 2$, otherwise there would have existed a proof that $0 = 1$. Therefore, $Z_{\tilde{b}} = 1$.

[8] Subsequent efforts have led to a construction of M_b with 748 states: https://turingmachinesimulator.com/shared/vgimygpuwi.

Assume to the contrary that there exists a proof that $Z_{\tilde{b}} = 1$ in ZFC. Then we know M_b did not halt — in other words, there is no proof that $0 = 1$ under ZFC. Therefore, this proof would also imply that ZFC is consistent, which violates Gödel's second incompleteness theorem. □

The existence of $\tilde{b}$ suggests that if there is an algorithm that approximates $Z_{\tilde{p}}$, *and* produces a witness of its approximation quality, then this algorithm will not work on *any* set of parameter vectors that can parametrize $\tilde{b}$ — even when this algorithm may work for some subsets of $\boldsymbol{\Theta}$. This is useful in allowing us to show negative results regarding finite subsets of $\boldsymbol{\Theta}$. §5.2.3 gives one such example.

5.2.3 Negative results in finite parameter subspaces

In §5.2.2, we mention that the pathological EBM $\tilde{b}$ can show negative results regarding finite subsets of $\boldsymbol{\Theta}$. Here, we provide a concrete example.

Corollary 5.2.2.1. *Assuming ZFC axioms and assuming they are consistent, there exists* $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ *such that* $Z_{\tilde{p}_{\boldsymbol{\theta}}} \in \mathbb{Q}_{>0}$ *exists, but there is no algorithm* $\hat{Z}_{\text{proof}}$ *that takes* $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ *as input, and outputs a set of strings* $\{\mathbf{x}_n : n < N\}, N \in \mathbb{N}$ *where*

- *there is a proof that* $\sum_{n=1}^{N} \tilde{p}_{\boldsymbol{\theta}}(\mathbf{x}) \geq \frac{1}{2} Z_{\tilde{p}_{\boldsymbol{\theta}}}$*; or*
- *there is a proof that* $\sum_{n=1}^{N} \tilde{p}_{\boldsymbol{\theta}}(\mathbf{x}) < \frac{1}{2} Z_{\tilde{p}_{\boldsymbol{\theta}}}$.

Proof. Assuming to the contrary that $\hat{Z}_{\text{proof}}$ existed. We construct $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ such that $\forall \mathbf{x} \in \mathbb{B}^*, \tilde{p}_{\boldsymbol{\theta}}(\mathbf{x}) = \tilde{b}(\mathbf{x})$. Either proof resulting from $\hat{Z}_{\text{proof}}(\boldsymbol{\theta})$: $\{\mathbf{x}_n : n < N\}$ can be used to prove or disprove the consistency of ZFC. □

Corollary 5.2.2.1 states that there exists a 'problematic EBM' — namely $\tilde{b}$ — where we cannot guarantee to well approximate its partition function, by accumulating finitely many string weights, regardless of the manner of accumulation (*i.e.*, how we choose strings) or the number of strings we enumerate over.

5.3 No randomized algorithm can estimate Z accurately

We've shown by Theorem 5.2.1 that for an EC-complete family, there are no exact estimators than can get Z perfectly right. In this section, we show that no randomized exact estimator for this is unbiased. Further, there isn't even an estimator whose bias is within some factor ϵ, regardless of the variance's magnitude.

Lemma 5.3.1. *Let $\boldsymbol{\Theta}$ be an* EC*-complete parametric family. There is no multiplicative factor $\epsilon \in \mathbb{Q}_{>1}$ for which every $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ can have its partition function approximated with $\hat{\mathbf{Z}}_\epsilon(\boldsymbol{\theta})$ within a factor of ϵ — with probability greater than $2/3$. That is, we cannot have*

$$P\left((1/\epsilon)\, Z_{\tilde{p}_{\boldsymbol{\theta}}} \leq \hat{\mathbf{Z}}_\epsilon(\boldsymbol{\theta}) \leq \epsilon Z_{\tilde{p}_{\boldsymbol{\theta}}}\right) > 2/3. \tag{5.2}$$

Proof. In this proof, we make use of the distribution family $\mathcal{G}_{\epsilon^2}$ (§5.2.1). We assume to the contrary that a multiplicative bound ϵ satisfying equation (5.2) exists. Recall that our assumptions state that $P\left(1/\epsilon \le \hat{\mathbf{Z}}_\epsilon(\theta)/Z_{\tilde{p}_\theta} \le \epsilon\right) > 2/3$.[9] Let M be an input-free Turing machine. And let $\tilde{p}_{\boldsymbol{\theta}} = \tilde{p}_{M,\epsilon^2} \in \mathcal{G}_{\epsilon^2}$ where $\boldsymbol{\theta} \in \boldsymbol{\Theta}$. If the Turing machine M halts, $Z_{\tilde{p}_{M,\epsilon^2}} = 1 + \epsilon^2$; therefore, $P\left((1+\epsilon^2)1/\epsilon \le \hat{\mathbf{Z}}_\epsilon(\boldsymbol{\theta}) \le (1+\epsilon^2)\epsilon\right) > 2/3$. Similarly, if M does not halt, we have $Z_{\tilde{p}_{M,\epsilon^2}} = 1$; therefore, $P\left(1/\epsilon \le \hat{\mathbf{Z}}_\epsilon(\boldsymbol{\theta}) \le \epsilon\right) > 2/3$. By combining the two conditions, we know that $P\left(\mathbb{I}(M \text{ halts})\right) = P\left(\mathbb{I}(\hat{\mathbf{Z}}_\epsilon(\boldsymbol{\theta}) \ge \epsilon + 1/\epsilon \wedge \hat{\mathbf{Z}}_\epsilon(\boldsymbol{\theta}) > \epsilon)\right) = P\left(\mathbb{I}(\hat{\mathbf{Z}}_\epsilon(\boldsymbol{\theta}) \ge \epsilon + 1/\epsilon)\right)$.

Therefore, given $(1/\epsilon)\, Z_{\tilde{p}_\theta} \le \hat{\mathbf{Z}}_\epsilon(\boldsymbol{\theta}) \le \epsilon Z_{\tilde{p}_\theta}$, we can decide if M halts by checking if $\hat{\mathbf{Z}}_\epsilon(\boldsymbol{\theta}) \ge \epsilon + 1/\epsilon$. Since the condition $(1/\epsilon)\, Z_{\tilde{p}_\theta} \le \hat{\mathbf{Z}}_\epsilon(\boldsymbol{\theta}) \le \epsilon Z_{\tilde{p}_\theta}$ only holds $2/3$ of all time, we then derandomize the randomized $\hat{\mathbf{Z}}_\epsilon$ to get a deterministic estimator of Z that has bounded error: recall that a randomized exact estimator finishes computation in finite time, regardless of the content of the random tape τ. Therefore, there are only finitely many finitely long possible 'random' sequences that $\hat{Z}_\epsilon$ will read, which we can enumerate. More concretely:

$$\begin{aligned} P\big(\hat{\mathbf{Z}}_\epsilon(\boldsymbol{\theta}) \ge \epsilon + 1/\epsilon\big) &= \mathop{\mathbb{E}}_{\tau}\left[\mathbb{I}(\hat{Z}_\epsilon(\boldsymbol{\theta}, \tau) \ge \epsilon + 1/\epsilon)\right] \\ &= \sum_{\tau \in \mathbb{B}^{m_{\boldsymbol{\theta},\epsilon}}} 1/2^{m_{\boldsymbol{\theta},\epsilon}} \mathbb{I}(\hat{Z}_\epsilon(\boldsymbol{\theta}, \tau) \ge \epsilon + 1/\epsilon) \end{aligned} \tag{5.3}$$

where $m_{\boldsymbol{\theta},\epsilon} \in \mathbb{N}$ is the maximum prefix length of the random tape that $\hat{Z}_\epsilon(\boldsymbol{\theta}, \tau)$ will use on any $\tau \in \mathbb{B}^{\mathbb{N}}$. Again $m_{\boldsymbol{\theta},\epsilon}$ is guaranteed to exist because of our assumption that $\hat{Z}_\epsilon(\boldsymbol{\theta}, \tau)$

[9] As is common, *e.g.*, in defining the complexity class BPP (Arora and Barak, 2009), the fraction $2/3$ is arbitrary. Any proportion bounded away from $1/2$ will work.

ends in finite time. Since equation (5.3) is a finite sum of computable functions, it is also computable.

From the computability of equation (5.3), it follows that we can derive a deterministic algorithm from $\hat{\mathbf{Z}}_\epsilon$ that decides whether an arbitrary input-free Turing machine halts, which is not possible. Therefore, there is no $\epsilon \in \mathbb{Q}_{>1}$, such that the randomized exact estimator $\hat{\mathbf{Z}}_\epsilon$ satisfies the multiplicative bound $P(1/\epsilon \leq \hat{\mathbf{Z}}_\epsilon(G)/Z \leq \epsilon) > 2/3$ for all $\boldsymbol{\theta} \in \boldsymbol{\Theta}$. □

Taken together, Theorem 5.2.1 and Lemma 5.3.1 state that no exact estimator $\hat{Z}$ — whether randomized or deterministic — can approximate Z with good confidence. In Theorem 5.3.2 below, we will make an even stronger claim: regardless of the dispersion magnitude, it is impossible to bound the mean of random exact estimators of Z to within any (computable) multiplicative factor. This is because the mean of $\hat{\mathbf{Z}}_\epsilon$ can be computed in finite time, by derandomizing $\hat{\mathbf{Z}}_\epsilon$ similarly to our proof of Lemma 5.3.1:

Theorem 5.3.2. *Let $\boldsymbol{\Theta}$ be an* EC*-complete parametric family. There is no multiplicative bound $\epsilon \in \mathbb{Q}_{>1}$ such that there exists a randomized exact estimator $\hat{\mathbf{Z}}_\epsilon$ that guarantees $1/\epsilon \leq \mathbb{E}[\hat{\mathbf{Z}}_\epsilon(\boldsymbol{\theta})]/Z_{\tilde{p}_{\boldsymbol{\theta}}} \leq \epsilon$, for every $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ where $\tilde{p}_{\boldsymbol{\theta}}$ is normalizable.*

Proof. We define τ to be distributed according p_τ. Therefore the mean $\mathbb{E}[\hat{\mathbf{Z}}_\epsilon(\boldsymbol{\theta})]$ can be expanded as $\mathbb{E}_{\tau\sim p_\tau}[\hat{\mathbf{Z}}_\epsilon(\boldsymbol{\theta})] = \sum_{\tau\in\mathbb{B}^{\mathbb{N}}} p_\tau(\tau)\hat{Z}_\epsilon(\boldsymbol{\theta}, \tau)$. Following the same derandomization technique in the proof of Lemma 5.3.1, we can find some $m \in \mathbb{N}$ such that $\sum_{\tau\in\mathbb{B}^{\mathbb{N}}} p_\tau(\tau)\hat{Z}_\epsilon(\boldsymbol{\theta}, \tau) = \sum_{\tau\in\mathbb{B}^m} 1/2^m \hat{Z}_\epsilon(\boldsymbol{\theta}, \tau)$ in finite time. And subsequently, we can compute $\mathbb{E}[\hat{\mathbf{Z}}_\epsilon(\boldsymbol{\theta})]$ exactly in finite time. Let the exact estimator be $\overline{Z}_\epsilon$. We then have $\overline{Z}_\epsilon(\boldsymbol{\theta}) = \mathbb{E}[\hat{\mathbf{Z}}_\epsilon(\boldsymbol{\theta})]$.

Assuming to the contrary that we could guarantee $1/\epsilon \leq \overline{Z}_\epsilon(\boldsymbol{\theta}) \leq \epsilon$. Since $\overline{Z}_\epsilon$ is a deterministic estimator, we can write $P(1/\epsilon \leq \overline{Z}_\epsilon(\boldsymbol{\theta}) \leq \epsilon) = 1$, which contradicts Lemma 5.3.1. Therefore such an estimator $\overline{Z}_\epsilon(\boldsymbol{\theta})$ does not exist. □

5.4 Common asymptotic estimates do not give useful guarantees

Let's now recap the progress we've made so far. We've shown that no (deterministic) exact estimator can get Z exactly right in general (Theorem 5.2.1), lest it need to solve Halt. Further, no randomized exact estimator can approximate it within any given relative tolerance, with good confidence (Theorem 5.3.2).

But what about *asymptotic* estimators? We do know there are correct asymptotic estimators of Z. For example, consider the following asymptotic estimator $\hat{Z}(\boldsymbol{\theta})$ backed by a weighted Turing machine that takes $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ and $i \in \mathbb{N}$ as inputs, and returns $f_{\boldsymbol{\theta},i} \triangleq \sum_{\mathbf{x}:\mathbf{x}\in\mathbb{B}^*,|\mathbf{x}|\leq i} \tilde{p}_{\boldsymbol{\theta}}(\mathbf{x})$. We have $\lim_{i\to\infty} \hat{Z}_{\boldsymbol{\theta},i} = Z_{\tilde{p}_{\boldsymbol{\theta}}}$, so $\hat{Z}$ is asymptotically correct. However, $\hat{Z}_{\text{asym}}$ does not have a convergence rate guarantee: for any $i \in \mathbb{N}$, $\|\hat{Z}_{\boldsymbol{\theta},i} - Z_{\tilde{p}_{\boldsymbol{\theta}}}\|$ is uncomputable. We also do not know how much can we improve our estimator when we increment i. As Corollary 5.2.2.1 suggests, we likely cannot have such a guarantee.

In this section, we formalize this intuition for two popular asymptotic estimators: rejection and importance sampling methods (with other asymptotic estimators left as future work). Specifically, we show that any parametric family that is able to parametrize $\tilde{b}$ from

§5.2.2 cannot have provably useful locally normalized distributions (§2.2.4) as proposal distributions.

5.4.1 Rejection sampling estimator of Z cannot be guaranteed to be possible

Rejection sampling (Owen, 2013) is a common exact sampling method, applicable even when we cannot sample from an unnormalized distribution $\tilde{p}$. We instead sample from an easy-to-sample distribution q, then stochastically reject samples, to ensure the probability that a sample $\mathbf{x}$ is kept is proportional to $\tilde{p}(\mathbf{x})$.

For rejection sampling to work, the candidate q's support must contain the target $\tilde{p}$'s entire support, so that all true points can be sampled. We also need some finite constant c so that cq envelops $\tilde{p}$:

$$\exists c \in \mathbb{R}_{>0} \text{ such that } \forall \mathbf{x} \in \mathbb{B}^*, (\tilde{p}(\mathbf{x})/q(\mathbf{x})) \leq c.$$

We will show that for certain EBMs, one cannot formally guarantee the existence of an eligible $q \in \text{LN}$ using only ZFC.

Theorem 5.4.1. *Using ZFC as our axiom set and assuming ZFC is consistent, then there exists a normalizable* EC *weighted language $\tilde{p}$, where there does not exist a consistent locally normalized proposal distribution $q \in \text{LN}$, and $c_q \in \mathbb{Q}_{>0}$, such that it can be proven $\forall \mathbf{x} \in \mathbb{B}^*$, $\tilde{p}(\mathbf{x})/q(\mathbf{x}) \leq c_q$.*

Proof. By contradiction; let $\tilde{p} = \tilde{b}$, where $\tilde{b}$ was first introduced in §5.2.2. Assume that there exists $q \in \text{LN}$ where it is proven that $\forall \mathbf{x} \in \mathbb{B}^*, {}^{\tilde{p}(\mathbf{x})}/_{q(\mathbf{x})} < c$, where $c \in \mathbb{Q}_{>0}$.

We can either prove ZFC is consistent, or is inconsistent, as follows:

1. By our assumptions, $q \in \text{LN}$ scores every string $\mathbf{x}$ as $q_\theta(\mathbf{x}) f(\boldsymbol{\theta}) \in \mathbb{Q}_{>0}$. Also, because q is consistent and locally normalized, there exists $n \in \mathbb{N}$ such that we can prove $\forall \mathbf{x} \in \mathcal{X}_{>n} \{\mathbf{x} : \mathbf{x} \in \mathbb{B}^*, |\mathbf{x}| > n\}, q(\mathbf{x}) < {}^1/_c$ (Proposition 2.2.1). We will just prove the existence of n constructively by enumerating the strings in $\mathbb{B}^*$ in increasing length. Let the complement set $\mathcal{X}_{\leq n} = \mathbb{B}^* - \mathcal{X}_{>n}$.

2. The proof then examines the finitely many strings in $\mathcal{X}_{\leq n}$.

   (a) If any of these strings $\mathbf{x}'$ has $\tilde{b}(\mathbf{x}') > 1$, then we know $\mathbf{x}'$ encodes an accepting trace of M_b (§5.2.2). Therefore, M_b halts, which implies there is a proof of the inconsistency of ZFC (from $\mathbf{x}'$).

   (b) If none of these strings $\in \mathcal{X}_{\leq n}$ has $\tilde{b}(\mathbf{x}) > 1$, then we know there is also no string $\mathbf{x}'' \in \mathcal{X}_{>n}$, such that $\tilde{b}(\mathbf{x}'') > 1$. This is because of our assumption $c \geq {}^{\tilde{b}(\mathbf{x})}/_{q(\mathbf{x})}$, which in turn means that $\tilde{b}(\mathbf{x})$ is less than 1 on these long strings $\mathcal{X}_{>n}$. Therefore, M_b does not halt, which implies there is no proof of the inconsistency of ZFC — in other words, ZFC is consistent.

Assuming ZFC is consistent, neither a proof of ZFC, or a disproof of ZFC, is possible. We have therefore arrived at a contradiction. Therefore, q does not exist. □

Theorem 5.4.1 implies that there is no way of ensuring rejection sampling works, not only for any EC-complete families, but also for any parametric family that can parametrize $\tilde{b}$.

5.4.2 Importance sampling estimator of Z cannot be guaranteed to be effective.

Similar to the case of rejection sampling, one cannot guarantee an importance-sampling estimator of Z to be 'good' — in this case, we mean that there cannot be a proof that the importance sampling variance is finite.

We first formalize importance sampling estimators of Z as randomized asymptotic estimators (§5.1). Let

$$\hat{\mathbf{Z}}^{q}_{\theta,N} = \frac{1}{N}\sum_{n=1}^{N}\frac{\tilde{p}_{\theta}(\mathbf{x}^{(n)})}{q(\mathbf{x}^{(n)})}$$

be an N-sample importance sampling estimator of $Z_{\tilde{p}_\theta}$ under q, so all $\mathbf{x}^{(n)}$ are samples from $q \in \mathrm{LN}$.

We generally want to minimize the variance of $\hat{\mathbf{Z}}^{q}_{\theta,N}$ under q: $\mathrm{Var}_q\left(\hat{\mathbf{Z}}^{q}_{\theta,N}\right)$ (Owen and Zhou, 2000). And we certainly do not want $\mathrm{Var}_q\left(\hat{\mathbf{Z}}^{q}_{\theta,N}\right) = \infty$. Unfortunately, for certain EBMs, we cannot guarantee there is a good locally normalized proposal distribution that has finite variance:

Theorem 5.4.2. *Let $\boldsymbol{\Theta}$ be an* EC*-complete parametric family. Assuming ZFC axioms and assuming they are consistent, there exists $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ where there does not exist a consistent*

locally normalized "proposal" distribution $q \in \mathrm{LN}$ *such that it can be proven under ZFC that* $\mathrm{Var}_q\left(\hat{\mathbf{Z}}^q_{\theta,N}\right) < c \neq \infty$, *where* $c \in \mathbb{Q}_{>0}$.

Proof. let $\tilde{p} = \tilde{b}$, where $\tilde{b}$ is first introduced in §5.2.2. Assume that there exists $q \in \mathrm{LN}$ where it is proven that $\mathrm{Var}_q(\hat{\mathbf{Z}}^q_{\theta,N}) = \sigma^2 < c \in \mathbb{Q}_{<0} \neq \infty$.

We have

$$\begin{aligned}
&\mathrm{Var}_q(\hat{\mathbf{Z}}^q_{\theta,N}) \\
&\quad \triangleq \frac{1}{N}\left\{\sum_{\mathbf{x}\in\mathbb{B}^*}\left[\left(\frac{\tilde{p}_\theta(\mathbf{x})}{q(\mathbf{x})}\right)^2 q(\mathbf{x})\right] - {Z_{\tilde{p}_\theta}}^2\right\} \\
&\quad = \frac{Z^2_{\tilde{p}_\theta}}{N}\left(\sum_{\mathbf{x}\in\mathbb{B}^*}\frac{p^2_\theta(\mathbf{x})}{q(\mathbf{x})} - 1\right) \\
&\quad = \sigma^2,
\end{aligned}$$

where $p_\theta(\mathbf{x}) \triangleq \frac{\tilde{p}_\theta(\mathbf{x})}{Z_{\tilde{p}_\theta}}$. After some manipulation, we have

$$\sum_{\mathbf{x}\in\mathbb{B}^*}\frac{p^2_\theta(\mathbf{x})}{q(\mathbf{x})} = \underbrace{\frac{N\sigma^2 + {Z_{\tilde{p}_\theta}}^2}{{Z_{\tilde{p}_\theta}}^2}}_{=s\leq Nk+1}.$$

Since $\forall \mathbf{x} \in \mathbb{B}^*, \frac{p^2_\theta(\mathbf{x})}{q(\mathbf{x})} \geq 0$, we have

$$\forall \mathbf{x} \in \mathbb{B}^*, \frac{p^2_\theta(\mathbf{x})}{q(\mathbf{x})} \leq s; \tag{5.4}$$

in particular, if $\mathbf{x}'$ encodes a halting trace of M_b, from §5.2.1 we know

$$\begin{aligned} p_\theta(\mathbf{x}') &= \frac{\tilde{p}_\theta(\mathbf{x}')}{Z_{\tilde{p}_\theta}} \\ &= \frac{1 + \left(\frac{1}{3}\right)^{|\mathbf{x}'|+1}}{2} \\ &> 1/2. \end{aligned} \tag{5.5}$$

Combining equations (5.4) and (5.5), we have for any halting trace $\mathbf{x}'$ of M_b,

$$s \geq \frac{p_\theta^2(\mathbf{x}')}{q(\mathbf{x}')} > \frac{1/4}{q(\mathbf{x}')}.$$

After some more arrangement,

$$q(\mathbf{x}') \geq \frac{1}{4s} \geq \frac{1}{4(N\sigma^2 + 1)} \geq \frac{1}{4(Nc + 1)} > 0.$$

As in the proof of Theorem 5.4.1, the existence of such a q allows us to either prove or disprove the consistency of ZFC. For the sake of completeness, we include a proof sketch below:

1. By our assumptions, $q \in \text{LN}$ scores every string $\mathbf{x}$ as $q_\theta(\mathbf{x})f(\theta) \in \mathbb{Q}_{>0}$. Also, because q is consistent and locally normalized, there exists $n \in \mathbb{N}$ such that we can prove $\forall \mathbf{x} \in \mathcal{X}_{>n}\{\mathbf{x} : \mathbf{x} \in \mathbb{B}^*, |\mathbf{x}| > n\}, q(\mathbf{x}) < \frac{1}{4(Nc+1)}$ (Proposition 2.2.1). We will just prove the existence of n constructively by enumerating the strings in $\mathbb{B}^*$ in increasing length. Let the complement set $\mathcal{X}_{\leq n} = \mathbb{B}^* - \mathcal{X}_{>n}$.

2. The proof then examines the finitely many strings in $\mathcal{X}_{\leq n}$.

   (a) If any of these strings $\mathbf{x}'$ has $\tilde{p}_{\theta}(\mathbf{x}') > 1$, then we know $\mathbf{x}'$ encodes an accepting trace of M_b (§5.2.2). Therefore, M_b halts, which implies there is a proof of the inconsistency of ZFC.

   (b) If none of these strings $\in \mathcal{X}_{\leq n}$ has $\tilde{p}_{\theta}(\mathbf{x}) > 1$, then we know there is also no string $\mathbf{x}'' \in \mathcal{X}_{>n}$, such that $\tilde{p}_{\theta}(\mathbf{x}'') > 1$, since $\tilde{p}_{\theta}(\mathbf{x}') > 1 \iff \tilde{b}(\mathbf{x}') > 1 \iff M_b$ halts; and we have already shown that for any accepting trace $\mathbf{x}'$ of M_b, $q(\mathbf{x}') \geq 1/4(Nc+1)$. Therefore, M_b does not halt, which implies there is no proof of the inconsistency of ZFC — and therefore ZFC is consistent.

Assuming ZFC is consistent, neither a proof of ZFC, or a disproof of ZFC, is possible. We have therefore therefore arrived at a contradiction. Therefore, such a q does not exist. □

5.5 Uncomputable Z causes parameter estimation problems

Theorems 5.2.1 and 5.2.2 state that it is generally impossible to estimate partition functions in an expressive parametric family, such as certain subsets of an EC-complete family. Here we show how parameter estimation is made difficult as well: parameter identifiability is formally undecidable for an EC-complete family. Likelihood-based model selection is not possible, either, despite attempts to circumvent this (Table 5.2). Negative results in

this section are particularly relevant to practitioners who seek to design a parametric model family of energy-based sequence models — they must proactively limit the family's expressiveness so that it *cannot* capture certain 'pathological' weighted languages, otherwise there will be no likelihood-based model selection algorithm for this family. We will further show that these pathological weighted languages are not that exotic (§5.5.3): they can be captured by fixed-size Transformer EBMs. This also implies that there is generally no likelihood-based model selection algorithm for fixed-size Transformer EBMs.

5.5.1 Parameter identifiability under EC-complete families is undecidable

Parameter identifiability refers to the property that two different parameter vectors define different string distributions (Lehmann and Casella, 2006). But we in general cannot ascertain whether this condition holds, even for finite subsets of an EC-complete family:

Theorem 5.5.1. *Let Θ be an* EC*-complete family. There is no algorithm that takes $\boldsymbol{\theta}_1 \in \Theta$, $\boldsymbol{\theta}_2 \in \Theta$ as input, and decides whether $\tilde{p}_{\boldsymbol{\theta}_1}$ and $\tilde{p}_{\boldsymbol{\theta}_2}$ are the same weighted language.*

Proof. Assuming to the contrary that such an algorithm DISTINCT exists. We would be able to reduce HALT of input-free Turing machines to DISTINCT: we first construct $\boldsymbol{\theta}_1 \in \Theta$ such that $\tilde{p}_{\boldsymbol{\theta}_1}(\mathbf{x}) = 1/3^{|\mathbf{x}|+1}$. We then construct $\boldsymbol{\theta}_2 \in \Theta$ such that $\tilde{p}_{\boldsymbol{\theta}_2} = \tilde{p}_{M,1} \in \mathcal{G}_1$ (§5.2.1). DISTINCT$(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2)$ is YES if and only if M halts. □

By Theorem 5.5.1, we know whether two parameter vectors within an expressive

sequence model family identify with the same distribution is undecidable. Since identifiability is a necessary condition of the existence of a consistent estimator (Martín and Quintana, 2002), it is therefore impossible to formally prove an estimator's consistency for EC-complete sequence model families.

5.5.2 Deciding which model is better is generally impossible for EC-complete sequence model families

The negative results of §5.5.1 might not deter adventurous machine learning practitioners: 'off-label' use of parameter estimators that are *not* known to be consistent is quite common in the machine learning literature — one usually just selects the best model $\boldsymbol{\theta}^*$ among finitely many candidates (say $\{\boldsymbol{\theta}_1 \dots \boldsymbol{\theta}_N\}$) that achieves highest held-out likelihood: $\boldsymbol{\theta}^* = \text{argmax}_{\boldsymbol{\theta}_n : 1 \leq n \leq N} \prod_{\mathbf{x} \in \mathcal{D}} p_{\boldsymbol{\theta}_n}(\mathbf{x})$, where $\mathcal{D}$ is a finite set of strings.

The activity of choosing the best model out of a set of candidates can be characterized as **model selection**. How *best* is defined may differ across tasks. In this chapter we assume the criterion of held-out log likelihood. And we call our model selection task **likelihood-based model selection**. It is often desirable if we can do likelihood-based model selection, deterministically and in finite time (*i.e.*, as an exact estimator).[10] Following Theorems 5.3.2

[10] Alternatively, one may also consider model selection as deterministic or probabilistic anytime algorithms, which correspond to deterministic and randomized asymptotic estimators (Table 5.1). These estimators likely exist, in contrast to the negative results in Theorem 5.5.2.

and 5.5.2, it may not seem surprising that for any fixed data sample and any constant $c \geq 1$, there is no algorithm that can approximate held-out likelihood for all models in an EC-complete family $\boldsymbol{\Theta}$.[11] Below we formally show that log-likelihood-based model selection is generally impossible for EC-complete sequence model families:

Theorem 5.5.2. *For any $\tilde{p} \in$ EC and for any EC-complete family $\boldsymbol{\Theta}$, there is no algorithm $\textsc{Better}_{\tilde{p}}$ that takes two parameter vectors $\boldsymbol{\theta}_1 \in \boldsymbol{\Theta}$, $\boldsymbol{\theta}_2 \in \boldsymbol{\Theta}$, and returns YES if $\mathrm{KL}(p||p_{\theta_1}) \geq \mathrm{KL}(p||p_{\theta_2})$, and NO otherwise, where $p, p_{\theta_1}, p_{\theta_2}$ are string distributions defined by $\tilde{p}, \tilde{p}_{\theta_1}, \tilde{p}_{\theta_2}$ respectively.*

Proof. By contradiction; assume that for some $\tilde{p} \in$ EC, $\textsc{Better}_{\tilde{p}}$ exists. We know there exists a weighted Turing machine (also denoted as $\tilde{p}$) that on any string $\mathbf{x}$, terminates in $O(\mathrm{poly}(|\mathbf{x}|))$ and outputs $\tilde{p}(\mathbf{x})$ (§2.2.2).

We show how we can reduce from $\textsc{Halt}$ to $\textsc{Better}_{\tilde{p}}$. Given any arbitrary input-less Turing machine M, we can define a new weighted Turing machine $\tilde{p}'_M$ that on input string $\mathbf{x}$, $\tilde{p}'_M$ first simulates $\tilde{p}$ on it and keeps a record of $\tilde{p}(\mathbf{x})$ somewhere on the tape. $\tilde{p}'_M$ then verifies whether $\mathbf{x}$ is an encoded accepting trace of M. If $\mathbf{x}$ is indeed an encoded accepted trace of M, $\tilde{p}'_M$ outputs $\tilde{p}(\mathbf{x}) + 1$. Otherwise $\tilde{p}'_M$ outputs $\tilde{p}(\mathbf{x})$.

Let $\boldsymbol{\theta}_1 \in \boldsymbol{\Theta}$ parametrize $\tilde{p}'_M$, and let $\boldsymbol{\theta}_2 \in \boldsymbol{\Theta}$ parametrize $\tilde{p}$, such that $\tilde{p}_{\boldsymbol{\theta}_1}(\mathbf{x}) = \tilde{p}'_M(\mathbf{x}), \tilde{p}_{\boldsymbol{\theta}_2}(\mathbf{x}) = \tilde{p}(\mathbf{x}), \forall \mathbf{x} \in \mathbb{B}^*$.

[11] A proof sketch is as follows: For any deterministic Turing machine M and any normalized distribution q_0, let q be a variant of q_0 defined by $\tilde{q}(\mathbf{x}) = q_0(\mathbf{x}) + c^2$ if $\mathbf{x}$ is out-of-sample and is an accepting trace of M on empty input, else $\tilde{q}(\mathbf{x}) = q_0(\mathbf{x})$. So the N-sample likelihood of q will vary by a factor of $(1 + c^2)^N > c^2$ according to whether M halts on an out-of-sample input. We can easily determine whether M halts on an in-sample input. So if there were a c-approximation algorithm, we could determine whether M halts or not.

We know that $\mathrm{KL}(p||p) = 0$ for any distribution p. $\textsc{better}_{\tilde{p}}(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2)$ returns YES if and only if $\mathrm{KL}(p||p_{\boldsymbol{\theta}_1}) = 0 \geq \mathrm{KL}(p||p_{\boldsymbol{\theta}_2})$, which implies that $\tilde{p}_{\boldsymbol{\theta}_2}$ and $\tilde{p}_{\boldsymbol{\theta}_1}$ define the same distribution, which in turn implies that $\forall \mathbf{x} \in \mathbb{B}^*, \tilde{p}(\mathbf{x}) = \tilde{p}'_M(\mathbf{x})$, and that M never halts. Similarly, $\textsc{Better}_{\tilde{p}}(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2)$ returns NO if and only if M halts. We have thus completed our reduction from Halt; and therefore algorithm $\textsc{Better}_{\tilde{p}}$ does not exist for any $\tilde{p} \in \mathrm{EC}$. □

A 'best effort' non-terminating program exists for likelihood-based model selection — but it may not be useful. One may wonder why we need a likelihood-based model selection *algorithm* (which by definition terminates in finite time) — why cannot we resort to a 'best effort' approximation? Indeed, if we were to drop the assumption that $\textsc{Better}_{\tilde{p}}$ is an algorithm, one could simply implement it as a program that has a loop which is potentially infinite. The program would brute force through all strings $\in \mathbb{B}^*$ to improve its estimate of either $p_{\boldsymbol{\theta}_1}$ or $p_{\boldsymbol{\theta}_2}$, and stop its execution either when it could confidently make a decision, or when it was interrupted[12] by the user the time they deemed opportune — by which time the program would use the collected samples to make best effort estimates of held-out likelihood and decide, before finally stopping.

Unfortunately, there are two issues that stop the program we described above to be useful:

- Theorem 5.5.2 implies that there are certain $\boldsymbol{\theta}$'s which would make the program stuck in an infinite loop, and must be interrupted to get it out of the loop.

[12] If this program were to be run on a real-life computer (that somehow magically had access to infinite storage and power, to keep the program running indefinitely), a user could *interrupt*, *e.g.*, by sending the `SIGINT` signal using the key combination `CTRL - C` (IEEE, 2018).

- The 'best effort estimates' derived are not useful, in the sense we could not establish any confidence interval — otherwise we could also have decided HALT by using the derandomization technique (as in our proof of Theorem 5.3.2). Therefore, we could not vouch for the decision we made using them, either.

Analogously, one can show that while there do exist consistent model selection criteria that are based on the maximum likelihood principle (*e.g.*, BIC (Nishii, 1984)), accurate approximation of these criteria may not be computed in finite-time for an EC-complete family, which render them not particularly useful, for a similar reason. Finally, the undecidability of likelihood-based model selection implies practitioners cannot have a confidence interval-based argument that goes like 'with 95% confidence, $\tilde{p}_{\theta_1}$ is a better approximation than $\tilde{p}_{\theta_2}$ of the true distribution p': such arguments rely on the uniform convergence of estimates (*e.g.*, Heinrich and Kahn (2018)). Even if such formal results existed for some EC-complete families — that variance of sample likelihood decreases asymptotically in sample size at a known rate — we still cannot establish any confidence interval, simply because we cannot compute sample likelihood.

5.5.3 Impossibility of likelihood-based model selection in fixed-size Transformer EBM families

Theorem 5.5.3 is a sibling theorem to Theorem 5.5.2. Just as we show $Z_{\tilde{b}}$ is uncomputable (§5.2.2), we can prove that likelihood-based model selection is not only impossible for

EC-complete parametric families (where the length of a parameter vector is unbounded), but also impossible for fixed-size Transformer EBMs with a large enough embedding size.[13]

Theorem 5.5.3. *Assuming ZFC as our axiomatic system, for any $\tilde{p} \in \mathrm{EC}$, there exists $d_0 \in \mathbb{N}$ such that for all $d \geq d_0$, $\tilde{p}$ can be captured by one-encoder-layer four-decoder-layer Transformer networks $\Theta^{(d)}$ (Theorem 2.3.2) with embedding size d, where there is no provably correct algorithm $\textsc{Better}_{\tilde{p},d}$ that takes two parameter vectors $\boldsymbol{\theta}_1 \in \Theta^{(d)}$, $\boldsymbol{\theta}_2 \in \Theta^{(d)}$, and returns YES if $\mathrm{KL}(p||p_{\theta_1}) \geq \mathrm{KL}(p||p_{\theta_2})$, and NO otherwise, where $p, p_{\theta_1}, p_{\theta_2}$ are string distributions defined by $\tilde{p}, \tilde{p}_{\theta_1}, \tilde{p}_{\theta_2}$ respectively.*

Proof. Let M_b be the input-free unweighted Turing machine we built in our proof of Theorem 5.2.2, whose behavior is independent of ZFC. Let M_0 be any weighted Turing machine that defines the EC weighted language $\tilde{p}$. And let M_1 be a weighted Turing machine that weights $\mathbf{x}$ as $\tilde{p}(\mathbf{x}) + 1$ if and only if $\mathbf{x}$ encodes an accepting trace of M_b; and M_1 weights $\mathbf{x}$ as $\tilde{p}(\mathbf{x})$ otherwise. Since checking whether $\mathbf{x}$ is a valid trace of M_b is in $O(\mathrm{poly}(|\mathbf{x}|))$, M_1 defines an EC weighted language.

Let p_{M_1} be the string distribution defined by M_1. By an argument similar to our proof of Theorem 5.5.2, no algorithm that is provably correct can decide if $\mathrm{KL}(p||p_{M_1}) \geq \mathrm{KL}(p||p)$.

We note that for any weighted Turing machine M with fewer than n states, we can build another weighted Turing machine M' which has n states, such that M and M' define the same weighted language, simply by having (finitely many) additional unreachable states in M'. Since any weighted Turing machine with n states can be implemented as a

[13] We use $\Theta^{(d)}$ to denote a Transformer family with embedding size d.

1-encoder-layer-4-decoder-layer Transformer networks with an embedding size $\in O(n)$, it follows that there exists $d_0 \in \mathbb{N}$ such that both M_0 and M_1 can be encoded as parameter vectors within a fixed-size model family with $d \geq d_0$. □

5.6 Palliative alternatives

What is a practitioner to do, given that this class of models is unlearnable in general? We emphasize that a model family does *not* have to be EC-complete to suffer from model selection problems (§5.5.2) — for example, model selection is also impossible for fixed-size Transformer networks with large enough d's, by an extension to Theorem 5.5.2 (see Theorem 5.5.3 in §5.5.3).[14] [15] In other words, to ensure the problem of uncomputability does not haunt us, the best we can do is to cripple $\tilde{p}$ so severely that uncomputability is impossible.

We identify three palliative choices that restrict the family of EBMs. Each cripples the model $\tilde{p}$ in its own way, affording computability at the cost of expressiveness. The choice of triage infuses the model with an inductive bias; we must tailor our models based on our prior beliefs about the problem domain.

[14] In addition to model selection issues, it may also be difficult to acquire unbiased gradients of $\log Z$: $\nabla(\log Z) \triangleq 1/Z \nabla Z$, which are needed for MLE-based training.

[15] Limiting ourselves to small d's to avoid uncomputability issues may not be practical; we leave finding the largest d that provably does not involve uncomputability problems — if it is even possible — as future work.

Restricting support($\tilde{p}$) **to be finite.** If $\tilde{p}$ assigns non-zero weights to only finitely many strings, then $Z_{\tilde{p}}$ is a finite sum of rational numbers, and is also rational. Here, sampling-based inference and estimation methods return to their usefulness. One way to ensure support($\tilde{p}$) is finite is by upper-bounding the maximum string length (*e.g.*, (Bakhtin et al., 2021)).

The finite-support restriction imposes an obvious limitation that it cannot handle long strings. Moreover, while $Z_{\tilde{p}}$ is computable when support($\tilde{p}$) is finite, this quantity can still be practically inapproximable, assuming that $\tilde{p}$ is expressive (*e.g.*, $\tilde{p}$ is an EC weighted language that has finite support), except for very short strings. Let n be the longest string under $\tilde{p}$ to have non-zero weight. Assuming NP $\not\subseteq$ P/poly, no randomized estimator of $Z_{\tilde{p}}$ that is a good approximation can have a guaranteed $O(\text{poly}(n))$ time complexity (Chandrasekaran, Srebro, and Harsha, 2008). However, if $\tilde{p}$ has limited expressiveness (*e.g.*, when $\tilde{p}$ is an Ising model where a string weight is the sum of pairwise weights), then FPRAS algorithms for approximating $Z_{\tilde{p}}$ may exist when $\tilde{p}$ describe a low-degree (≤ 2) graph (Jerrum and Sinclair, 1993; Luby and Vigoda, 1999). However, for high-degree graphs (≥ 3) it can be shown no FPRAS algorithm for approximating $Z_{\tilde{p}}$ exists, assuming RP $\neq$ NP (Galanis, Stefankovic, and Vigoda, 2016).

Autoregressive parametrization of $\tilde{p}$. An alternative choice is to confine ourselves to autoregressive models, *i.e.*, locally normalized string distributions (§2.2.4). Under an autoregressive model p, $Z_p = 1$ by definition. We also note that any (unnormalized) distribution $\tilde{p}$ obtained by removing probability mass from p will have a *computable*

partition function, as long as $\tilde{p} \in \text{EC}$:

Theorem 5.6.1. *Let p be any* LN *weighted language. Any $\tilde{p} \in \text{EC}$ where $\forall \mathbf{x} \in V^*, \tilde{p}(\mathbf{x}) \leq p(\mathbf{x})$ has a computable $Z_{\tilde{p}}$.*

Proof. We prove Theorem 5.6.1 by constructing an algorithm $\hat{Z}_{\tilde{p}} : \mathbb{Q}_{>0} \longrightarrow \mathbb{Q}_{\geq 0}$ that approximates $Z_{\tilde{p}}$ within any desired positive rational error ϵ: namely $|\hat{Z}_{\tilde{p}}(\epsilon) - Z_{\tilde{p}}| \leq \epsilon$.

Let $\hat{Z}_n = \sum_{\ell=0}^{n} \sum_{\mathbf{x}:|\mathbf{x}|=\ell} p(\mathbf{x})$. We first observe that $\lim_{n\to\infty} Z_n = 1$ by Proposition 2.2.1. In other words, $\lim_{n\to\infty} 1 - \hat{Z}_n = 0$, or equivalently, given any $\epsilon > 0$, there exists $n \in \mathbb{N}$ such that for all $n' \geq n, n' \in \mathbb{N}$, $(1 - \hat{Z}_{n'}) < \epsilon$.

Therefore, given any $\epsilon > 0$, $\exists n \in \mathbb{N}$ that divides $\mathbb{B}^*$ in two sets: $\mathcal{X}_{\geq n} = \{\mathbf{x} : \mathbf{x} \in \mathbb{B}^*, |\mathbf{x}| \geq n\}$, where $\sum_{\mathbf{x}\in\mathcal{X}_{\geq n}} p(\mathbf{x}) < \epsilon$, and $\mathcal{X}_{<n} = \mathbb{B}^* - \mathcal{X}_{\geq n}$. We are guaranteed to find n by enumerating all candidates from $\mathbb{N}$.

We can thus implement $\hat{Z}_{\tilde{p}}$ as the following program: given $\epsilon \in \mathbb{Q}_{>0}$, we first find the smallest $n \in \mathbb{N}$, and partition $\mathbb{B}^*$ into two sets $\mathcal{X}_{<n}, \mathcal{X}_{\geq n}$ as we described in the previous paragraph. We then return $\hat{Z}_{\tilde{p}}(\epsilon) = \sum_{\mathbf{x}\in\mathcal{X}_{<n}} \tilde{p}(\mathbf{x})$.

We argue that $\hat{Z}_{\tilde{p}}$ is a computable function. We first repeat that since $n \in \mathbb{N}$ exists, we will find n in finite time, simply by enumerating. And since the set $\mathcal{X}_{<n} \subset \mathbb{B}^*$ is finite, $\hat{Z}_{\tilde{p}}(\epsilon) = \sum_{\mathbf{x}\in\mathcal{X}_{<n}} \tilde{p}(\mathbf{x})$ can computed in finite time (under our assumption $\tilde{p} \in \text{EC}, \forall \mathbf{x} \in \mathbb{B}^*, \tilde{p}(\mathbf{x})$ can be computed in time $O(\text{poly}(|\mathbf{x}|))$). Since the program we described above terminates in finite time, $\hat{Z}_{\tilde{p}}$ is a computable function.

What remains is to show that the approximation error $|Z_{\tilde{p}} - \hat{Z}_{\tilde{p}}(\epsilon)|$ is no greater than ϵ; *i.e.*, that $|Z_{\tilde{p}} - \hat{Z}_{\tilde{p}}(\epsilon)| \leq \epsilon$.

It is easy to show that $0 \leq Z_{\tilde{p}} - \hat{Z}_{\tilde{p}}(\epsilon)$, after which we only need to show that $Z_{\tilde{p}} - \hat{Z}_{\tilde{p}}(\epsilon) \leq \epsilon$. Expressing both terms as sums, we have that $\sum_{\mathbf{x} \in \mathbb{B}^*} \tilde{p}(\mathbf{x}) - \sum_{\mathbf{x} \in \mathcal{X}_{<n}} \tilde{p}(\mathbf{x}) = \sum_{\mathbf{x} \in \mathcal{X}_{\geq n}} \tilde{p}(\mathbf{x})$, which is a sum of nonnegative terms and thus nonnegative.

To show that $Z_{\tilde{p}} - \hat{Z}_{\tilde{p}}(\epsilon) \leq \epsilon$, we express both terms as sums again:

$$\sum_{\mathbf{x} \in \mathbb{B}^*} \tilde{p}(\mathbf{x}) - \sum_{\mathbf{x} \in \mathcal{X}_{<n}} \tilde{p}(\mathbf{x}) = \sum_{\mathbf{x} \in \mathcal{X}_{\geq n}} \tilde{p}(\mathbf{x}).$$

By the definition of $\tilde{p}$, we have $\sum_{\mathbf{x} \in \mathcal{X}_{\geq n}} \tilde{p}(\mathbf{x}) \leq \sum_{\mathbf{x} \in \mathcal{X}_{\geq n}} p(\mathbf{x})$. The right-hand side is equal to $1 - \hat{Z}_n$, which by construction is less than ϵ. Therefore, by substituting to get $Z_{\tilde{p}} - \hat{Z}_{\tilde{p}}(\epsilon) \leq 1 - \hat{Z}_n$ and using the transitive property of inequality, we have that $Z_{\tilde{p}} - \hat{Z}_{\tilde{p}}(\epsilon) < \epsilon$. Combined with the previous paragraph's result, we have shown that $|Z_{\tilde{p}} - \hat{Z}_{\tilde{p}}(\epsilon)| \leq \epsilon$.

□

Theorem 5.6.1 implies that conditionalization operations on p, which remove strings from the support of p to get weighted language $\tilde{p}$, result in a computable $Z_{\tilde{p}}$ (as long as we can decide which strings are removed); and such a $\tilde{p}$ is therefore not subjected to the limitations of Theorem 5.3.2.

Theorem 5.6.1 also implies that $\tilde{p}_{\text{ELN}}$ we have just described above has a computable partition function:

Theorem 5.6.2. *Let Z be the partition function of $\tilde{p}_{\text{ELN}}$. Z is computable.*

Proof. Let $p(\mathbf{x}) = 1/3^{|\mathbf{x}|+1}$. $p \in \text{ELN} \subset \text{LN}$ because $p(\cdot \mid \hat{\mathbf{x}}) = 1/3$ for all valid prefixes $\hat{\mathbf{x}}$. Since $\tilde{p}_{\text{ELN}} \in \text{EC}$ by Lin et al. (2021, Theorem 5) and $\forall \mathbf{x} \in \mathbb{B}^*, \tilde{p}_{\text{ELN}}(\mathbf{x}) \leq p(\mathbf{x})$, we have Z is

computable by Theorem 5.6.1. □

Similarly, the 'sparse version' of $\tilde{p}_{\text{ELN}}$ introduced in (Lin et al., 2021, Theorem 6) can be shown to have a computable partition function as well.

Since one of our goals is to clearly demonstrate that we can construct a single weighted language ∈ EC that has an uncomputable partition function assuming ZFC (*e.g.*, $\tilde{b}$ in §5.2.2), we define weighted languages in $\mathcal{G}_k$ to have at most one 'high' weight, as opposed to $\tilde{p}$ in Lin et al. (2021, Theorem 5), where each $\mathbf{x}^{(1)}$ that encodes a halting machine has a 'high weight' suffix $\mathbf{x}^{(2)}$.

In fact, under our construction we can directly show EC ≠ ELN, using the uncomputability of $\tilde{b}$ from Theorem 5.2.2:

Corollary 5.6.2.1 (EC ≠ ELN under ZFC; slightly weaker version of Theorem 5 in (Lin et al., 2021))**.** *Assuming the axiomatic system of ZFC,* EC ≠ ELN.

Proof. We know there exists a weighted language $\tilde{b} \in$ EC (§5.2.2) that has an uncomputable partition function. Since $\tilde{b} \in$ EC, $\tilde{b}(\mathbf{x}) \in \mathbb{Q}_{\geq 0}, \forall \mathbf{x} \in \mathbb{B}^*$. Therefore for all strings $\mathbf{x} \in \mathbb{B}^*$, $b(\mathbf{x}) = {}^{\tilde{b}(\mathbf{x})}/_{Z_{\tilde{b}}}$ is an uncomputable number.

Assuming to the contrary that $\tilde{b} \in$ ELN. By definition $b(\mathbf{x}) = \prod b(x_t \mid \mathbf{x}_{<t})$, where each $b(x_t \mid \mathbf{x}_{<t}) \in \mathbb{Q}_{\geq}$ and is computable. Since the set of computable numbers is closed under multiplication, $b(\mathbf{x})$ would also be computable, which contradicts our assumption. Therefore $\tilde{b} \neq$ ELN, which implies EC ≠ ELN. □

Unlike the 'finite support' fix, an autoregressively parametrized (or subsequently

conditionalized) $\tilde{p}$ can have an infinite support. A conditionalized $\tilde{p}$ can have an intractable (but computable) partition function, and they are still subject to the expressive limitations imposed on LN languages: namely there is an EC language whose string weight rankings cannot be honored by any such conditionalized $\tilde{p}$ (Lin et al., 2021).

$\tilde{p}$ as low-treewidth factor graph grammars. Finally, we may limit ourselves to weighted languages defined by low-treewidth factor graph grammars (Chiang and Riley, 2020). Factor graph grammars generalize factor graphs, which cover many classes of graphical sequence models, such as n-gram, HMM, and whole-sentence language models (Jelinek, 1980; Kuhn, Niemann, and Schukat-Talamazzini, 1994; Rosenfeld, Chen, and Zhu, 2001), linear CRF models (Lafferty, McCallum, and Pereira, 2001), and weighted FSAs in general (Dreyer and Eisner, 2009): a factor graph grammar describes a (possibly infinite) set of factor graphs, generated from a hyperedge replacement graph grammar.

Assuming that an FGG G contains at most one n-observed-node factor graph for all $n \in \mathbb{N}$, it then defines a weighted language $\tilde{p}_G(\mathbf{x}) = \prod_{\psi \in \Psi_{|\mathbf{x}|}} \psi$, where **factor** ψ is a positive function of nodes. The treewidth of an FGG $W(G)$ is the maximum number of nodes any ψ can be a function of, and $\Psi_{|\mathbf{x}|}$ is the set of all factors of string length $|\mathbf{x}|$.

If $Z_{\tilde{p}_G} \in \mathbb{R}$ exists, it can be computed exactly by an algorithm, in time exponential in $W(G)$ following (Chiang and Riley, 2020). Exact computation of $Z_{\tilde{p}_G}$ may be manageable as long as $W(G)$ is small, which would allow us to exactly compute held-out data likelihood, and also train with a (marginalized) MLE objective function. However, limiting $W(G)$ directly limits the expressiveness of $\tilde{p}$.

5.7 Related work

Turing completeness of formal languages and associated uncomputability issues emerge repeatedly in computer science. For example, Turing completeness may emerge as an unwanted side effect in programming languages, since it implies undecidability. One of the best known examples is the Turing completeness of the C++ grammar (Veldhuizen, 2003; Josh, 2013), which makes both parsing and compiling C++ programs undecidable. Similar problems exist for Java (Grigore, 2016) and Haskell with (unrestricted) instance declarations (Wansbrough, 1998). Another example is the (possibly inadvertently introduced) Turing completeness of the page fault handling mechanism on modern computer systems, which raises security concerns in the context of trusted computing (Bangert et al., 2013).

Our work is not the first to discuss the consequences of computability in machine learning: assuming we can acquire training data from an oracle, under a supervised learning setting, recognition of an undecidable language is PAC-learnable (Lathrop, 1996). (Agarwal et al., 2020) extended the definition of PAC learnability (Valiant, 1984) to computable learners. By contrast, we are focused on the computability of EBMs for sequences, such as a language model as a component of a larger system for automatic speech recognition. Designing an appropriate, efficient loss functional is a challenge that several prior works have compared. With the plethora of learning strategies for EBMs, it is untenable to point out the deficiency in each. Table 5.2 gives a handful of examples; none share the four properties we desire:

Technique	Energy-based (*i.e.*, globally normalized)	Infinite language support	Scoring function has unbounded treewidth	Consistency
Noise-contrastive estimation ((Ma and Collins, 2018); used in (Lin et al., 2021; Bakhtin et al., 2021))	✓	✗	✓	✓
MLE with variable elimination in factor graph grammars ((Chiang and Riley, 2020); used in (Eisner, 2001; Finkel, Kleeman, and Manning, 2008), *inter alia*)	✓	✓	✗	✓
MLE with autoregressive parametrization (Mikolov et al., 2010)	✗	✓	✓	✓
Contrastive divergence (Hinton, 2002)	✓	✓	✓	✗
Contrastive estimation (Smith and Eisner, 2005)	✓	✓	✓	✗

Table 5.2: Deficiencies of some common alternatives to overly expressive EBMs.

1. Global normalization (without which the model would be in LN)

2. Support over infinite languages (of finite strings)

3. Unbounded treewidth in the function assigning weights to strings

4. Estimator consistency (*i.e.*, asymptotic guarantee to recover the true parameters).

In § 3 we noted autoregressive factors of EC languages can be uncomputable (see also Theorem 5.6.2). We also noted that a weighted language can have an uncomputable partition function (presumably resulting from the sum over infinitely many string weights). But we did not dwell on the question whether such a weighted language could lie within the EC class, much less providing a constructive proof (see also Corollary 5.6.2.1). Instead, we emphasized that under the assumption that oracular access to trained parameter strings is possible, arbitrarily good approximations of the (possibly uncomputable) partition function

can be memorized in the (autoregressive) model parameters. There is an interesting constrast between the stances of these two chapters: in § 3 we saw the uncomputability of Z as a trivial issue from the model capacity viewpoint, since good approximations take few bits in the parameters to store. On the other hand, we see that the uncomputability problem can be a parameter estimation disaster — there will be no guarantee of good approximations can be found in finite time at all.

5.8 Conclusion and future work

Energy-based models are posed as an efficient tool for decision problems, circumventing probabilities and expensive normalization (LeCun et al., 2006). Extending this vision to generic sequence models, though, can involve complexity/computability problems that are difficult, or even impossible. We've shown that as energy-based sequence models become more powerful, the partition function becomes uncomputable — even when we are restricted to polytime-computable weighted languages. More precisely, we have shown in this chapter that exact estimators, even if randomized, cannot have accuracy guarantees (§5.3). We also show that we will not be able to derive some popular asymptotic estimators either (§§5.4.1 and 5.4.2). Parameter identifiability is undecidable, which makes formally proving estimator consistency impossible as well (§5.5.1). Furthermore, deciding which model achieves lower held-out likelihood is generally impossible, even if we limit ourselves to fixed-size sequence model families (§5.5.3).

This chapter continues a discussion started by § 3, who posture energy-based models as a more powerful alternative to autoregressive sequence models. Autoregressive sequence models, after all, have wide adoption and empirical successes (Radford et al., 2019; Brown et al., 2020). By contrast, more general neural energy-based sequence models have not caught on. Why not? We give *unlearnability* — due to *uncomputability* — as a possible explanation: unless we give up the ability to learn parameters from data, we likely cannot use the full expressiveness afforded by powerful neural networks. Just like the model capacity problems brought up in § 3, this result is independent of the amount of training data.

Nonetheless, we emphasize that our results do not invalidate the findings of § 3: regardless of the actual neural parametrization, autoregressive models can never capture certain distributions that energy-based models can. Instead, one of our main messages is that *we may not be able to find those EBM parameters in finite time, if we do not know what the parameters are*. Of course, if we know the task perfectly well and can in fact manually assign the model parameters, we will not need to learn from data at all. The middle ground — when we have *some* prior knowledge about the task, but cannot really design the parameter vectors — is an interesting direction for future work: the three palliative alternatives outlined in §5.6 do not take task-specific information into account at all. Can we do better than that, without suffering uncomputability problems?

There are also some other possible directions for future work: for example, while we have shown that any subfamily of EC that contains certain pathological parameter

vectors will have undecidable problems when it comes to comparing model performance on held-out data likelihood, we stopped short at identifying a subfamily of EC that does not suffer from this problem, while being more expressive than LN. Another possible research direction is formally proving/disproving that *every* estimator will stop being useful on overly expressive EBMs: while we have looked into some well-known estimators in Table 5.2, the list is far from exhaustive. For example, one may consider an extension of noise-contrastive estimation for weighted languages that have an infinite support. Would such a method *work*? However, we again point out the impossibility of confidently choosing a good model using held-out data will remain a problem, even if such an estimator were to exist.

Results from this chapter are largely theoretical. However, they do justify a preference of autoregressive latent-variable sequence models (§3.4.2), over the other expressive sequence model families identified in §3.4, as a more practical expressive weighted language class: unlike energy-based sequence models (including energy-based latent-variable sequence models), autoregressive latent-variable sequence models have computable partition functions. Their conditional distributions also have computable partition functions due to Theorem 5.6.1, as long as it is decidable to tell whether a string is within support. Our parametrization of neural finite-state machines (§ 7) is based on this observation.

Chapter 6

Amortized inference with neural particle smoothing

Earlier in this thesis we have discussed expressiveness problems of sequence model families (§§ 3–5). In this chapter, we focus on a specific inference problem: sampling from *conditional* sequence distributions. Such distributions arise when we fix some variables in a joint distribution to known values. Many common machine learning techniques and applications require sampling from such conditional distributions (a more thorough exploration is in §6.1.3).

There are well-known generic Monte Carlo methods for sampling from such distributions (Doucet and Johansen, 2009). In this chapter, we describe **neural particle smoothing**, a class of specialized Monte Carlo methods that are more sample-efficient, by 1) learning 'smarter' heuristics that better match the target distribution, and 2) by leveraging the

monotonic alignment between observed and unobserved variables.

6.1 Introduction

Many structured prediction problems in NLP can be reduced to labeling a length-T input string $\mathbf{x}$ with a length-T sequence $\mathbf{y}$ of tags. In some cases, these tags are annotations such as syntactic parts of speech. In other cases, they represent actions that incrementally build an output structure: IOB tags build a chunking of the input (Ramshaw and Marcus, 1999), shift-reduce actions build a tree (Yamada and Matsumoto, 2003), and finite-state transducer arcs build an output string (Pereira and Riley, 1997).

One may wish to score the possible taggings using a recurrent neural network, which can learn to be sensitive to complex patterns in the training data. A globally normalized conditional probability model is particularly valuable because it quantifies uncertainty and does not suffer from label bias (Lafferty, McCallum, and Pereira, 2001); also, such models often arise as the predictive conditional distribution $p(\mathbf{y} \mid \mathbf{x})$ corresponding to some well-designed generative model $p(\mathbf{x}, \mathbf{y})$ for the domain. In the neural case, however, inference in such models becomes intractable. It is hard to know what the model actually predicts and hard to compute gradients to improve its predictions.

In such intractable settings, one generally falls back on approximate inference or sampling. In the NLP community, beam search and importance sampling are common. Unfortunately, beam search considers only the approximate-top-k taggings from an expo-

nential set (Wiseman and Rush, 2016), and importance sampling requires the construction of a good proposal distribution (Dyer et al., 2016).

In this work we exploit the sequential structure of the tagging problem to do *sequential* importance sampling, which resembles beam search in that it constructs its proposed samples incrementally — one tag at a time, taking the actual model into account at every step. This method is known as particle filtering (Doucet and Johansen, 2009). We extend it here to take advantage of the fact that the sampler has access to the entire input string as it constructs its tagging, which allows it to look ahead or — as we will show — to use a neural network to approximate the effect of lookahead. Our resulting method is called neural particle smoothing.

6.1.1 What this chapter provides

For $\mathbf{x} = x_1 \cdots x_T$, let $\mathbf{x}_{:t}$ and $\mathbf{x}_{t:}$ respectively denote the prefix $x_1 \cdots x_t$ and the suffix $x_{t+1} \cdots x_T$.

We develop *neural particle smoothing* — a sequential importance sampling method which, given a string $\mathbf{x}$, draws a sample of taggings $\mathbf{y}$ from $p_\theta(\mathbf{y} \mid \mathbf{x})$, where $|\mathbf{y}| = |\mathbf{x}| = T$. Our method works for any conditional probability model of the quite general form[1]

$$p_\theta(\mathbf{y} \mid \mathbf{x}) \stackrel{\text{def}}{\propto} \exp G_T \tag{6.1}$$

where G is an *incremental stateful global scoring model* that recursively defines scores G_t

[1] A model may require for convenience that each input end with a special end-of-sequence symbol: that is, $x_T = \text{EOS}$.

of prefixes of $(\mathbf{x}, \mathbf{y})$ at all times $0 \le t \le T$:[2]

$$G_t \triangleq G_{t-1} + g_\theta(\mathbf{s}_{t-1}, x_t, y_t) \qquad \text{(with } G_0 \triangleq 0\text{)} \tag{6.2}$$

$$\mathbf{s}_t \triangleq f_\theta(\mathbf{s}_{t-1}, x_t, y_t) \qquad \text{(with } \mathbf{s}_0 \text{ given)} \tag{6.3}$$

These quantities implicitly depend on $\mathbf{x}, \mathbf{y}$ and θ. Here $\mathbf{s}_t$ is the model's *state* after observing the pair of length-t prefixes $(\mathbf{x}_{:t}, \mathbf{y}_{:t})$. G_t is the *score-so-far* of this prefix pair, while $G_T - G_t$ is the *score-to-go*. The state $\mathbf{s}_t$ summarizes the prefix pair in the sense that the score-to-go depends only on $\mathbf{s}_t$ and the length-$(T-t)$ suffixes $(\mathbf{x}_{t:}, \mathbf{y}_{t:})$. The *local scoring function* g_θ and *state update function* f_θ may be any functions parameterized by θ — perhaps neural networks. We assume θ is fixed and given.

This model family is expressive enough to capture any desired $p(\mathbf{y} \mid \mathbf{x})$. Why? Take any distribution $p(\mathbf{x}, \mathbf{y})$ with this desired conditionalization (e.g., the true joint distribution) and factor it as

$$\begin{aligned} \log p(\mathbf{x}, \mathbf{y}) &= \textstyle\sum_{t=1}^{T} \log p(x_t, y_t \mid \mathbf{x}_{:t-1}, \mathbf{y}_{:t-1}) \\ &= \textstyle\sum_{t=1}^{T} \underbrace{\log p(x_t, y_t \mid \mathbf{s}_{t-1})}_{\text{use as } g_\theta(\mathbf{s}_{t-1}, x_t, y_t)} = G_T \end{aligned} \tag{6.4}$$

by making $\mathbf{s}_t$ include as much information about $(\mathbf{x}_{:t}, \mathbf{y}_{:t})$ as needed for (6.4) to hold (possibly $\mathbf{s}_t = (\mathbf{x}_{:t}, \mathbf{y}_{:t})$).[3] Then by defining g_θ as shown in (6.4), we get $p(\mathbf{x}, \mathbf{y}) = \exp G_T$

[2] Note that we parametrize G_t and $\mathbf{s}_t$ to take the new increments (x_t, y_t), rather than prefixes $(\mathbf{x}_{:t+1}, \mathbf{y}_{:t+1})$ as arguments at time step t, since both functions can already have access to $(\mathbf{x}_{:t-1}, \mathbf{y}_{:t-1})$ through $\mathbf{s}_{t-1}$.

[3] Furthermore, $\mathbf{s}_t$ could even depend on all of $\mathbf{x}$ (if $\mathbf{s}_0$ does), allowing direct expression of models such as stacked BiRNNs.

and thus (6.1) holds for each $\mathbf{x}$.

6.1.2 Relationship to particle filtering

Our method is spelled out in §6.4 (one may look now). It is a variant of the popular *particle filtering* method that tracks the state of a physical system in discrete time (Ristic, Arulampalam, and Gordon, 2004). Our particular *proposal distribution* for y_t can be found in equations (6.5), (6.6), (6.28) and (6.29). It considers not only past observations $\mathbf{x}_{:t}$ as reflected in $\mathbf{s}_{t-1}$, but also future observations $\mathbf{x}_{t:}$, as summarized by the state $\bar{\mathbf{s}}_t$ of a right-to-left recurrent neural network $\bar{f}$ that we will train:

$$\hat{H}_t \triangleq h_\phi(\bar{\mathbf{s}}_{t+1}, x_{t+1}) + \hat{H}_{t+1} \tag{6.5}$$

$$\bar{\mathbf{s}}_t \triangleq \bar{f}_\phi(\bar{\mathbf{s}}_{t+1}, x_{t+1}) \qquad \text{(with } \mathbf{s}_T \text{ given)} \tag{6.6}$$

Conditioning the distribution of y_t on future observations $\mathbf{x}_{t:}$ means that we are doing "smoothing" rather than "filtering" (in signal processing terminology). Doing so can reduce the bias and variance of our sampler. It is possible so long as $\mathbf{x}$ is provided in its entirety before the sampler runs — which is often the case in NLP.

6.1.3 Applications

Why sample from p_θ at all? Many NLP systems instead simply search for the *Viterbi sequence* $\mathbf{y}$ that maximizes G_T and thus maximizes $p_\theta(\mathbf{y} \mid \mathbf{x})$. If the space of states $\mathbf{s}$ is small,

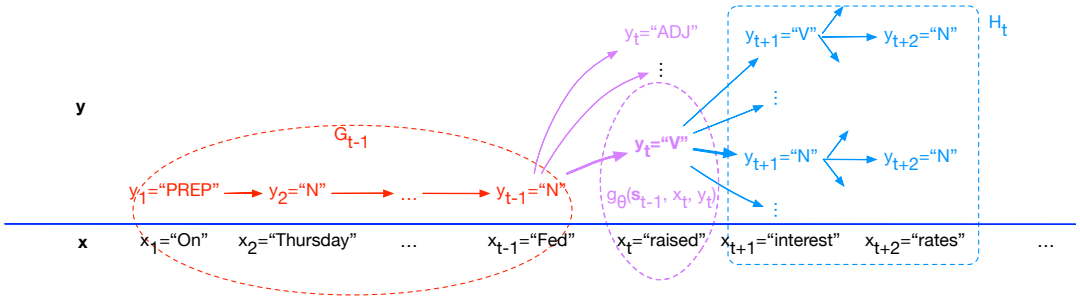


Figure 6.1: Sampling a single particle from a tagging model. $y_1, \ldots, y_{t-1}$ (orange) have already been chosen, with a total model score of G_{t-1}, and now the sampler is constructing a proposal distribution q (purple) from which the next tag y_t will be sampled. Each y_t is evaluated according to its contribution to G_t (namely g_θ) and its future score H_t (blue). The figure illustrates quantities used throughout this chapter, beginning with exact sampling in equations (6.8)–(6.13). Our main idea (§6.3) is to *approximate* the H_t computation (a log-sum-exp over exponentially many sequences) when exact computation by dynamic programming is not an option. The form of our approximation uses a right-to-left recurrent neural network but is *inspired* by the exact dynamic programming algorithm.

this can be done efficiently by dynamic programming (Viterbi, 1967); if not, then A^* may be an option (see §6.2). More common is to use an approximate method: beam search, or perhaps a sequential prediction policy trained with reinforcement learning. Past work has already shown how to improve these approximate search algorithms by conditioning on the future (Bahdanau et al., 2017; Wiseman and Rush, 2016).

Sampling is essentially a generalization of maximization: sampling from $\exp \frac{G_T}{\text{temperature}}$ approaches maximization as temperature $\longrightarrow 0$. It is a fundamental building block for other algorithms, as it can be used to take expectations over the whole space of possible y values. Below we review how sampling is crucially used in minimum-risk decoding, supervised training, unsupervised training, imputation of missing data, pipeline decoding, and inference in graphical models.

6.1.3.1 Minimum-risk decoding

Given a loss function and $\mathbf{x}$, *minimum-risk decoding* seeks the output

$$\operatorname*{argmin}_{\mathbf{z}} \sum_{\mathbf{y}} p_\theta(\mathbf{y} \mid \mathbf{x}) \cdot \text{loss}(\mathbf{z} \mid \mathbf{y}) \tag{6.7}$$

In the special case where $\text{loss}(\mathbf{z} \mid \mathbf{y})$ simply asks whether $\mathbf{z} \neq \mathbf{y}$, this simply returns the "Viterbi" sequence $\mathbf{y}$ that maximises $p_\theta(\mathbf{y} \mid \mathbf{x})$. However, it may give a different answer if the loss function gives partial credit (when $\mathbf{z} \approx \mathbf{y}$), or if the space of outputs $\mathbf{z}$ is simply coarser than the space of taggings $\mathbf{y}$ — for example, if there are many action sequences $\mathbf{y}$ that could build the same output structure $\mathbf{z}$. In these cases, the optimal $\mathbf{z}$ may win due to the combined support of many suboptimal $\mathbf{y}$ values, and so finding the optimal $\mathbf{y}$ (the Viterbi sequence) is not enough to determine the optimal $\mathbf{z}$.

The risk objective (6.7) is a expensive expectation under the distribution $p_\theta(\mathbf{y} \mid \mathbf{x})$. To approximate it, one can replace $p_\theta(\mathbf{y} \mid \mathbf{x})$ with an approximation $\hat{p}(\mathbf{y})$ that has small support so that the summation is efficient. Particle smoothing returns such a $\hat{p}$ — a non-uniform distribution (6.31) over M particles. Since those particles are randomly drawn, $\hat{p}$ is itself stochastic, but $\mathbb{E}[\hat{p}(\mathbf{y})] \approx p_\theta(\mathbf{y} \mid \mathbf{x})$, with the approximation improving with the quality of the proposal distribution (which is the focus of this work).

6.1.3.2 Supervised training

In *supervised* training of the model (6.1) by maximizing conditional log-likelihood, the gradient of $\log p(\mathbf{y}^* \mid \mathbf{x})$ on a single training example $(\mathbf{x}, \mathbf{y}^*)$ is $\nabla_\theta \log p_\theta(\mathbf{y}^* \mid \mathbf{x}) = \nabla_\theta G_T^* - \sum_{\mathbf{y}} p_\theta(\mathbf{y} \mid \mathbf{x}) \cdot \nabla_\theta G_T$. The sum is again an expectation that can be estimated by using $\hat{p}$. Since $\mathbb{E}[\hat{p}(\mathbf{y})] \approx p_\theta(\mathbf{y} \mid \mathbf{x})$, this yields a stochastic estimate of the gradient that can be used in the stochastic gradient ascent algorithm (Robbins and Monro, 1951).[4]

6.1.3.3 Unsupervised training

In *unsupervised or semi-supervised training* of a generative model $p_\theta(\mathbf{x}, \mathbf{y})$, one has some training examples where $\mathbf{y}^*$ is unobserved or observed incompletely (e.g., perhaps only $\mathbf{z}$ is observed). The Monte Carlo EM algorithm for estimating θ (Wei and Tanner, 1990) replaces the missing $\mathbf{y}^*$ with samples from $p_\theta(\mathbf{y} \mid \mathbf{x}, \text{partial observation})$ (this is the Monte Carlo "E step"). This *multiple imputation* procedure has other uses as well in statistical analysis with missing data (Little and Rubin, 1987).

[4]Notice that the gradient takes this "difficult" form only because the model is globally normalized. If we were training a locally normalized conditional model (McCallum, Freitag, and Pereira, 2000), or a locally normalized joint model like equation (6.4), then sampling methods would not be needed, because the gradient of the (conditional or joint) log-likelihood would decompose into T "easy" summands that each involve an expectation over the small set of y_t values for some t, rather than over the exponentially larger set of strings $\mathbf{y}$. However, this simplification goes away outside the fully supervised case, as the next paragraph discusses.

6.1.3.4 Inference in graphical models

Modular architectures provide another use for sampling. If $p_\theta(\mathbf{y} \mid \mathbf{x})$ is just one stage in an NLP annotation *pipeline*, Finkel, Manning, and Ng (2006) recommend passing a diverse sample of $\mathbf{y}$ values on to the next stage, where they can be further annotated and rescored or rejected. More generally, in a *graphical model* that relates multiple strings (Bouchard-Côté et al., 2007; Dreyer and Eisner, 2009; Cotterell, Sylak-Glassman, and Kirov, 2017), inference could be performed by particle belief propagation (Ihler and McAllester, 2009; Lienart, Teh, and Doucet, 2015), or with the help of stochastic-inverse proposal distributions (Stuhlmüller, Taylor, and Goodman, 2013). These methods call conditional sampling as a subroutine.

6.2 Exact sequential sampling

To develop our method, it is useful to first consider exact samplers. Exact sampling is tractable for only some of the models allowed by §6.1.1. However, the form and notation of the exact algorithms in §6.2 will guide our development of approximations in §6.3.

An *exact sequential sampler* draws y_t from $p_\theta(y_t \mid \mathbf{x}, \mathbf{y}_{:t-1})$ for each $t = 1, \ldots, T$ in sequence. Then $\mathbf{y}$ is exactly distributed according to $p_\theta(\mathbf{y} \mid \mathbf{x})$.

For each given $\mathbf{x}, \mathbf{y}_{:t-1}$, observe that

$$p_\theta(y_t \mid \mathbf{x}, \mathbf{y}_{:t-1}) \tag{6.8}$$

$$\propto p_\theta(\mathbf{y}_{:t} \mid \mathbf{x}) = \textstyle\sum_{\mathbf{y}_{t:}} p_\theta(\mathbf{y} \mid \mathbf{x}) \tag{6.9}$$

$$\propto \textstyle\sum_{\mathbf{y}_{t:}} \exp G_T \tag{6.10}$$

$$= \exp\,(G_t + \underbrace{\log \textstyle\sum_{\mathbf{y}_{t:}} \exp\,(G_T - G_t)}_{\text{call this } H_t}) \tag{6.11}$$

$$= \exp\,(G_{t-1} + g_\theta(\mathbf{s}_{t-1}, x_t, y_t) + H_t) \tag{6.12}$$

$$\propto \exp\,(g_\theta(\mathbf{s}_{t-1}, x_t, y_t) + H_t) \tag{6.13}$$

Thus, we can easily construct the needed distribution (6.8) by normalizing (6.13) over all possible values of y_t. The challenging part of (6.13) is to compute H_t: as defined in (6.11), H_t involves a sum over exponentially many futures $\mathbf{y}_{t:}$. (See Figure 6.1.)

We chose the symbols G and H in homage to the A^* search algorithm (Hart, Nilsson, and Raphael, 1968). In that algorithm (which could be used to find the Viterbi sequence), g denotes the score-so-far of a partial solution $\mathbf{y}_{:t}$, and h denotes the optimal score-to-go. Thus, $g + h$ would be the score of the *best* sequence with prefix $\mathbf{y}_{:t}$. Analogously, our $G_t + H_t$ is the log of the total exponentiated scores of *all* sequences with prefix $\mathbf{y}_{:t}$. G_t and H_t might be called the *logprob-so-far* and *logprob-to-go* of $\mathbf{y}_{:t}$.

Just as A^* approximates h with a "heuristic" $\hat{h}$, the next section will approximate H_t using a neural estimate $\hat{H}_t$ (equations (6.5)–(6.6)). However, the specific form of our approximation

is inspired by cases where H_t can be computed exactly. We consider those in the remainder of this section.

6.2.1 Exact sampling from HMMs

A *hidden Markov model* (HMM) specifies a normalized *joint* distribution $p_\theta(\mathbf{x}, \mathbf{y}) = \exp G_T$ over state sequence $\mathbf{y}$ and observation sequence $\mathbf{x}$,[5] Thus the posterior $p_\theta(\mathbf{y} \mid \mathbf{x})$ is proportional to $\exp G_T$, as required by equation (6.1).

The HMM specifically defines G_T by equations (6.2)–(6.3) with $\mathbf{s}_t = y_t$ and $g_\theta(\mathbf{s}_{t-1}, x_t, y_t) = \log p_\theta(y_t \mid y_{t-1}) + \log p_\theta(x_t \mid y_t)$.[6]

In this setting, H_t can be computed exactly by the *backward algorithm* (Rabiner, 1989). By the definition of H_t in equation (6.11),

$$\begin{aligned} \exp H_t &= \sum_{\mathbf{y}_{t:}} \exp\left(G_T - G_t\right) \qquad = \sum_{\mathbf{y}_{t:}} \exp \sum_{j=t+1}^{T} g_\theta(\mathbf{s}_{j-1}, x_j, y_j) \qquad (6.14) \\ &= \sum_{\mathbf{y}_{t:}} \prod_{j=t+1}^{T} p_\theta(y_j \mid y_{j-1}) \cdot p_\theta(x_j \mid y_j) \\ &= (\boldsymbol{\beta}_t)_{y_t} \quad \text{(backward prob of } y_t \text{ at time } t\text{)} \end{aligned}$$

[5]The HMM actually specifies a distribution over a pair of infinite sequences, but here we consider the marginal distribution over just the length-T prefixes.

[6]It defines $\mathbf{s}_0 = \text{BOS}$, a beginning-of-sequence symbol, so $p_\theta(y_1 \mid \text{BOS})$ specifies the initial state distribution.

where the vector $\boldsymbol{\beta}_t$ is defined by base case $(\boldsymbol{\beta}_T)_y = 1$ and for $0 \le t < T$ by the recurrence

$$\begin{aligned}(\beta_t)_y &\triangleq \sum_{\mathbf{y}_{t:}} p_\theta(\mathbf{x}_{t:}, \mathbf{y}_{t:} \mid y_t = y) \\ &= \sum_{y'} p_\theta(y' \mid y) \cdot p_\theta(x_{t+1} \mid y') \cdot (\boldsymbol{\beta}_{t+1})_{y'}\end{aligned} \tag{6.15}$$

6.2.2 Exact sampling from OOHMMs

For sequence tagging, a weakness of (first-order) HMMs is that the model state $\mathbf{s}_t = y_t$ may contain little information: only the most recent tag y_t is remembered, so the number of possible model states $\mathbf{s}_t$ is limited by the vocabulary of output tags.

We may generalize so that the data generating process is in a latent state $u_t \in \{1, \dots, k\}$ at each time t, and the observed y_t — along with x_t — is generated from u_t. Now k may be arbitrarily large. The model has the form

$$\begin{aligned}p_\theta(\mathbf{x}, \mathbf{y}) &= \exp G_T \\ &= \sum_{\mathbf{u}} \prod_{t=1}^{T} p_\theta(u_t \mid u_{t-1}) \cdot p_\theta(x_t, y_t \mid u_t)\end{aligned} \tag{6.16}$$

This is essentially a pair HMM (Knudsen and Miyamoto, 2003) without insertions or deletions, also known as an "ϵ-free" or "same-length" probabilistic finite-state transducer. We refer to it here as an *output-output HMM* (OOHMM).[7]

[7] This is by analogy with the *input-output HMM* (IOHMM) of Bengio and Frasconi (1996), which defines $p(\mathbf{y} \mid \mathbf{x})$ directly and conditions the transition to u_t on x_t. The OOHMM instead defines $p(\mathbf{y} \mid \mathbf{x})$ by conditionalizing (6.16) — which avoids the *label bias* problem (Lafferty, McCallum, and Pereira, 2001) that in the IOHMM, y_t is independent of future input $\mathbf{x}_{t:}$ (given the past input $\mathbf{x}_{:t}$).

Is this still an example of the general model architecture from §6.1.1? Yes. Since u_t is latent and evolves stochastically, it cannot be used as the state $\mathbf{s}_t$ in equations (6.2)–(6.3) or (6.4). However, we *can* define $\mathbf{s}_t$ to be the model's *belief state* after observing $(\mathbf{x}_{:t}, \mathbf{y}_{:t})$. The belief state is the posterior probability distribution over the underlying state u_t of the system. That is, $\mathbf{s}_t$ deterministically keeps track of all possible states that the OOHMM might be in — just as the state of a determinized FSA keeps track of all possible states that the original nondeterministic FSA might be in.

We may compute the belief state in terms of a vector of *forward probabilities* that starts at $\boldsymbol{\alpha}_0$,

$$(\boldsymbol{\alpha}_0)_u \triangleq \begin{cases} 1 & \text{if } u = \textsc{bos} \text{ (see footnote 6)} \\ 0 & \text{if } u = \text{any other state} \end{cases} \tag{6.17}$$

and is updated deterministically for each $0 < t \leq T$ by the *forward algorithm* (Rabiner, 1989):

$$(\boldsymbol{\alpha}_t)_u \triangleq \sum_{u'=1}^{k} (\boldsymbol{\alpha}_{t-1})_{u'} \cdot p_\theta(u \mid u') \cdot p_\theta(x_t, y_t \mid u) \tag{6.18}$$

$(\boldsymbol{\alpha}_t)_u$ can be interpreted as the logprob-so-far *if* the system is in state u after observing $(\mathbf{x}_{:t}, \mathbf{y}_{:t})$. We may express the update rule (6.18) by $\boldsymbol{\alpha}_t^\top = \boldsymbol{\alpha}_{t-1}^\top P$ where the matrix P depends on (x_t, y_t), namely $P_{u'u} \triangleq p_\theta(u \mid u') \cdot p_\theta(x_t, y_t \mid u)$.

The belief state $\mathbf{s}_t \triangleq [\![\boldsymbol{\alpha}_t]\!] \in \mathbb{R}^k$ simply normalizes $\boldsymbol{\alpha}_t$ into a probability vector, where

$[\![\mathbf{u}]\!] \triangleq \mathbf{u}/(\mathbf{u}^\top \mathbf{1})$ denotes the *normalization operator*. The state update (6.18) now takes the form (6.3) as desired, with f_θ a normalized vector-matrix product:

$$\mathbf{s}_t^\top = f_\theta(\mathbf{s}_{t-1}, x_t, y_t) \triangleq [\![\mathbf{s}_{t-1}^\top P]\!] \tag{6.19}$$

As in the HMM case, we define G_t as the log of the generative prefix probability,

$$G_t \triangleq \log p_\theta(\mathbf{x}_{:t}, \mathbf{y}_{:t}) = \log \textstyle\sum_u (\boldsymbol{\alpha}_t)_u \tag{6.20}$$

which has the form (6.2) as desired if we put

$$\begin{aligned} g_\theta(\mathbf{s}_{t-1}, x_t, y_t) &\triangleq G_t - G_{t-1} \\ &= \log \frac{\boldsymbol{\alpha}_{t-1}^\top P \mathbf{1}}{\boldsymbol{\alpha}_{t-1}^\top \mathbf{1}} = \log\, (\mathbf{s}_{t-1}^\top P \mathbf{1}) \end{aligned} \tag{6.21}$$

Again, exact sampling is possible. It suffices to compute (6.10). For the OOHMM, this is given by

$$\textstyle\sum_{\mathbf{y}_{t:}} \exp G_T = \boldsymbol{\alpha}_t^\top \boldsymbol{\beta}_t \tag{6.22}$$

where $\boldsymbol{\beta}_T \triangleq \mathbf{1}$ and the *backward algorithm*

$$\begin{aligned}(\boldsymbol{\beta}_t)_u &\triangleq p_\theta(\mathbf{x}_{t:} \mid u_t = u) \\ &= \sum_{\mathbf{u}_{t:}, \mathbf{y}_{t:}} p_\theta(\mathbf{u}_{t:}, \mathbf{x}_{t:}, \mathbf{y}_{t:} \mid u_t = u) \\ &= \sum_{u'} \underbrace{p_\theta(u' \mid u) \cdot p(x_{t+1} \mid u')}_{\text{call this } \bar{P}_{uu'}} \cdot (\boldsymbol{\beta}_{t+1})_{u'}\end{aligned} \tag{6.23}$$

for $0 \le t < T$ uses dynamic programming to find the total probability of all ways to generate the future observations $\mathbf{x}_{t:}$. Note that $\boldsymbol{\alpha}_t$ is defined for a *specific prefix* $\mathbf{y}_{:t}$ (though it sums over all $\mathbf{u}_{:t}$), whereas $\boldsymbol{\beta}_t$ sums over *all suffixes* $\mathbf{y}_{t:}$ (and over all $\mathbf{u}_{t:}$), to achieve the asymmetric summation in (6.22).[8]

Define $\bar{\mathbf{s}}_t \triangleq [\![\boldsymbol{\beta}_t]\!] \in \mathbb{R}^k$ to be a normalized version of $\boldsymbol{\beta}_t$. The $\boldsymbol{\beta}_t$ recurrence (6.23) can clearly be expressed in the form $\bar{\mathbf{s}}_t = [\![\bar{P}\bar{\mathbf{s}}_{t+1}]\!]$, much like (6.19).

6.2.3 The logprob-to-go for OOHMMs

Let us now work out the definition of H_t for OOHMMs (cf. equation (6.14) in §6.2.1 for HMMs). We will write it in terms of $\hat{H}_t$ from §6.1.2. Let us define $\hat{H}_t$ symmetrically to G_t (see (6.20)):

$$\hat{H}_t \triangleq \log \sum_u (\boldsymbol{\beta}_t)_u \quad (= \log \mathbf{1}^\top \boldsymbol{\beta}_t) \tag{6.24}$$

[8] Note that both $\boldsymbol{\alpha}_t$ and $\boldsymbol{\beta}_t$ look at a *specific* $\mathbf{x}_{:t}$.

which has the form (6.5) as desired if we put

$$h_\phi(\bar{\mathbf{s}}_{t+1}, x_{t+1}) \triangleq \hat{H}_t - \hat{H}_{t+1} = \log\left(\mathbf{1}^\top \overline{P} \bar{\mathbf{s}}_{t+1}\right) \tag{6.25}$$

From equations (6.11), (6.20), (6.22) and (6.24), we see

$$\begin{aligned} H_t &= \log\Big(\sum_{\mathbf{y}_{t:}} \exp G_T\Big) - G_t \\ &= \log \frac{\boldsymbol{\alpha}_t^\top \boldsymbol{\beta}_t}{(\boldsymbol{\alpha}_t^\top \mathbf{1})(\mathbf{1}^\top \boldsymbol{\beta}_t)} + \log\left(\mathbf{1}^\top \boldsymbol{\beta}_t\right) \\ &= \underbrace{\log \mathbf{s}_t^\top \bar{\mathbf{s}}_t}_{\text{call this } C_t} + \hat{H}_t \end{aligned} \tag{6.26}$$

where $C_t \in \mathbb{R}$ can be regarded as evaluating the *compatibility* of the state distributions $\mathbf{s}_t$ and $\bar{\mathbf{s}}_t$.

In short, the generic strategy (6.13) for exact sampling says that for an OOHMM, y_t is distributed as

$$\begin{aligned} p_\theta(y_t \mid \mathbf{x}, \mathbf{y}_{:t-1}) &\propto \exp\left(g_\theta(\mathbf{s}_{t-1}, x_t, y_t) + H_t\right) \\ &\propto \exp\Big(\underbrace{g_\theta(\mathbf{s}_{t-1}, x_t, y_t)}_{\text{depends on } \mathbf{x}_{:t}, \mathbf{y}_{:t}} + \underbrace{C_t}_{\text{on } \mathbf{x}, \mathbf{y}_{:t}} + \underbrace{\hat{H}_t}_{\text{on } \mathbf{x}_{t:}}\Big) \\ &\propto \exp\left(g_\theta(\mathbf{s}_{t-1}, x_t, y_t) + C_t\right) \end{aligned} \tag{6.27}$$

This is equivalent to choosing y_t in proportion to (6.22) — but we now turn to settings where it is infeasible to compute (6.22) exactly. There we will use the formulation (6.27)

but approximate C_t. For completeness, we will also consider how to approximate $\hat{H}_t$, which dropped out of the above distribution (because it was the same for all choices of y_t) but may be useful for other algorithms (see §6.4).

6.3 Neural modeling as approximation

6.3.1 Models with large state spaces

The expressivity of an OOHMM is limited by the number of states k. The state $u_t \in \{1, \ldots, k\}$ is a bottleneck between the past $(\mathbf{x}_{:t}, \mathbf{y}_{:t})$ and the future $(\mathbf{x}_{t:}, \mathbf{y}_{t:})$, in that past and future are *conditionally independent* given u_t. Thus, the mutual information between past and future is at most $\log_2 k$ bits.

In many NLP domains, however, the past seems to carry substantial information about the future. The first half of a sentence greatly reduces the uncertainly about the second half, by providing information about topics, referents, syntax, semantics, and discourse. This suggests that an accurate HMM language model $p(\mathbf{x})$ would require *very large* k — as would a generative OOHMM model $p(\mathbf{x}, \mathbf{y})$ of *annotated* language. The situation is perhaps better for discriminative models $p(\mathbf{y} \mid \mathbf{x})$, since much of the information for predicting $\mathbf{y}_{t:}$ might be available in $\mathbf{x}_{t:}$. Still, it is important to let $(\mathbf{x}_{:t}, \mathbf{y}_{:t})$ contribute enough additional information about $\mathbf{y}_{t:}$: even for short strings, making k too small (giving $\leq \log_2 k$ bits) may harm prediction (Dreyer, Smith, and Eisner, 2008).

Of course, (6.4) says that an OOHMM can express any joint distribution for which the mutual information is finite,[9] by taking k large enough for v_{t-1} to capture the relevant info from $(\mathbf{x}_{:t-1}, \mathbf{y}_{:t-1})$.

So why not just take k to be large — say, $k = 2^{30}$ to allow 30 bits of information? Unfortunately, evaluating G_T then becomes very expensive — both computationally and statistically. As we have seen, if we define $\mathbf{s}_t$ to be the belief state $[\![\boldsymbol{\alpha}_t]\!] \in \mathbb{R}^k$, updating it at each observation (x_t, y_t) (equation (6.3)) requires multiplication by a $k \times k$ matrix P. This takes time $O(k^2)$, and requires enough data to learn $O(k^2)$ transition probabilities.

6.3.2 Neural approximation of the model

As a solution, we might hope that for the inputs $\mathbf{x}$ observed in practice, the very high-dimensional belief states $[\![\boldsymbol{\alpha}_t]\!] \in \mathbb{R}^k$ might tend to lie near a d-dimensional manifold where $d \ll k$. Then we could take $\mathbf{s}_t$ to be a vector in $\mathbb{R}^d$ that compactly encodes the approximate coordinates of $[\![\boldsymbol{\alpha}_t]\!]$ relative to the manifold: $\mathbf{s}_t = v([\![\boldsymbol{\alpha}_t]\!])$, where v is the encoder.

In this new, nonlinearly warped coordinate system, the functions of $\mathbf{s}_{t-1}$ in (6.2)–(6.3) are no longer the simple, essentially linear functions given by (6.19) and (6.21). They become nonlinear functions operating on the manifold coordinates. (f_θ in (6.19) should now ensure that $\mathbf{s}_t^\top \approx v([\![(v^{-1}(\mathbf{s}_{t-1}))^\top P]\!])$, and g_θ in (6.21) should now estimate $\log\,(v^{-1}(\mathbf{s}_{t-1}))^\top P\mathbf{1}$.) In a sense, this is the reverse of the "kernel trick" (Boser, Guyon, and Vapnik, 1992) that converts a low-dimensional nonlinear function to a high-dimensional linear one.

[9] This is not true for the language of balanced parentheses.

Our hope is that $\mathbf{s}_t$ has enough dimensions $d \ll k$ to capture the useful information from the true $[\![\boldsymbol{\alpha}_t]\!]$, **and** that θ has enough dimensions $\ll k^2$ to capture most of the dynamics of equations (6.19) and (6.21). We thus proceed to fit the neural networks f_θ, g_θ directly to the data, *without ever knowing* the true k or the structure of the original operators $P \in \mathbb{R}^{k \times k}$.

We regard this as the implicit justification for various published probabilistic sequence models $p_\theta(\mathbf{y} \mid \mathbf{x})$ that incorporate neural networks. These models usually have the form of §6.1.1. Most simply, (f_θ, g_θ) can be instantiated as one time step in an RNN (Aharoni and Goldberg, 2017), but it is common to use enriched versions such as deep LSTMs. It is also common to have the state $\mathbf{s}_t$ contain not only a vector of manifold coordinates in $\mathbb{R}^d$ but also some unboundedly large representation of $(\mathbf{x}, \mathbf{y}_{:t})$ (cf. equation (6.4)), so the f_θ neural network can refer to this material with an attentional (Bahdanau, Cho, and Bengio, 2015) or stack mechanism (Dyer et al., 2015).

A few such papers have used *globally* normalized conditional models that can be viewed as approximating some OOHMM, e.g., the parsers of Dyer et al. (2016) and Andor et al. (2016). That is the case (§6.1.1) that particle smoothing aims to support. Most papers are *locally* normalized conditional models (Kann and Schütze, 2016; Aharoni and Goldberg, 2017); these simplify supervised training and can be viewed as approximating IOHMMs (footnote 7). For locally normalized models, $H_t = 0$ by construction, in which case particle filtering (which estimates $H_t = 0$) is just as good as particle smoothing. Particle filtering is still useful for these models, but lookahead's inability to help them is an expressive limitation (known as *label bias*) of locally normalized models. We hope the existence of

particle smoothing (which learns an estimate H_t) will make it easier to adopt, train, and decode globally normalized models, as discussed in §6.1.3.

6.3.3 Neural approximation of logprob-to-go

We can adopt the same neuralization trick to approximate the OOHMM's logprob-to-go $H_t = C_t + \hat{H}_t$. We take $\bar{\mathbf{s}}_t \in \mathbb{R}^d$ on the same theory that it is a low-dimensional reparameterization of $[\![\beta_t]\!]$, and define $(\bar{f}_\phi, h_\phi)$ in equations (6.5)–(6.6) to be neural networks. Finally, we must replace the definition of C_t in (6.26) with another neural network c_ϕ that works on the low-dimensional approximations:[10]

$$C_t \triangleq c_\phi(\mathbf{s}_t, \bar{\mathbf{s}}_t) \qquad \text{(except that } C_T \triangleq 0\text{)} \tag{6.28}$$

The resulting approximation to (6.27) (which does not actually require h_ϕ) will be denoted $q_{\theta,\phi}$:

$$q_{\theta,\phi}(y_t \mid \mathbf{x}, \mathbf{y}_{:t-1}) \overset{\text{def}}{\propto} \exp\left(g_\theta(\mathbf{s}_{t-1}, x_t, y_t) + C_t\right) \tag{6.29}$$

The neural networks in the present section are all parameterized by ϕ, and are intended to produce an estimate of the logprob-*to-go* H_t — a function of $\mathbf{x}_{t:}$, which sums over all possible $\mathbf{y}_{t:}$.

By contrast, the OOHMM-inspired neural networks suggested in §6.3.2 were used to

[10] $C_T = 0$ is correct according to (6.26). Forcing this ensures $H_T = 0$, so our approximation becomes exact as of $t = T$.

specify an actual model of the logprob-*so-far* G_t — a function of $\mathbf{x}_{:t}$ and $\mathbf{y}_{:t}$ — using separate parameters θ.

Arguably ϕ has a harder modeling job than θ because it must implicitly sum over possible futures $\mathbf{y}_{t:}$.[11] We now consider how to get corrected samples from $q_{\theta,\phi}$ even if ϕ gives poor estimates of H_t, and then how to train ϕ to improve those estimates.

6.4 Particle smoothing

In this work, we assume nothing about the given model G_T except that it is given in the form of equations (6.1)–(6.3) (including the parameter vector θ).

Suppose we run the exact sampling strategy (§6.2) but approximate p_θ in (6.8) with a *proposal distribution* $q_{\theta,\phi}$ of the form in (6.28)–(6.29). Suppressing the subscripts on p and q for brevity, this means we are effectively drawing $\mathbf{y}$ not from $p(\mathbf{y} \mid \mathbf{x})$ but from

$$q(\mathbf{y} \mid \mathbf{x}) = \prod_{t=1}^{T} q(y_t \mid \mathbf{x}, \mathbf{y}_{:t-1}) \tag{6.30}$$

If $C_t \approx H_t + \text{const}$ within each y_t draw, then $q \approx p$.

Normalized importance sampling corrects (mostly) for the approximation by drawing *many* sequences $\mathbf{y}^{(1)}, \ldots \mathbf{y}^{(M)}$ IID from (6.30) and assigning $\mathbf{y}^{(m)}$ a relative *weight* of $w^{(m)} \triangleq$

[11] Relevant theoretical discussion is in § 3.

$\frac{p(\mathbf{y}^{(m)}|\mathbf{x})}{q(\mathbf{y}^{(m)}|\mathbf{x})}$. This *ensemble of weighted particles* yields a distribution

$$\hat{p}(\mathbf{y}) \triangleq \frac{\sum_{m=1}^{M} w^{(m)} \mathbb{I}(\mathbf{y} = \mathbf{y}^{(m)})}{\sum_{m=1}^{M} w^{(m)}} \approx p(\mathbf{y} \mid \mathbf{x}) \tag{6.31}$$

that can be used as discussed in §6.1.3. To compute $w^{(m)}$ in practice, we replace the numerator $p(\mathbf{y}^{(m)} \mid \mathbf{x})$ by the unnormalized version $\exp G_T$, which gives the same $\hat{p}$. Recall that each G_T is a sum $\sum_{t=1}^{T} g_\theta(\cdots)$.

Sequential importance sampling is an equivalent implementation that makes t the *outer* loop and m the *inner* loop. It computes a *prefix ensemble*

$$Y_t \triangleq \{(\mathbf{y}_{:t}^{(1)}, w_t^{(1)}), \ldots, (\mathbf{y}_{:t}^{(M)}, w_t^{(M)})\}$$

for each $0 \leq t \leq T$ in sequence. Initially, $(\mathbf{y}_{:0}^{(m)}, w_0^{(m)}) = (\epsilon, \exp C_0)$ for all m. Then for $0 < t \leq T$, we extend these particles in parallel:

$$\mathbf{y}_{:t}^{(m)} = \mathbf{y}_{:t-1}^{(m)} y_t^{(m)} \qquad \text{(concatenation)} \tag{6.32}$$

$$w_t^{(m)} = w_{t-1}^{(m)} \frac{\exp\left(g_\theta(\mathbf{s}_{t-1}, x_t, y_t) + C_t - C_{t-1}\right)}{q(y_t \mid \mathbf{x}, \mathbf{y}_{:t-1})} \tag{6.33}$$

where each $y_t^{(m)}$ is drawn from (6.29). Each Y_t yields a distribution $\hat{p}_t$ over prefixes $\mathbf{y}_{:t}$, which estimates the distribution $p_t(\mathbf{y}_{:t}) \overset{\text{def}}{\propto} \exp(G_t + C_t)$. We return $\hat{p} \triangleq \hat{p}_T \approx p_T = p$. This gives the same $\hat{p}$ as in (6.31): the final $\mathbf{y}_T^{(m)}$ are the same, with the same final weights $w_T^{(m)} = \frac{\exp G_T}{q(\mathbf{y}^{(m)}|\mathbf{x})}$, where G_T was now summed up as $C_0 + \sum_{t=1}^{T} g_\theta(\cdots) + C_t - C_{t-1}$.

That is our basic *particle smoothing* strategy. If we use the naive approximation $C_t = 0$ everywhere, it reduces to *particle filtering*. In either case, various well-studied improvements become available, such as various resampling schemes (Douc and Cappé, 2005) and the particle cascade (Paige et al., 2014).[12]

An easy improvement is *multinomial resampling* (Douc and Cappé, 2005). After computing each $\hat{p}_t$, this replaces Y_t with a set of M new draws from $\hat{p}_t$ ($\approx p_t$), each of weight 1 — which tends to drop low-weight particles and duplicate high-weight ones.[13] For this to usefully focus the ensemble on good prefixes $\mathbf{y}_{:t}$, p_t should be a good approximation to the true marginal $p(\mathbf{y}_{:t} \mid \mathbf{x}) \propto \exp(G_t + H_t)$ from (6.11). That is why we arranged for $p_t(\mathbf{y}_{:t}) \propto \exp(G_t + C_t)$. Without C_t, we would have only $p_t(\mathbf{y}_{:t}) \propto \exp G_t$ — which is fine for the traditional particle filtering setting, but in our setting it ignores future information in $\mathbf{x}_{t:}$ (which we have assumed is available) and also favors sequences $\mathbf{y}$ that happen to accumulate most of their global score G_T early rather than late (which is possible when the globally normalized model (6.1)–(6.2) is *not* factored in the generative form (6.4)).

6.5 Training the sampler heuristic

We now consider training the parameters ϕ of our sampler. These parameters determine the updates $\bar{f}_\phi$ in (6.6) and the compatibility function c_ϕ in (6.28). As a result, they determine

[12] The particle cascade would benefit from an estimate of $\hat{H}_t$, as it (like A* search) compares particles of different lengths.

[13] While resampling mitigates the degeneracy problem, it could also reduce the diversity of particles. In our experiments in this chapter, we only do multinomial resampling when the effective sample size of $\hat{p}_t$ is lower than $\frac{M}{2}$. Doucet and Johansen (2009) give a more thorough discussion on when to resample.

the proposal distribution q used in equations (6.30) and (6.33), and thus determine the stochastic choice of $\hat{p}$ that is returned by the sampler on a given input $\mathbf{x}$.

In this work, we simply try to tune ϕ to yield good proposals. Specifically, we try to ensure that $q_\phi(\mathbf{y} \mid \mathbf{x})$ in equation (6.30) is close to $p(\mathbf{y} \mid \mathbf{x})$ from equation (6.1). While this may not be necessary for the sampler to perform well downstream,[14] it does guarantee it (assuming that the model p is correct). Specifically, we seek to minimize

$$(1 - \lambda)\mathrm{KL}(p||q_\phi) + \lambda\mathrm{KL}(q_\phi||p) \quad (\text{with } \lambda \in [0, 1]) \tag{6.34}$$

averaged over examples $\mathbf{x}$ drawn from a training set.[15] (The training set need not provide true $\mathbf{y}$'s.)

The *inclusive KL divergence* $\mathrm{KL}\left(p||q_\phi\right)$ is an expectation under p. We estimate it by replacing p with a sample $\hat{p}$, which in practice we can obtain with our sampler under the current ϕ. (The danger, then, is that $\hat{p}$ will be biased when ϕ is not yet well-trained; this can be mitigated by increasing the sample size M when drawing $\hat{p}$ for training purposes.)

Intuitively, this term tries to encourage q_ϕ in future to re-propose those $\mathbf{y}$ values that turned out to be "good" and survived into $\hat{p}$ with high weights.

[14] In principle, one could attempt to train ϕ "end-to-end" on some downstream objective by using reinforcement learning or the Gumbel-softmax trick (Jang, Gu, and Poole, 2017; Maddison, Mnih, and Teh, 2017). For example, we might try to ensure that $\hat{p}$ closely matches the model's distribution p (equation (6.31)) — the "natural" goal of sampling. This objective can tolerate inaccurate local proposal distributions in cases where the algorithm could recover from them through resampling. Looking even farther downstream, we might merely want $\hat{p}$ — which is typically used to compute expectations — to provide accurate guidance to some decision or training process (see §6.1.3). This might not require fully matching the model, and might even make it desirable to deviate from an inaccurate model.

[15] Training a single approximation q_ϕ for all $\mathbf{x}$ is known as *amortized inference*.

The *exclusive KL divergence* $\mathrm{KL}(q_\phi||p)$ is an expectation under q_ϕ. Since we can sample from q_ϕ exactly, we can get an unbiased estimate of $\nabla_\phi \mathrm{KL}(q_\phi||p)$ with the likelihood ratio trick (Glynn, 1990).[16] (The danger is that such "REINFORCE" methods tend to suffer from very high variance.)

This term is a popular objective for variational approximation. Here, it tries to discourage q_ϕ from re-proposing "bad" $\mathbf{y}$ values that were proposed during training of q, but turned out to have low $\exp G_T$ relative to their proposal probability.

Our experiments balance "recall" (inclusive) and "precision" (exclusive) by taking $\lambda = \frac{1}{2}$ (which §6.6 compares to $\lambda \in \{0, 1\}$). Alas, because of our approximation to the inclusive term, neither term's gradient will "find" and directly encourage good $\mathbf{y}$ values that have never been proposed. §6.5.1 gives further discussion and formulas.

6.5.1 Gradients for training the proposal distribution

For a given $\mathbf{x}$, both forms of KL divergence achieve their minimum of 0 when $(\forall \mathbf{y})\ q_\phi(\mathbf{y} \mid \mathbf{x}) = p(\mathbf{y} \mid \mathbf{x})$. However, the two metrics penalize q_ϕ differently for mismatches. We simplify the notation below by writing $q_\phi(\mathbf{y})$ and $p(\mathbf{y})$, suppressing the conditioning on $\mathbf{x}$.

Inclusive KL Divergence The inclusive KL divergence has that name because it is finite only when $\mathrm{support}(q_\phi) \supseteq \mathrm{support}(p)$, i.e., when q_ϕ is capable of proposing any string $\mathbf{y}$ that has positive probability under p. This is required for q_ϕ to be a valid proposal distribution

[16] The normalizing constant of p from (6.1) can be ignored because the gradient of a constant is 0.

for importance sampling.

$$\mathrm{KL}\left(p||q_\phi\right) = \mathop{\mathbb{E}}_{\mathbf{y}\sim p}[\log p(\mathbf{y}) - \log q_\phi(\mathbf{y})]$$

The first term $\mathbb{E}_{\mathbf{y}\sim p}[\log p(\mathbf{y})]$ is a constant with regard to ϕ. As a result, the gradient of the above is just the gradient of the second term:

$$\nabla_\phi \mathrm{KL}(p||q_\phi) = \nabla_\phi \underbrace{\mathop{\mathbb{E}}_{\mathbf{y}\sim p}[-\log q_\phi(\mathbf{y})]}_{\text{the cross-entropy } H(p,q_\phi)}$$

We cannot directly sample from p. However, our weighted mixture $\hat{p}$ from equation (6.31) (obtained by sequential importance sampling) could be a good approximation:

$$\begin{aligned}\nabla_\phi \mathrm{KL}(p||q_\phi) &\approx \nabla_\phi \mathop{\mathbb{E}}_{\mathbf{y}\sim\hat{p}}[-\log q_\phi(\mathbf{y})] \\ &= \sum_{t=1}^{T} \mathop{\mathbb{E}}_{\hat{p}}[-\nabla_\phi \log q_\phi(y_t \mid y_{:t-1}, \mathbf{x})]\end{aligned} \tag{6.35}$$

Following this approximate gradient downhill has an intuitive interpretation: if a particular y_t value ends up with high relative weight in the final ensemble $\hat{p}$, then we will try to adjust q_ϕ so that it would have had a high probability of proposing that y_t value at step t in the first place.

Exclusive KL Divergence The exclusive divergence has that name because it is finite only when support(q_ϕ) $\subseteq$ support(p). It is defined by

$$\begin{aligned}
\mathrm{KL}(q_\phi \| p) &= \mathop{\mathbb{E}}_{\mathbf{y} \sim q_\phi} [\log q_\phi(\mathbf{y}) - \log p(\mathbf{y})] \\
&= \mathop{\mathbb{E}}_{\mathbf{y} \sim q_\phi} [\log q_\phi(\mathbf{y}) - \log \tilde{p}(\mathbf{y})] + \log Z \\
&= \sum_{\mathbf{y}} q_\phi(\mathbf{y}) \underbrace{\left[\log q_\phi(\mathbf{y}) - \log \tilde{p}(\mathbf{y})\right]}_{\text{call this } d_\phi(\mathbf{y})} + \log Z
\end{aligned}$$

where $p(\mathbf{y}) = \frac{1}{Z}\tilde{p}(\mathbf{y})$ for $\tilde{p}(\mathbf{y}) = \exp G_T$ and $Z = \sum_{\mathbf{y}} \tilde{p}(\mathbf{y})$. With some rearrangement, we can write its gradient as an expectation that can be estimated by sampling from q_ϕ.[17] Observing that Z is constant with respect to ϕ, first write

$$\begin{aligned}
\nabla_\phi \mathrm{KL}(q_\phi \| p) &= \sum_{\mathbf{y}} \nabla_\phi \left(q_\phi(\mathbf{y})\, d_\phi(\mathbf{y}) \right) \\
&= \sum_{\mathbf{y}} \left(\nabla_\phi q_\phi(\mathbf{y}) \right)\, d_\phi(\mathbf{y}) + \sum_{\mathbf{y}} \underbrace{q_\phi(\mathbf{y}) \nabla_\phi \log q_\phi(\mathbf{y})}_{= \nabla_\phi q_\phi(\mathbf{y})} \\
&= \sum_{\mathbf{y}} \left(\nabla_\phi q_\phi(\mathbf{y}) \right)\, d_\phi(\mathbf{y})
\end{aligned} \tag{6.36}$$

where the last step uses the fact that $\sum_{\mathbf{y}} \nabla_\phi q_\phi(\mathbf{y}) = \nabla_\phi \sum_{\mathbf{y}} q_\phi(\mathbf{y}) = \nabla_\phi 1 = 0$. We can turn this into an expectation with a second use of Glynn (1990)'s observation that $\nabla_\phi q_\phi(\mathbf{y}) =$

[17] This is an extension of the REINFORCE trick (Williams, 1992), which estimates the gradient of $\mathbb{E}_{\mathbf{y} \sim q_\phi}[\text{reward}(\mathbf{y})]$ when the reward is independent of ϕ. In our case, the expectation is over a quantity that does depend on ϕ.

$q_\phi(\mathbf{y})\nabla_\phi \log q_\phi(\mathbf{y})$ (the "likelihood ratio trick"):

$$\begin{aligned}\nabla_\phi \mathrm{KL}(q_\phi||p) &= \sum_{\mathbf{y}} q_\phi(\mathbf{y}) d_\phi(\mathbf{y}) \nabla_\phi \log q_\phi(\mathbf{y}) \\ &= \mathop{\mathbb{E}}_{\mathbf{y}\sim q_\phi}[d_\phi(\mathbf{y})\nabla_\phi \log q_\phi(\mathbf{y})] \end{aligned} \tag{6.37}$$

which can, if desired, be further rewritten as

$$\begin{aligned} &= \mathop{\mathbb{E}}_{\mathbf{y}\sim q_\phi}[d_\phi(\mathbf{y})\nabla_\phi\, d_\phi(\mathbf{y})] \\ &= \mathop{\mathbb{E}}_{\mathbf{y}\sim q_\phi}[\nabla_\phi \left(\tfrac{1}{2} d_\phi(\mathbf{y})^2\right)] \end{aligned} \tag{6.38}$$

If we regard $d_\phi(\mathbf{y})$ as a signed error (in the log domain) in trying to fit q_ϕ to $\tilde{p}$, then the above gradient of KL can be interpreted as the gradient of the mean squared error (divided by 2).

We would get the same gradient for any rescaled version of the unnormalized distribution $\tilde{p}$, but the formula for obtaining that gradient would be different. In particular, if we rewrite the above derivation but add a constant b to both $\log \tilde{p}(\mathbf{y})$ and $\log Z$ throughout (equivalent to adding b to G_T), we will get the slightly generalized expectation formulas

$$\mathop{\mathbb{E}}_{\mathbf{y}\sim q_\phi}[(d_\phi(\mathbf{y}) - b)\nabla_\phi \log q_\phi(\mathbf{y})] \tag{6.39}$$

$$\mathop{\mathbb{E}}_{\mathbf{y}\sim q_\phi}[\nabla_\phi \left(\tfrac{1}{2}\left(d_\phi(\mathbf{y}) - b\right)^2\right)] \tag{6.40}$$

in place of equations (6.37) and (6.38) respectively. By choosing an appropriate "baseline" b,

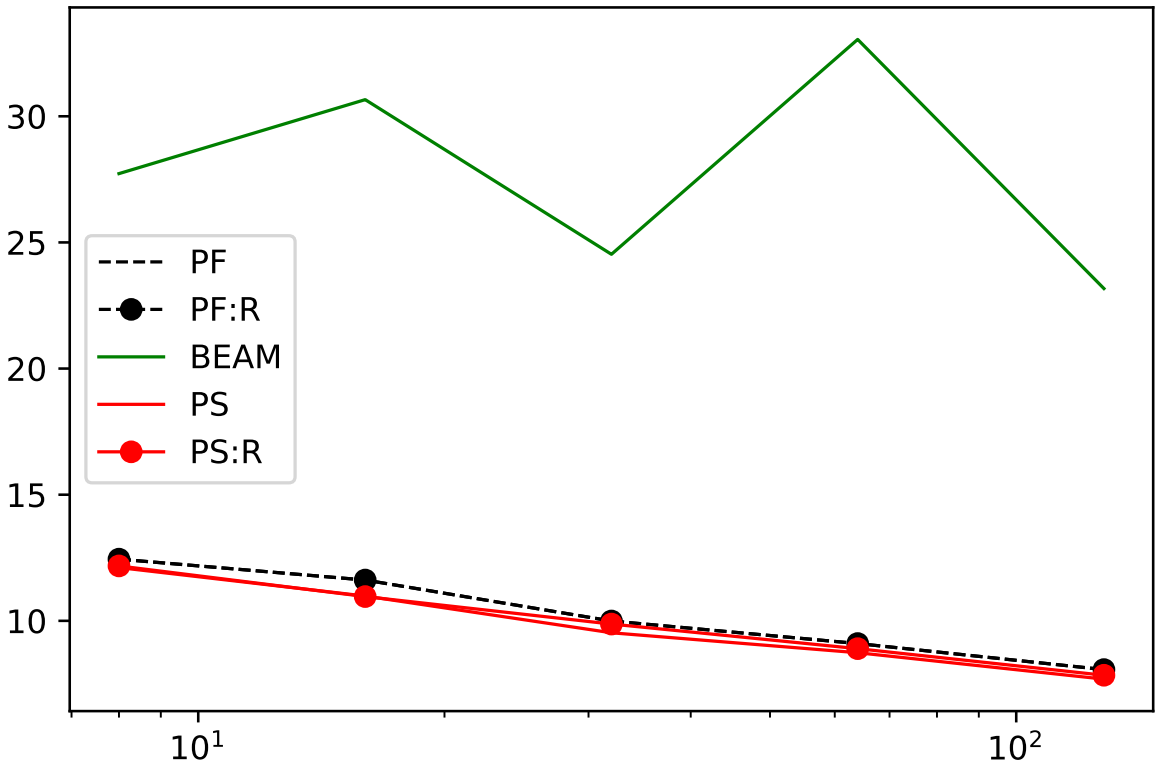


Figure 6.2: Offset KL divergence for the source separation task on phoneme sequences.

we can reduce the variance of the sampling-based estimate of these expectations. This is similar to the use of a baseline in the REINFORCE algorithm (Williams, 1992). In this work we choose b using an exponential moving average of past $\mathbb{E}[d_\phi(\mathbf{y})]$ values: at the end of each training minibatch, we update $b \leftarrow 0.1 \cdot b + 0.9 \cdot \bar{d}$, where $\bar{d}$ is the mean of the estimated $\mathbb{E}_{\mathbf{y} \sim q_\phi(\cdot|\mathbf{x})}[d_\phi(\mathbf{y})]$ values for all examples $\mathbf{x}$ in the minibatch.

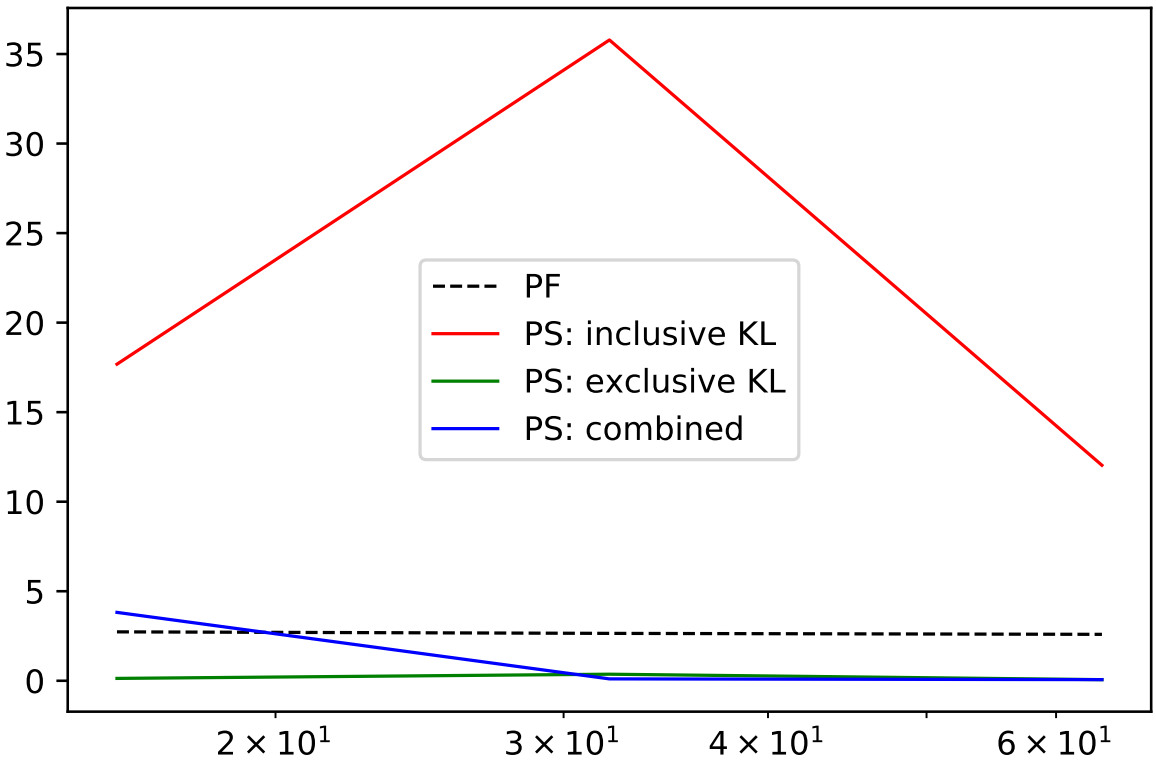


Figure 6.3: Offset KL divergence on the *last char* task: a pathological case where a naive particle filtering sampler does really horribly, and an ill-trained smoothing sampler even worse. The logarithmic x-axis is the particle size used to train the sampler. At test time we evaluate with a fixed particle size ($M = 32$).

6.6 Effect of different objective functions on lookahead optimization

§6.5 discussed inclusive and exclusive KL divergences, and gave our rationale for optimizing an interpolation of the two. Here we study the effect of the interpolation weight. We train the lookahead sampler, and the joint language model, on a toy problem called "last char," where $\mathbf{y}$ is a deterministic function of $\mathbf{x}$: either a lowercased version of $\mathbf{x}$, or an identical copy of $\mathbf{x}$, depending on whether the last character of $\mathbf{x}$ is `0` or `1`. Note that this problem requires lookahead.

We obtain our $\mathbf{x}$ sequences by taking the phoneme sequence data from the stressed syllable tagging task and flipping a fair coin to decide whether to append `0` or `1` to each sequence. Thus, the dataset may include ($\mathbf{x}$, $\mathbf{y}$) pairs such as (`K AU CH 0`, `k au ch 1`) or (`K AU CH 1`, `K AU CH 1`), but not (`K AU CH 1`, `k au ch 1`).

We treat this as a tagging problem, and treat it with our tagging model in §6.7.1. Results are in Figure 6.3. We see that optimizing for $\mathrm{KL}(\hat{p}||q)$ at a low particle size gives much worse performance than other methods, presumably due to the worse approximation quality (equation (6.35)). On the other hand, the objective function $\mathrm{KL}(q||p)$ achieves constantly good performance. The middle ground $\frac{\mathrm{KL}(\hat{p}||q)+\mathrm{KL}(q||p)}{2}$ improves when the particle size increases, and achieves slightly better results than $\mathrm{KL}(q||p)$ at larger particle sizes.

6.7 Models for the experiments

To evaluate our methods, we needed pre-trained models p_θ. We experimented on several models. In each case, we trained a *generative* model $p_\theta(\mathbf{x}, \mathbf{y})$, so that we could try sampling from its posterior distribution $p_\theta(\mathbf{y} \mid \mathbf{x})$. This is a very common setting where particle smoothing should be able to help. Details for replication are given in §6.7.4.

6.7.1 Tagging models

We can regard a tagged sentence $(\mathbf{x}, \mathbf{y})$ as a string over the "pair alphabet" $\mathcal{X} \times \mathcal{Y}$. We train an RNN language model over this "pair alphabet" — this is a neuralized OOHMM as suggested in §6.3.2:

$$\log p_\theta(\mathbf{x}, \mathbf{y}) = \sum_{t=1}^{T} \log p_\theta(x_t, y_t \mid \mathbf{s}_{t-1})$$

This model is locally normalized, so that $\log p_\theta(\mathbf{x}, \mathbf{y})$ (as well as its gradient) is straightforward to compute for a given training pair $(\mathbf{x}, \mathbf{y})$. Joint sampling from it would also be easy (§6.3.2).

However, $p(\mathbf{y} \mid \mathbf{x})$ is globally renormalized (by an unknown partition function that depends on $\mathbf{x}$, namely $\exp H_0$). Conditional sampling of $\mathbf{y}$ is therefore potentially hard. Choosing y_t optimally requires knowledge of H_t, which depends on the future $\mathbf{x}_{t:}$.

As we noted in §6.1, many NLP tasks can be seen as tagging problems. In this chapter

we experiment with two such tasks:

English stressed syllable tagging providing good reason to use the *lookahead* provided by particle smoothing: the stress of a syllable often depends on the number of remaining syllables.[18] This task tags a sequence of phonemes $\mathbf{x}$, which form a word, with their stress markings $\mathbf{y}$. Our training examples are the stressed words in the CMU pronunciation dictionary (Weide, 2005). We test the sampler on held-out unstressed words.

Chinese NER is a familiar textbook application and reminds the reader that our formal setup (tagging) provides enough machinery to treat other tasks (chunking). This task does named entity recognition in Chinese, by tagging the characters of a Chinese sentence in a way that marks the named entities. We use the dataset from Peng and Dredze (2015), whose tagging scheme is a variant of the BIO scheme mentioned in §6.1. We test the sampler on held-out sentences.

6.7.2 String source separation

This is an artificial task that provides a discrete analogue of speech source separation (Zibulevsky and Pearlmutter, 2001). The generative model is that J strings (possibly of different lengths) are generated IID from an RNN language model, and are then combined into a single string $\mathbf{x}$ according to a random *interleaving* string $\mathbf{y}$. We formally describe the generative process in §6.7.3. The posterior $p(\mathbf{y} \mid \mathbf{x})$ predicts the interleaving string,

[18] English, like many other languages, assigns stress from right to left (Hayes, 1995).

which suffices to reconstruct the original strings. The interleaving string is selected from the uniform distribution over all possible interleavings (given the J strings' lengths). For example, with $J = 2$, a possible generative story is that we first sample two strings Foo and Bar from an RNN language model. We then draw an interleaving string 112122 from the aforementioned uniform distribution, and interleave the J strings deterministically to get FoBoar.

$p(\mathbf{x}, \mathbf{y})$ is proportional to the product of the probabilities of the J strings. The only parameters of p_θ, then, are the parameters of the RNN language model, which we train on clean (non-interleaved) samples from a corpus. We test the sampler on random interleavings of held-out samples.

The state $\mathbf{s}$ (which is provided as an input to c_θ in (6.28)) is the concatenation of the J states of the language model as it independently generates the J strings, and $g_\theta(\mathbf{s}_{t-1}, x_t, y_t)$ is the log-probability of generating x_t as the next character of the $y_t{}^{\text{th}}$ string, given that string's language model state within $\mathbf{s}_{t-1}$. As a special case, $\mathbf{x}_T = \text{EOS}$ (see footnote 1), and $g_\theta(\mathbf{s}_{T-1}, \text{EOS}, \text{EOS})$ is the total log-probability of termination in all J language model states.

String source separation has good reason for lookahead: appending character "o" to a reconstructed string " gh" is only advisable if "s" and "t" are coming up soon to make "ghost." It also illustrates a powerful application setting — posterior inference under a generative model. This task conveniently allowed us to construct the generative model from a pre-trained language model. Our constructed generative model illustrates that the state $\mathbf{s}$ and transition function f can reflect interesting problem-specific structure.

CMU Pronunciation dictionary The CMU pronunciation dictionary (already used above) provides sequences of phonemes. Here we use words no longer than 5 phonemes. We interleave the (unstressed) phonemes of $J = 5$ words.

Penn Treebank The PTB corpus (Marcus, Marcinkiewicz, and Santorini, 1993) provides English sentences, from which we use only the sentences of length ≤ 8. We interleave the words of $J = 2$ sentences.

6.7.3 Generative process for source separation

Given an alphabet Σ, J strings $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(J)} \in \Sigma^*$ are independently sampled from the respective distributions $p^{(1)}, \ldots p^{(J)}$ over Σ^* (possibly all the same distribution $p^{(1)} = \cdots = p^{(J)}$). These source strings are then combined into a single observed string $\mathbf{x}$, of length $K = \sum_j K_j$, according to an **interleaving string** $\mathbf{y}$, also of length K. For example, $\mathbf{y} = 1132123$ means to take two characters from $\mathbf{x}^{(1)}$, then a character from $\mathbf{x}^{(3)}$, then a character from $\mathbf{x}^{(2)}$, etc. Formally speaking, $\mathbf{y}$ is an element of the mix language $\mathcal{Y}_{\mathbf{x}} = \textsc{mix}(1^{k_1}, 2^{k_2}, \ldots, j^{k_j})$, and we construct $\mathbf{x}$ by specifying the character $x_k \in \Sigma$ to be $x^{(y_k)}_{|\{i \leq k\,:\, y_i = y_k\}|}$. We assume that $\mathbf{y}$ is drawn from some distribution over $\mathcal{Y}_{\mathbf{x}}$. The source separation problem is to recover the interleaving string $\mathbf{y}$ from the interleaved string $\mathbf{x}$.

We assume that each source model $p^{(j)}(\mathbf{x}^{(j)})$ is an RNN language model — that is, a locally normalized state machine that successively generates each character of $\mathbf{x}^{(j)}$ given its left context. Thus, each source model is in some state $\mathbf{s}^{(j)}_t$ after generating the prefix $\mathbf{x}^{(j)}_{:t}$. In the

remainder of this paragraph, we suppress the superscript $^{(j)}$ for simplicity. The model now stochastically generates character x_{t+1} with probability $p(x_{t+1} \mid \mathbf{s}_t)$, and from $\mathbf{s}_t$ and this x_{t+1} it deterministically computes its new state $\mathbf{s}_{t+1}$. If x_{t+1} is a special "end-of-sequence" character EOS, we return $\mathbf{x} = \mathbf{x}_{:t}$.

Given only $\mathbf{x}$ of length T, we see that $\mathbf{y}$ could be any element of $\{1, 2, \ldots, J\}^T$. We can write the posterior probability of a given $\mathbf{y}$ (by Bayes' Theorem) as

$$p(\mathbf{y} \mid \mathbf{x}) \propto p(\mathbf{y}) \prod_{j=1}^{J} p^{(j)}\left(\mathbf{x}^{(j)}\right) \tag{6.41}$$

where (for this given $\mathbf{y}$) $\mathbf{x}^{(j)}$ denotes the subsequence of $\mathbf{x}$ at indices k such that $y_k = j$. In our experiments, we assume that y was drawn uniformly from $\mathcal{Y}_{\mathbf{x}}$, so $p(\mathbf{y})$ is constant and can be ignored. In general, the set of possible interleavings $\mathcal{Y}_{\mathbf{x}}$ is so large that computing the constant of proportionality (Z) for a given $\mathbf{x}$ becomes prohibitive.

6.7.4 Implementation details

We implement all RNNs in this chapter as GRU networks (Cho et al., 2014) with $d = 32$ hidden units (state space $\mathbb{R}^{32}$). Each of our models (§6.7) always specifies the logprob-so-far in equations (6.2) and (6.3) using a 1-layer left-to-right GRU,[19] while the corresponding proposal distribution (§6.3.3) always specifies the state $\bar{\mathbf{s}}_t$ in (6.6) using a 2-layer right-to-left

[19]For the tagging task described in §6.7.1, $g_\theta(\mathbf{s}_{t-1}, x_t, y_t) \triangleq \log p_\theta(x_t, y_t \mid \mathbf{s}_{t-1})$, where the GRU state $\mathbf{s}_{t-1}$ is used to define a softmax distribution over possible (x_t, y_t) pairs in the same manner as an RNN language model (Mikolov et al., 2010). Likewise, for the source separation task (§6.7.2), the source language models described in §6.7.3 are GRU-based RNN language models.

GRU, and specifies the compatibility function C_t in (6.26) using a 4-layer feedforward ReLU network.[20] For the Chinese social media NER task (§6.7.1), we use the Chinese character embeddings provided by Peng and Dredze (2015), while for the source separation tasks (§6.7.2), we use the 50-dimensional GloVe word embeddings (Pennington, Socher, and Manning, 2014). In other cases, we train embeddings along with the rest of the network. We optimize with the Adam optimizer using the default parameters (Kingma and Welling, 2014) and L_2 regularization coefficient of 10^{-5}.

6.7.5 Training procedures

In all our experiments, we train the incremental scoring models (the tagging and source separation models described in §§6.7.1 and 6.7.2, respectively) on the training dataset T. We do early stopping, using perplexity on a held-out development set D_1 to choose the number of epochs to train (maximum of 3).

Having obtained these model parameters θ, we train our proposal distributions $q_{\theta,\phi}$ on T, keeping θ fixed and only tuning ϕ. Again we use early stopping, using the KL divergence from §6.8.1 on a separate development set D_2 to choose the number of epochs to train (maximum of 20 for the two tagging tasks and source separation on the PTB dataset, and maximum of 50 for source separation on the phoneme sequence dataset). We then evaluate q_{θ^*,ϕ^*} on the test dataset E.

[20] As input to C_t, we actually provide not only $\mathbf{s}_t, \bar{\mathbf{s}}_t$ but also the states $f_\theta(\mathbf{s}_{t-1}, x_t, y)$ (including $\mathbf{s}_t$) that could have been reached for *each* possible value y of y_t. We have to compute these anyway while constructing the proposal distribution, and we find that it helps performance to include them.

6.8 Experiments

In our experiments, we are given a pre-trained scoring model p_θ, and we train the parameters ϕ of a particle smoothing algorithm.[21]

We now show that our proposed neural particle smoothing sampler does better than the particle filtering sampler. To define "better," we evaluate samplers on the *offset KL divergence* from the true posterior.

6.8.1 Evaluation metrics

Given $\mathbf{x}$, the "natural" goal of conditional sampling is for the sample distribution $\hat{p}(\mathbf{y})$ to approximate the true distribution $p_\theta(\mathbf{y} \mid \mathbf{x}) = \exp G_T / \exp H_0$ from (6.1). We will therefore report — averaged over all held-out test examples $\mathbf{x}$ — the KL divergence

$$\mathrm{KL}(\hat{p} \| p) = \mathop{\mathbb{E}}_{\mathbf{y} \sim \hat{p}}[\log \hat{p}(\mathbf{y})] - (\mathop{\mathbb{E}}_{\mathbf{y} \sim \hat{p}}[\log \tilde{p}(\mathbf{y} \mid \mathbf{x})] - \log Z(\mathbf{x})),$$

where $\tilde{p}(\mathbf{y} \mid \mathbf{x})$ denotes the *unnormalized* distribution given by $\exp G_T$ in (6.2), and $Z(\mathbf{x})$ denotes its normalizing constant, $\exp H_0 = \sum_{\mathbf{y}} \tilde{p}(\mathbf{y} \mid \mathbf{x})$.

As we are unable to compute $\log Z(\mathbf{x})$ in practice, we replace it with an estimate $z(\mathbf{x})$ to obtain an *offset KL divergence*. This change of constant does not change the measured difference between two samplers, $\mathrm{KL}(\hat{p}_1 \| p) - \mathrm{KL}(\hat{p}_2 \| p)$. Nonetheless, we try to use a

[21] For the details of the training procedures and the specific neural architectures in our models, see §§6.7.4 and 6.7.5.

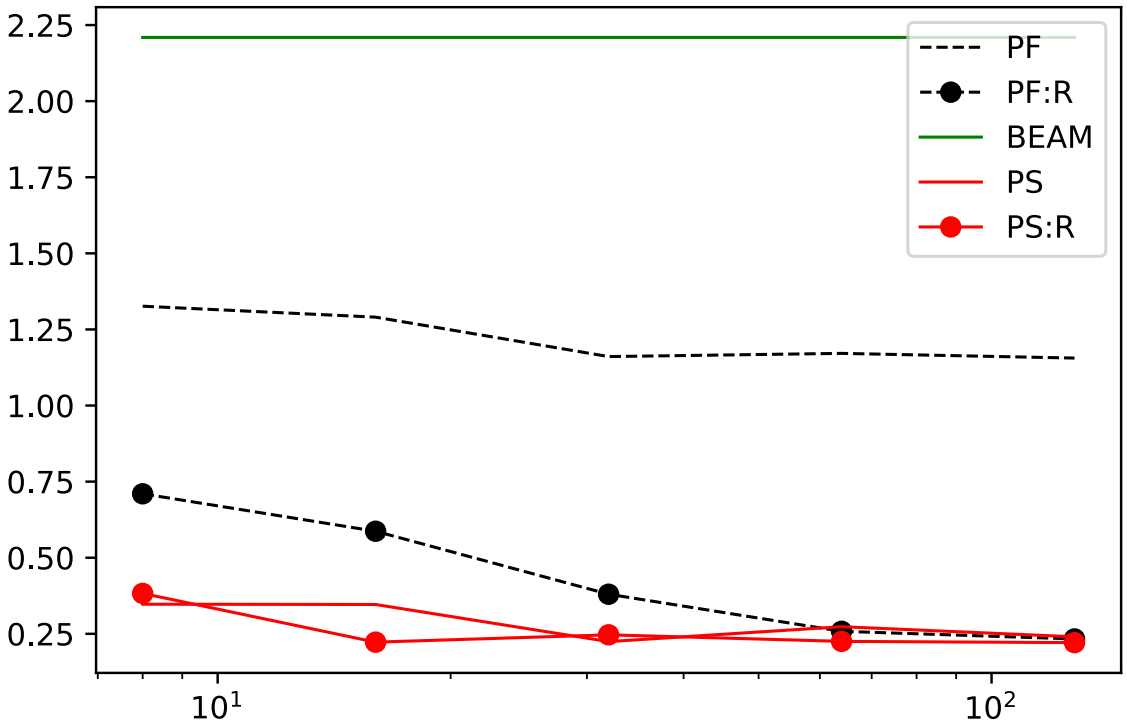


Figure 6.4: Tagging: stressed syllables. Abbreviations in the legend (for Figures 6.4–6.7): PF=particle filtering. PS=particle smoothing. BEAM=beam search. ':R' suffixes indicate resampled variants.

reasonable estimate so that the reported KL divergence is interpretable in an absolute sense. Specifically, we take $z(\mathbf{x}) = \log \sum_{\mathbf{y} \in \mathcal{Y}} \tilde{p}(\mathbf{y} \mid \mathbf{x}) \leq \log Z$, where $\mathcal{Y}$ is the full set of distinct particles $\mathbf{y}$ that we ever drew for input $\mathbf{x}$, including samples from the beam search models, while constructing the experimental results graph.[22] Thus, the offset KL divergence is a "best effort" lower bound on the true exclusive KL divergence $\mathrm{KL}(\hat{p} \| p)$.

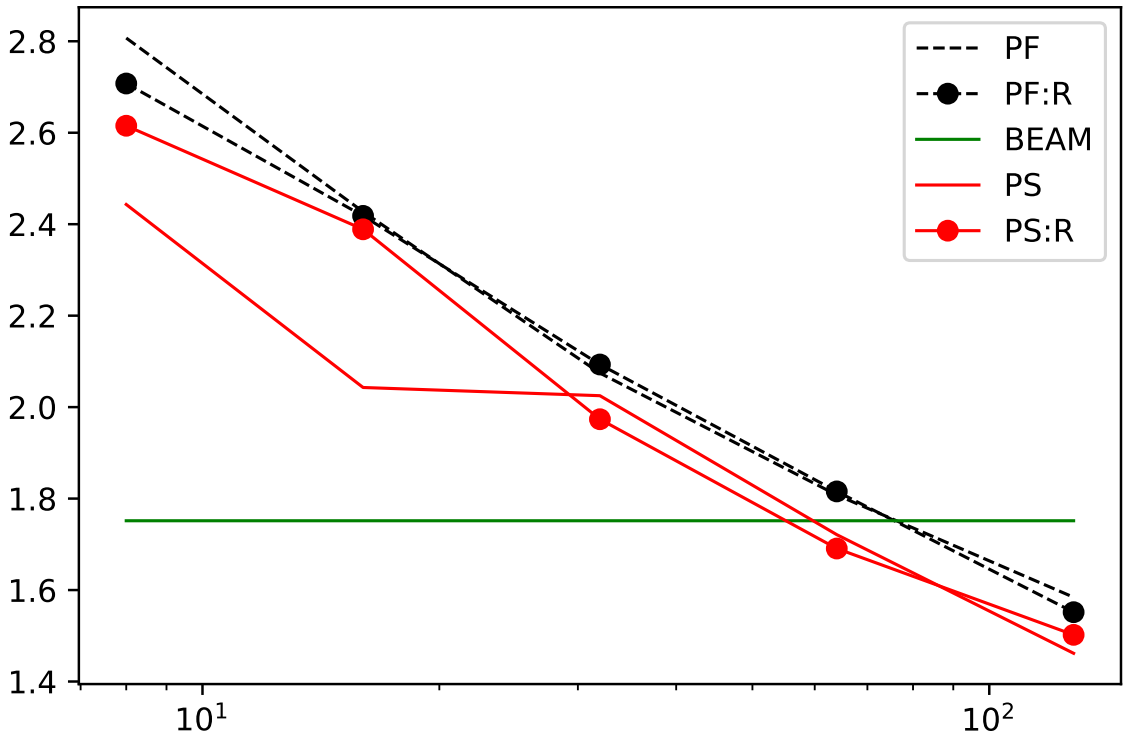


Figure 6.5: Tagging: Chinese NER

6.8.2 Results

In all experiments (§§6.7.1 and 6.7.2) we compute the offset KL divergence for both the particle filtering samplers and the particle smoothing samplers, for varying ensemble sizes M. We also compare against a beam search baseline that keeps the highest-scoring M particles at each step (scored by $\exp G_t$ with no lookahead). The results are in Figures 6.4–6.7. In these figures, the logarithmic x-axis is the size of particles M ($8 \leq M \leq 128$). The y-axis is the offset KL divergence described in §6.8.1 (in bits per sequence). The smoothing samplers

[22] Thus, $\mathcal{Y}$ was collected across all samplings, iterations, and ensemble sizes M, in an attempt to make the summation over $\mathcal{Y}$ as complete as possible. For good measure, we added some extra particles: whenever we drew M particles via particle smoothing, we drew an additional $2M$ particles by particle filtering and added them to $\mathcal{Y}$.

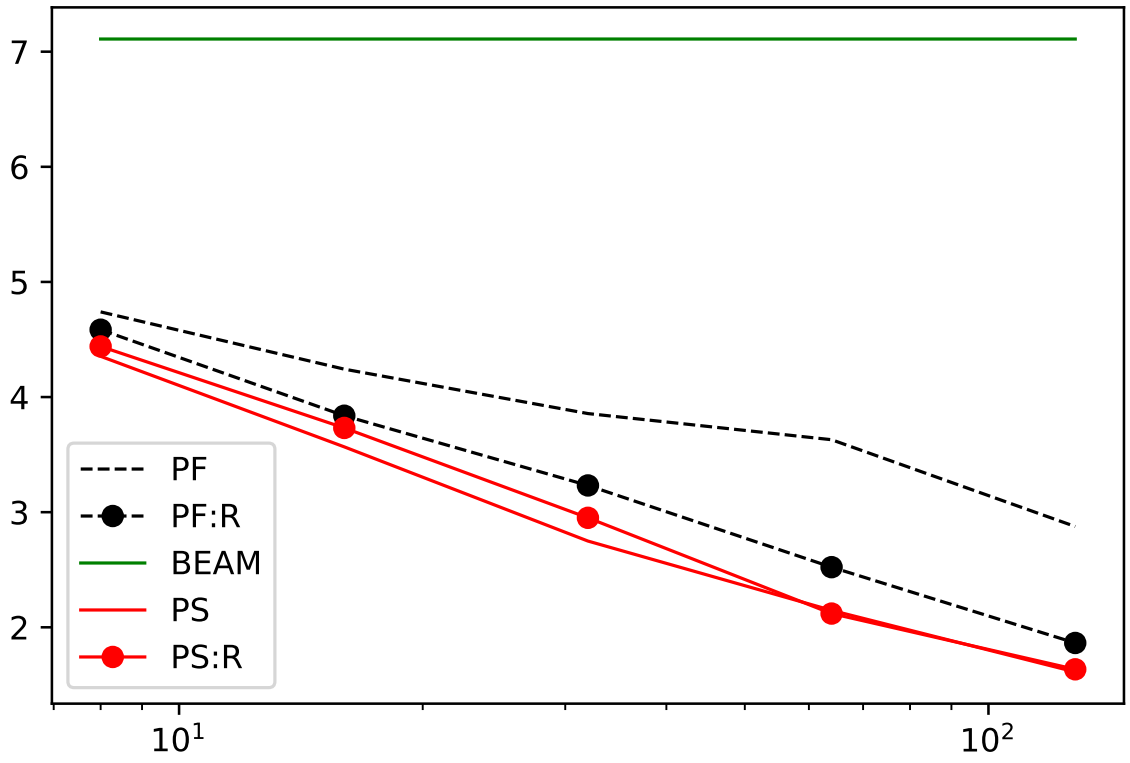


Figure 6.6: Source separation: PTB

offer considerable speedup: for example, in Figure 6.4, the non-resampled smoothing sampler achieves comparable offset KL divergences with only ¼ as many particles as its filtering counterparts. For readability, beam search results are omitted from Figure 6.7, but appear in Figure 6.2.

Given a fixed ensemble size, we see the smoothing sampler consistently performs better than the filtering counterpart. It often achieves comparable performance at a fraction of the ensemble size.

Beam search on the other hand falls behind on three tasks: stress prediction and the two source separation tasks. It does perform better than the stochastic methods on the Chinese NER task, but only at small beam sizes. Varying the beam size barely affects performance at

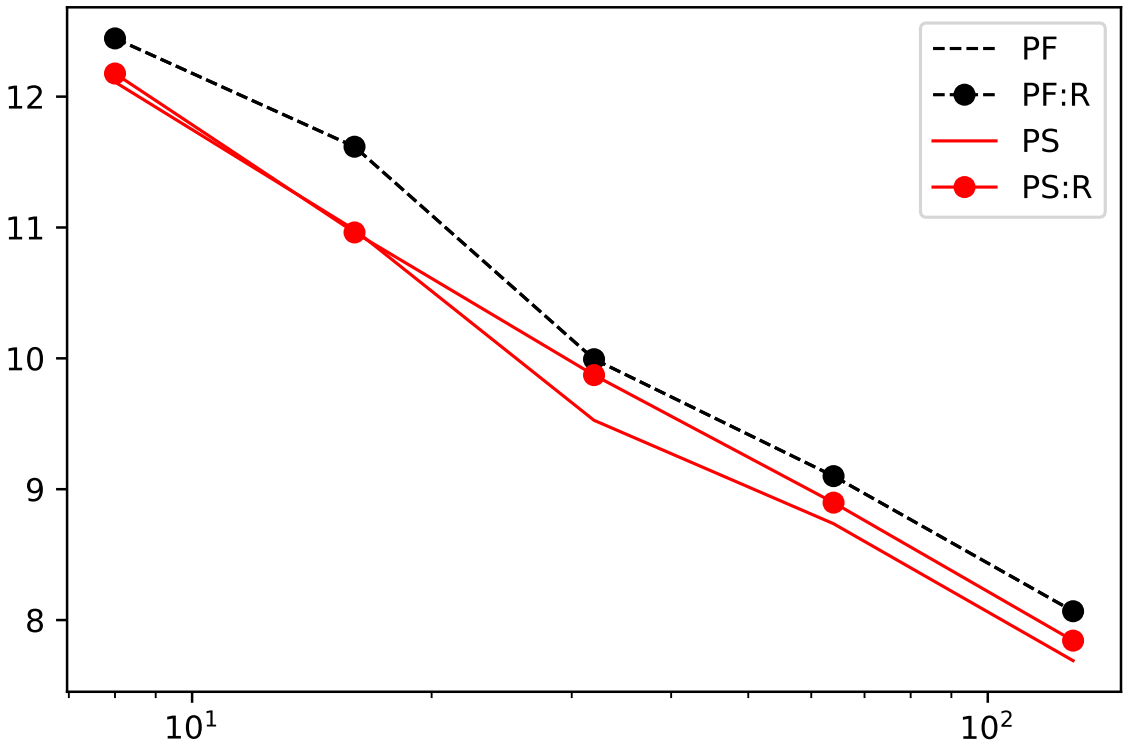


Figure 6.7: source separation: CMUdict

all, across all tasks. This suggests that beam search is unable to explore the hypothesis space well.

We experiment with resampling for both the particle filtering sampler and our smoothing sampler. In source separation and stressed syllable prediction, where the right context contains critical information about how viable a particle is, resampling helps particle filtering *almost* catch up to particle smoothing. Particle smoothing itself is not further improved by resampling, presumably because its effective sample size is high. The goal of resampling is to kill off low-weight particles (which were overproposed) and reallocate their resources to higher-weight ones. But with particle smoothing, there are fewer low-weight particles, so the benefit of resampling may be outweighted by its cost (namely, increased

variance).

6.9 Related work

Much previous work has employed sequential importance sampling for approximate inference of intractable distributions (Thrun, 2000; Andrews et al., 2017). Some of this work learns adaptive proposal distributions in this setting (Gu, Ghahramani, and Turner, 2015; Paige and Wood, 2016). The key difference in our work is that we consider future inputs, which is impossible in online decision settings such as robotics. Klaas et al. (2006) did do particle smoothing, like us, but they did not learn adaptive proposal distributions.

Just as we use a right-to-left RNN to guide *posterior sampling* of a left-to-right generative model, Krishnan, Shalit, and Sontag (2017) employed a right-to-left RNN to guide *posterior marginal inference* in the same sort of model. Serdyuk et al. (2018) used a right-to-left RNN to regularize training of such a model.

6.10 Conclusion

We have described neural particle smoothing, a sequential Monte Carlo method for approximate sampling from the posterior of incremental neural scoring models. Sequential importance sampling has arguably been underused in the natural language processing community. It is quite a plausible strategy for dealing with rich, globally normalized probability models such as neural models—particularly if a good sequential proposal

distribution can be found. Our contribution is a neural proposal distribution, which goes beyond particle filtering in that it uses a right-to-left recurrent neural network to "look ahead" to future symbols of $\mathbf{x}$ when proposing each symbol y_t. The form of our distribution is well-motivated.

There are many possible extensions to the work in this chapter. For example, we can learn the generative model and proposal distribution jointly; we can also infuse them with hand-crafted structure, or use more deeply stacked architectures; and we can try training the proposal distribution end-to-end (footnote 14). Another possible extension would be to allow each step of q to propose a *sequence* of actions, effectively making the tagset size ∞. This extension relaxes our $|\mathbf{y}| = |\mathbf{x}|$ restriction from §6.1 and would allow us to do general sequence-to-sequence transduction.

In § 7 we will also use neural particle smoothing for sampling from conditional distributions.

Chapter 7

Neuralization of Finite-State Transducers

7.1 Introduction

In this chapter we define neural finite-state transducers (NFSTs). NFSTs generalize both neural sequence models (*e.g.*, autoregressive sequence models in §2.1.3) and WFSTs (§2.1.2, Mohri, Pereira, and Riley (2008)). When seen as an extension of neural sequence models, NFSTs have an additional finite-state machine component that encodes monotonically aligned annotations.[1] On the other hand, compared to WFSTs, NFSTs allow much more powerful scoring functions of the annotations. In fact, we will describe an NFST as a pair of these two elements — namely a finite-state machine and a scoring function. Therefore, at a

[1] By *annotations* we mean both descriptions provided by (human) annotators, and discrete features which may be derived using a previously described algorithm.

high level, we may say the distinguishing features of NFSTs are the *monotonically aligned annotations* and *powerful scoring functions*.

7.1.1 NFSTs encode monotonically aligned annotations (as finite-state machines)

Recall that WFSTs can *featurize* string transduction as annotations (along the finite-state machine paths). These annotations are monotonically aligned with the string pairs that these models transduce. Likewise, NFSTs also featurize string transduction as annotations in their finite-state machine component. Representing these annotations as finite-state machines provides the following benefits:

Compact representation Many string transduction tasks can intuitively be described by monotonically aligned annotations. And finite-state machines represent these annotations compactly. Moreover, their topology can be engineered to incorporate domain knowledge (possibly by compiling a regular expression), so that its states reflect interpretable properties such as syllable boundaries or linguistic features. Indeed, we will show below how to make these properties explicit by "marking" the FST arcs. In addition to encoding annotations of *possible* transductions, FSTs can also be engineered to encode knowledge of *impossible* transductions, by making sure that there does not exist any annotation that accompanies the transductions we want to prohibit. Specifically, if the FST has no accepting path for some "illegal" $(\mathbf{x}, \mathbf{y})$ pairs,

then $p(\mathbf{x}, \mathbf{y}) = 0$ for any annotation weighting function.

Interpretability FSTs "explain" why it mapped $\mathbf{x}$ to $\mathbf{y}$ in terms of a latent path $\mathbf{a}$, which specifies a hard monotonic labeled alignment. If the annotation weighting function p defines a distribution over these annotations, then the posterior distribution $p(\mathbf{a} \mid \mathbf{x}, \mathbf{y})$ specifies which paths $\mathbf{a}$ are the best explanations.

Latent variables In addition to inductive bias and interpretability, finite-state machines can also encode discrete (and possibly latent) variables, by encoding these variables on the paths. We will show that, even when we restrict annotation weight functions to be autoregressive sequence models (which makes learning possible), NFSTs still retain greater expressiveness than such models, and even energy-based sequence models (under the non-uniform computation relaxation).

7.1.2 NFSTs make use of powerful scoring functions

Under WFSTs, the scoring function of annotations takes the form of a product of piecewise annotation (also known as 'feature') weights, and are often parametrized at the time of finite-state topology design. While such design implies that we can efficiently aggregate over annotation weights (or machine weights in §2.1.2), it also severely limits the expressive of WFSTs: piecewise annotations can only look at local contexts of string transduction. On the other hand, NFSTs can have arbitrary scoring functions: they can generally consider annotations that are separated by a long distance.

7.1.3 NFSTs are *not* merely WFSTs with powerful scoring functions

Our high-level overview in §§7.1.1 and 7.1.2 might seem to suggest that NFSTs are merely a straightforward extension of WFSTs, where the only change is that we now score a path using an arbitrary function (rather than using a lookup table). We note that this is *not* the case: if we were to approach NFSTs taking this simplistic view, many theoretical problems will arise. The root problem is that under WFSTs, *machine weight* (§2.1.2) is defined as the sum of all path weights in a machine. It is known that for a WFST this quantity may diverge (*i.e.*, may not be a real number).[2] But we cannot in general decide in finite time whether such a quantity diverges, if we naively allow the path weight to be an arbitrary function of symbols on the path. Furthermore, even when we limit ourselves to functions that results in finite machine weights (*i.e.*, is a real number), the machine weight can still be *uncomputable*, as implied by theoretical results in § 5. Lastly, we will show that these theoretical problems still affect us, even if we choose to parametrize path weights using common and popular neural sequence models.

Despite these theoretical restrictions, it might 'feel nice' if we could naively 'neuralize' weighted FSTs this way — each path in the machine would have its own contribution — its *path weight* — towards the machine weight, as in the (normalizable) WFST case. Can we

[2]Whether such a quantity diverges is decidable when all arc weights are non-negative (Salomaa and Soittola, 2012), and semi-decidable otherwise (Bailly and Denis, 2011).

identify a subclass of NFSTs that exhibit this familiar behavior?

These concerns lead us to a discussion on two different extensions of NFSTs' machine weights. Ultimately, the two extensions correspond to two possible answers to the question: is the notion of individual path weights useful? Depending on the application, both *yes* and *no* can be possible answers. Suppose we do *not* need to evaluate paths' weights, then different paths that correspond to the same string transduction annotation can be collapsed into a single one. Monotonically aligned annotations can thus be captured using *unweighted* finite-state machines, or equivalently regular languages. We will further show how NFST machine weights can be interpreted as event probabilities under this assumption.[3] But again we lose the path weight property this way: given a marked finite-state machine (which we will formally define in §7.2.1), its corresponding NFST machine may have a weight that differs from the sum of 'path weights', This might seem limiting to practitioners who seek to engineer NFSTs the way they did WFSTs.

On the other hand, if we *do* want individual path weights, monotonically aligned annotations can *sometimes* be captured using *weighted* finite-state machines, which may bear some more resemblance to WFSTs. However, there will be some aligned annotations that have no corresponding finite-weight WFSTs, if we require that the machine weight to be the sum of path weights.

These two different answers lead to two different definitions of **neuralization operators**, which we call **feature neuralization** and **literal neuralization** respectively.[4] We

[3]Note that paths under an NFST *cannot* necessarily be interpreted as events. Instead, *mark strings* are events. And their weights are not computed by the NFST automaton.

[4]The term is due to the fact that under literal neuralization, the machine weight—also known as a

will show that while either allows the NFSTs to capture all WFSTs (Definition 2.1.3), they make different trade-offs. Under the literal neuralization, the notion of individual path weights is preserved. But literal neuralization cannot be guaranteed to be well-behaving, unless we make stringent assumptions about both components of the NFST, namely the finite-state machine and scoring function. On the other hand, under feature neuralization, the NFST machine weight can be guaranteed to exist and well-behave, which in turn allows us to develop efficient parameter estimation algorithms. The downside of feature neuralization is that it forgoes the familiar behavior that every path in a finite-state machine contributes individually towards the machine weights. Finally, we will show that for some NFSTs, their feature and literal neuralizations coincide — so it is not necessarily a forced choice footnote 15.

7.1.4 Chapter outline

In this chapter, we first formally define NFSTs in §7.2. We then argue for the formal expressiveness of NFSTs in §7.3. We describe inference and parameter estimation methods for NFSTs in §7.4. Finally, we describe experiment results on several string transduction tasks in §7.5.

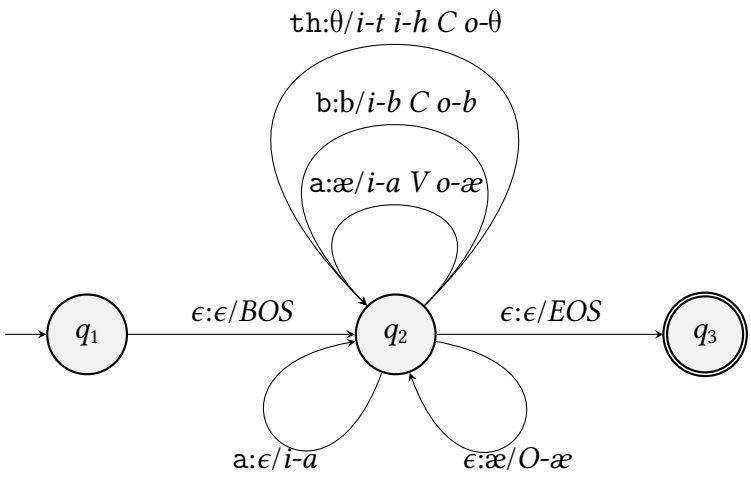
pathsum — is *literally* a sum over all paths.

7.2 Definition

An NFST is a pair (τ, G), where τ is a marked finite-state machine (which we formally define in §7.2.1, along with marked regular expressions in §7.2.2), and G a mark string scoring function (§7.2.4). We also define two different semirings that work with marked finite-state machines in §7.2.3, which are subsequently used to define feature and literal neuralization (§7.2.4.1).

7.2.1 Marked finite-state machines

MFST 7.2.1: A marked grapheme-phoneme transducer.



Marked finite-state machines are finite-state machines where one of the tapes is

designated as the **mark tape**, whose symbols are called **mark symbols**. For example, a marked finite-state acceptor (**MFSA**) has two tapes, one for input, and the other for mark symbols. Similarly, a marked finite-state transducer (**MFST**) has three tapes, two for input and output symbols, and another one for mark symbols. More formally:

Definition 7.2.1. *A marked finite-state transducer* $\tau = (\Sigma, \Delta, \Omega, Q, E, q_{init}, F)$ *is given by*

- *finite input,output, mark alphabets* Σ, Δ, Ω,
- *a finite set of states* Q,
- *a finite set of transitions* $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Delta \cup \{\epsilon\}) \times (\Omega \cup \{\epsilon\}) \times Q$,
- *an initial state* q_{init}, *and*
- *a set of final states* $F \subseteq Q$.

MFSAs are similarly defined as in Definition 7.2.1 without output symbols. When needed, an MFSA can be type-converted into an MFST, by introducing an output symbol that is identical to the input symbol per each arc. Loosely speaking, each accepting path **a** of a marked finite-state machine is *featurized* by the concatenation of all mark strings on arcs in the path.

We give an MFST example in MFST 7.2.1.[5] [6] Note that a mark symbol can contain

[5]Each arc in MFST 7.2.1 is annotated with `input : output/`*mark*. In our finite-state diagrams, an arc can have input/output/mark string(s) that contain multiple symbols. Such arcs are actually 'implemented' as a series of consecutive arcs through intermediate strings. We suppress such details in our diagrams for clarity, when appropriate.

[6]Mark strings in MFST 7.2.1 keep record of input and output strings they transduce. For example, the *i-t* signifies an arc whose input symbols contain '`t`'. In fact, under the topology of MFST 7.2.1, we can recover corresponding input and output strings from a mark string. We will see how such property allows for more efficient training in Lemma 7.4.1.

finer-grained information than input/output symbols: for example, marks such as *C* and *V* indicate whether an output IPA symbol is a consonant, or a vowel. Some information from an arc's mark symbol can also indicate state transition, which may not be reflected in the input/output symbols (*e.g.*, *BOS* and *EOS* marks).

Since a marked finite-state machine is a multitape finite-state machine, we can derive single-tape finite-state acceptors using the projection operation. We denote the **mark projection** of τ as τ_Ω. Similarly, the **input projection** τ_Σ and **output projection** τ_Δ.[7]

7.2.2 Marked regular expressions

We develop **marked regular expressions** (MRE) as both a convenient notation for marked finite-state machines,[8] and as a way to operate on marked finite-state machines. Specifically, we use the notation `a`:`b`/*c* to describe an marked finite-state transducer that accepts a single symbol `a` on the input tape, `b` on the output tape, and *c* on the mark tape. MREs are closed under standard rational operations — namely concatenation '.', union '|', and closure '$*$'.

Definition 7.2.2. *Let Σ, Δ, and Ω be three finite sets. A marked regular expression (MRE) with input alphabet Σ, output alphabet Δ, and mark alphabet Ω is either*

- *an empty set $\{\}$, or*
- *a transition of the form a:b/c, where $a \in \Sigma \cup \{\epsilon\}$, $b \in \Delta \cup \{\epsilon\}$, $c \in \Omega \cup \{\epsilon\}$, or*

[7] Besides single-tape projections such as τ_Σ, one can also do two-tape projections: for example, $\tau_{\Sigma,\Delta}$ would denote an unweighted FST, whose input and output projections are τ_Σ and τ_Δ respectively. Such projections are *lossy*; see `fstencode` (Allauzen et al., 2007) for 'lossless' projections to transducers or acceptors.

[8] In this work, we specifically design our MRE notation to work with MFSTs.

- *R.S where R and S are both MREs, or*
- *(R|S) where R and S are both MREs, or*
- *R^* where R is an MRE.*

There is also a special **composition** operator '$\circ$' in our (extended) set of MRE operators. As with ordinary FSTs, the MFST operator $\circ$ takes two MFSTs as input, which have input/alphabets (Σ, Γ) and (Γ, Δ) respectively, and output an MFST with input/output alphabets (Σ, Δ). But unlike their (W)FST counterparts, the MFST compositional operator is not commutative. We will discuss it in further detail in §7.2.2.1.

In this work, we leverage the brevity of rational operations under MREs to succinctly illustrate the compositional structure of larger MFSTs we build. As an example, MRE 7.2.8 describes an MRE that is equivalent to MFST 7.2.1, using components from MREs 7.2.5–7.2.7.[9]

MRE 7.2.5: BosMachine

$\epsilon : \epsilon/BOS$

MRE 7.2.6: EosMachine

$\epsilon : \epsilon/EOS$

[9]The isomorphism between MREs and MFSTs allows us to mingle MREs and MFSTs as operands under rational operations. Therefore MFSTs and MREs are used interchangeably in this work — for example, we may write $M_1.M_2$ where we depict M_1 as an MFST but M_2 as an MRE, assuming that there's an implicit coercion that converts M_1 into an MRE before the concatenation operation.

MRE 7.2.7: SingleTransition

(`th` : θ/*i-t i-h C o-*θ | ... `th` : θ/*i-t i-h V o-*θ)

MRE 7.2.8: An MRE equivalent to MFST 7.2.1

BosMachine.SingleTransition*.EosMachine

7.2.2.1 Composition of MREs

Many NLP applications that make use of finite-state techniques (*e.g.*, morphological and phonological analysis) are based on the **composition** of transducers, where strings are transduced into and from intermediate forms (either observed or hypothesized), to produce a monolithic transducer that models a (possibly weighted) relation between desired domains.

However, the composition operation is ultimately a *relational algebraic* one — unlike the *rational algebraic* operations: union, concatenation and closure. Multi-tape automata (a family where MFSTs belong, due to their 3 tapes) are generally not closed under composition (Kempe, Champarnaud, and Eisner, 2004). We would like MFSTs to be closed under composition (as common real-valued WFSTs are). Therefore we do not define MFST compositions as ordinary multi-tape automaton compositions (where the number of tapes increases after compositions (Hulden, 2015)). Instead, we define the composition operation using the $\otimes$ operation under both string-set and string-bag semirings (which we will formally introduce in §7.2.3) — a composed MFST is effectively a weighted finite-state transducer,

where each arc weight $w[a]$ is itself either an unweighted finite-state acceptor in alphabet Ω (or equivalently, a regular language $\subseteq \Omega^*$) or a bag of strings in alphabet Ω.

However, neither string-set nor string-bag semirings define a commutative $\otimes$ operation (*i.e.*, **a**.**b** — defined as the concatenation of strings **a** followed by **b** — is generally different from **b** followed by **a**: **b**.**a**). As a result, the composition of MFSTs will be sensitive to how mark symbols are distributed along consecutive arcs. The particular interleaving obtained also depends on the details of the weighted composition algorithm chosen.[10] Therefore, when we compose MFSTs τ_1 against τ_2 to obtain another MFST $\tau = \tau_1 \circ \tau_2$, mark strings in τ_Ω will be interleavings from mark strings in $\tau_{1\Omega}$ and $\tau_{2\Omega}$. But suppose we have another τ_3 where $\tau_{3\Omega} = \tau_{1\Omega}$ (*e.g.*, suppose we 'push' (Mohri, Pereira, and Riley, 2008) mark symbols of τ_1 to obtain τ_3), $(\tau_3 \circ \tau_2)_\Omega$ is still generally different from $(\tau_1 \circ \tau_2)_\Omega$.

To summarize, MRE compositions[11] are *not* algebraic operations of multitape relations: a composed MFST's mark projection is sensitive to the topology of the MFSTs, from which it is derived, and is *not* a function of just their mark projections. Nonetheless, regular expressions that include $\circ$ are still a valid way of building up interesting MFSTs that featurize input-output string pairs, which we subsequently 'neuralize' (a concept that we will introduce in §7.2.4) to produce flexible distributions over string pairs. We simply trust that the resulting mark strings in the MFST we built, whatever they are, will prove useful to modeling the string transductions that they are aligned to. A flexible mark string scoring

[10] We arbitrarily select from Allauzen, Riley, and Schalkwyk (2010) the version of composition that uses an epsilon-sequencing filter.

[11] Due to the equivalence between MREs and MFSTs, throughout the text we will be using the two terms interchangeably.

function (§7.2.4) should be able to adapt to the interleaving pattern.

7.2.3 Marked finite-state machines as *weighted* automata: two different semirings

In §7.2.1 we describe marked finite-state machines as *unweighted* multitape finite-state machines. However algorithm and software development have largely been focused on *weighted* one- and two-tape finite-state machines (§2.1.2). By transforming marked finite-state acceptors/transducers into weighted one- or two-tape machines, we can leverage existing highly-optimized software for various operations on these automata. To this end, we make use two different semirings — string-set (§7.2.3.1) and string-bag (§7.2.3.2) semirings — which allows us to see marked finite-state machines as *weighted* finite-state machines, under the two semirings respectively.

In this thesis, we denote machine weights of T under the string-set semiring as $[T]_{\text{set}}$, and string-bag as $[T]_{\text{bag}}$.

7.2.3.1 String-set semiring

Definition 7.2.3. *Let ϵ denote the empty string. The **string-set semiring** over S, where $\{\epsilon\} \in S$, $\{\} \in S$, has*

- $\overline{0} \triangleq \{\} \in S$,
- $\overline{1} \triangleq \{\epsilon\} \in S$,

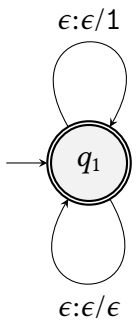
- $\oplus : S \times S \longrightarrow S, \mathbf{a} \oplus \mathbf{b} \triangleq \mathbf{a} \cup \mathbf{b}$,
- $\otimes : S \times S \longrightarrow S, \mathbf{a} \otimes \mathbf{b} \triangleq \{a.b \mid a \in \mathbf{a}, b \in \mathbf{b}\}$.

Equivalence between MFSTs and string-set-weighted FSTs. The string-set semiring is defined over sets of strings. In other words, the machine weight of a string-set-weighted finite-state machine T: $[T]_{\text{set}}$ is an unweighted language. Specifically, $[T]_{\text{set}}$ is the regular language recognized by the finite-state machine formed by regular operations on the arc weights (which are themselves regular languages), following the topology of T. Therefore, one can transform between MFSTs and equivalent string-set-weighted FSTs. Given string-set-weighted FST T where each arc a has a regular language (whose alphabet is Ω) weight $[a] \in S$, we can build an MFST τ where there is an accepting path π_1 in T with weight L (a language) if and only if there exists an accepting path π_2 in τ, where the input and output projections of π_1 and π_2 coincide, and that the mark projection of $\pi_2 \in L$. On the other hand, given MFST τ, it is also straightforward to build a string-set weighted FST T which has input projection τ_Σ, output projection τ_Δ, and a weight equal to the regular language τ_Ω, as follows: we build T as a WFST (§2.1.2) that has states identical to that of τ. And for every transition of τ: $(q, x, y, \omega, q') \in E_\tau$, we add a transition $(q, x, y, \{\omega\}, q') \in E_T$ in the weighted machine, where E_τ and E_T are sets of transitions of τ and T, respectively.

In short, we can easily convert between τ_Ω and $[T]_{\text{set}}$ easily. And with a slight abuse of notation, we write $[\tau]_{\text{set}}$ to denote the coercion of MFST τ into a string-set-weighted FST, followed by the evaluation of its machine weight. Such conversion is useful in that it

allows us to reuse finite-state operations (*e.g.*, Allauzen et al. (2007)) designed for weighted finite-state machines on MFSTs under the string-set semiring, which can be composed with other machines subsequently (§7.2.2.1). The string-set semiring view also offers a way to mechanically derive the set of mark strings on paths in τ that recognize an input string $\mathbf{x}$, since the weight sum of paths in a finite-state machine τ that recognize string $\mathbf{x}$ is the pathsum of $\mathbf{x} \circ \tau$; and likewise the weight sum of paths that recognize $(\mathbf{x}, \mathbf{y})$ transduction is the pathsum of $\mathbf{x} \circ \tau \circ \mathbf{y}$.

MFST 7.2.2: An MFST with ϵ-cycle.



7.2.3.2 String-bag semiring

A notable feature of the string-set semiring is that if two different paths in a machine have the same mark string $\boldsymbol{\omega}$, then they will both be represented by $\boldsymbol{\omega}$. As we noted in §7.2.3.1, different paths in a machine can always be disambiguated by having a unique identifier for each arc. Nonetheless, in this section we introduce an alternative — the string-bag semiring — which accounts for every path, regardless of whether they have unique mark

strings, by adopting a *bag semantics*.

Definition 7.2.4. *Let ϵ denote the empty string. A* ***string-bag semiring*** *over S has*

- $\overline{0} \triangleq f_0 \in S$ *where* $\forall \boldsymbol{\omega} \in \mathcal{P}(\Omega^*), f_0(\boldsymbol{\omega}) = 0$,
- $\overline{1} \triangleq f_1 \in S$ *where* $f_1(\epsilon) = 1, \forall \boldsymbol{\omega} \neq \epsilon \in \mathcal{P}(\Omega^*), f_1(\boldsymbol{\omega}) = 0$,
- $\oplus : S \times S \longrightarrow S, f_a \oplus f_b \triangleq f$ *where* $\forall \boldsymbol{\omega} \in \mathcal{P}(\Omega^*), f(\boldsymbol{\omega}) = f_a(\boldsymbol{\omega}) + f_b(\boldsymbol{\omega})$,
- $\otimes : S \times S \longrightarrow S, f_a \otimes f_b \triangleq f$ *where* $\forall \boldsymbol{\omega}_a \in \mathcal{P}(\Omega^*), \boldsymbol{\omega}_b \in \mathcal{P}(\Omega^*), f(\boldsymbol{\omega}_a.\boldsymbol{\omega}_b) = f_a(\boldsymbol{\omega}_a) \cdot f_b(\boldsymbol{\omega}_b)$.

String-bag-weighted FSTs can capture paths of some MFSTs. While the machine weights of string-set-weighted FSTs are unweighted languages, the machine weights of string-bag-weighted FSTs are generally *weighted* languages. Intuitively, the weight of string $\boldsymbol{\omega}$: $f(\boldsymbol{\omega})$ under the weighted language $[\![T]\!]_{\text{bag}}$ — which is the machine weight of the string-bag-weighted FST T — captures the number of accepting paths in T that have a single string $\boldsymbol{\omega}$ in their (singleton) supports.

However, string-bag-weighted FSTs that have ϵ-cycles (*e.g.*, MFST 7.2.2) can have *infinitely many* accepting paths. And as a result, they do not have well-defined weighted languages as their machine weights (*i.e.*, some of the string weights would diverge). Consequently, they do not have equivalent string-bag-weighted FSTs that have well-defined machine weights.

Nonetheless, for MFSTs that do not have ϵ-cycles, we can still convert between τ_Ω and $[\![T]\!]_{\text{bag}}$. As in the case of string-set semirings, with a slight abuse of notation, we write

$[\tau]_{\text{bag}}$ to denote the coercion of MFST τ into a string-bag-weighted FST, followed by the evaluation of its machine weight.

7.2.4 NFSTs as real-weighted relations

In §7.2.3 we described how MFSTs can be regarded as *weighted* relations between string pairs. However, the weights themselves are (both weighted and unweighted) languages under our characterization. In many machine learning applications, though, real-weighted relations are needed.

In this section we describe **neuralization** operators, which, with the help of a **mark string scoring function** G, *neuralizes* an MFST τ. We call the pair (τ, G) a neural finite-state transducer (or NFST for short). And finally, the aggregation of all strings' weights under G is defined as the NFST's machine weight.

So, specifically, how does neuralization work? Below we will introduce two types of neuralization operators — $N_{\text{feature}}[]$ and $N_{\text{literal}}[]$ respectively — that correspond to string-set and string-bag semirings.

7.2.4.1 Feature and literal neuralization

A mark string scoring function $G : \Omega^* \longrightarrow \mathbb{R}_{\geq 0}$ is a weighted language that maps mark strings to non-negative scalars. We define **neuralization** operators $N_{\text{literal}}[]$ and $N_{\text{feature}}[]$ to be real functions that take two inputs: a marked finite-state machine τ, and a mark string scoring function G:

Definition 7.2.5. *Let τ be a marked finite-state machine. And let G be a mark string scoring function. The feature pathsum of τ under G is*

$$N_{\text{feature}}[\tau, G] \triangleq \sum_{\boldsymbol{\omega} \in [\tau]_{\text{set}}} G(\boldsymbol{\omega}) \in \mathbb{R}. \tag{7.1}$$

And

Definition 7.2.6. *Let τ be an acyclic marked finite-state machine. And let G be a mark string scoring function. The literal pathsum of τ under G is*

$$N_{\text{literal}}[\tau, G] \triangleq \sum_{\boldsymbol{\omega} \in [\tau]_{\text{bag}}} G(\boldsymbol{\omega}) f(\boldsymbol{\omega}) \in \mathbb{R}, \tag{7.2}$$

where $f : \Omega^* \longrightarrow \mathbb{R}_{\geq 0}$ is the weight of $\boldsymbol{\omega}$ in the weighted language $[\tau]_{\text{bag}}$.

$N_{\text{feature}}[\tau, G]$ is a real-valued function of both τ and G. Recall that τ is a weighted relation over $\Sigma^* \times \Delta^*$ (§7.2.3). Here we extend the feature neuralization operator to build a real-weighted relation from a language-weighted relation $[\tau]_{\text{bag}}$, by mapping the weight associated with each string pair recognized by $[\tau]_{\text{bag}}$ to a real scalar. We denote the real-weighted relation over $\Sigma^* \times \Delta^*$ under G as follows:[12]

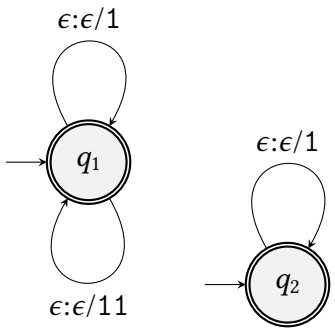
$$N_{\text{feature}}[\tau, G](\mathbf{x}, \mathbf{y}) \triangleq N_{\text{feature}}[\mathbf{x} \circ \tau \circ \mathbf{y}, G]. \tag{7.3}$$

[12] This is a notation abuse since $N_{\text{feature}}[\tau, G]$ is by definition a scalar (Definition 7.2.5).

$N_{\text{literal}}[\tau, G](\mathbf{x}, \mathbf{y})$ is also defined in a similar fashion.

7.2.4.2 Which neuralization operator should I use?

MFST 7.2.3: Two MFSTs that have the same mark projection.



Should one define an NFST using $N_{\text{literal}}[]$ or $N_{\text{feature}}[]$? One may wonder if the $N_{\text{feature}}[]$ is capable of modeling so-called 'globally normalized' WFSTs, whose arc weights $w[a] = \prod_{f \in \mathbf{f}_a} \exp \lambda_f$ are parametrized as products of exponentiated feature weights λ_f (Dreyer and Eisner, 2009). These parametric families of WFSTs define two paths $\mathbf{a}_1$ and $\mathbf{a}_2$ that have the same bag of features to have the same path weight, both of which contribute towards the machine weight $[\![T]\!]$. At a first glance, $N_{\text{feature}}[]$ may seem unable to capture such WFSTs, since they use the string-set semiring, which consider only mark string *sets*: two paths that have the same mark string only contribute once towards $[\![T]\!]$. It might even appear that the double-counting of paths grants the aforementioned WFSTs an expressiveness edge over models that only consider the string set τ_Ω. We emphasize that *every* WFST T with a finite machine weight $[\![T]\!]$ *can* be expressed as $N_{\text{feature}}[\tau, G]$, where τ is an MFST and G is a

unigram model.

In this thesis, we make use of MFSTs that have a unique mark string per each path in all our experiments: for those MFSTs τ, $N_{\text{literal}}[\tau, G] = N_{\text{feature}}[\tau, G]$. The uniqueness of each path's mark string can be guaranteed if each arc in the MFST contains a unique arc identifier.[13]. And to reduce clutter, we will denote $N_{\text{feature}}[\tau, G]$ as $N[\tau, G]$ in future sections. But in the case where they can differ, one needs to consider the following trade-off between their different properties:

Finite machine weights. In §7.2.3.2 we said *some* MFSTs' paths cannot be captured under the string-bag semiring. Consequently, for those MFSTs τ, $N_{\text{literal}}[\tau, G]$ — which makes use of the string-bag semiring — cannot be (finite) real numbers, regardless of the choice of G. This can pose as a problem in cases where they are regarded as 'goodness' quantities. On the other hand, $N_{\text{feature}}[]$ does not have this problem.

Path weights. As we previously said, a big difference between $N_{\text{feature}}[]$ and $N_{\text{literal}}[]$ is that $N_{\text{feature}}[]$ lacks the notion of path weights. Specifically, $N_{\text{literal}}[]$ uses the string-bag semiring, which keeps track of distinct paths that have the same mark string. But $N_{\text{feature}}[]$, which uses the string-set semiring, conflate them into one mark string. For example, consider the two illustrated MFSTs (τ_1, τ_2) in MFST 7.2.3. τ_1 and τ_2 have the same mark projection. Therefore, $N_{\text{feature}}[\tau_1, G] = N_{\text{feature}}[\tau_2, G]$ for any G, even

[13]Indeed, any MFST can be 'uniquified' into one such MFST, by appending a unique mark symbol to every arc. However doing so changes the mark projection of the MFST. Moreover, when coupled with autoregressive mark string scoring functions, the lack of flexibility of these arc identifiers' positions (they can only appear in the arcs where they belong) may cause expressiveness problems, per our discussion in § 3

though they differ in topology. On the other hand, under $N_{\text{literal}}[]$, each path in τ that shares the same mark string $\boldsymbol{\omega}$ will share an equal weight $G(\boldsymbol{\omega})$ to the machine weight $N_{\text{literal}}[\tau, G]$ following equation (7.2). Also, in general, $N_{\text{feature}}[h(\tau), G] \neq N[\tau, G \circ h]$ where h is a homomorphism of marks. In the notation here, h is extended over MFSTs (on the left-hand side it coarsens the marks of τ) and over mark strings (on the right-hand side it coarsens the marks in a particular mark string). Because of the lack of linearity, in general $\sum_{\mathbf{y} \in \Delta^*} N_f eature[\tau, G](\mathbf{x}, \mathbf{y}) \neq N_f eature[\tau, \mathbf{x} \circ G]$, which may present as a surprise to WFST practitioners. On the other hand, this WFST property is preserved in N_{literal}: $\sum_{\mathbf{y} \in \Delta^*} N_{\text{literal}}[\tau, G](\mathbf{x}, \mathbf{y}) N_{\text{literal}}[\tau, \mathbf{x} \circ G]$.

Speaking very crudely, the behavior of $N_{\text{feature}}[]$ is somewhat similar to the Boolean semiring because of its use of the string-set semiring:[14] $N_{\text{feature}}[\tau_1, G] + N_{\text{feature}}[\tau_2, G] \neq N_{\text{feature}}[\tau_1 \mid \tau_2, G]$; and $N_{\text{feature}}[\tau \mid \tau, G] = N_{\text{feature}}[\tau, G]$.

$N_{\text{literal}}[]$ on the other hand is more similar to the conventional real semiring, in that $N_{\text{feature}}[\tau_1, G] + N_{\text{feature}}[\tau_2, G] = N_{\text{feature}}[\tau_1 \mid \tau_2, G]$.[15] Therefore, $N_{\text{literal}}[]$ may be a good choices when linearity is expected.[16]

[14] To be more precise, it is possible to simulate Boolean WFSTs using feature neuralized NFSTs where $G(\boldsymbol{\omega}) \in \{0, 1\}$.

[15] Assuming $N_{\text{feature}}[\tau_1, G]$ and $N_{\text{feature}}[\tau_2, G]$ are both real numbers.

[16] On the other hand, MFSTs can always be engineered to ensure that each path has its own unique mark string (*e.g.*, by encoding arc indices as a part of every arc's marks). We will make use of such techniques to recover familiar regular expression operations for feature neuralized NFSTs in § 8.

7.2.4.3 Interpretation as probabilities

In §7.2.4.1 we introduced neuralization operators, which allow us to map sets of mark strings (represented by MFSTs) to non-negative scalars (*i.e.*, NFST machine weights). Here we further discuss how we can interpret the NFST machine weights as probabilities, which are crucial in many machine learning applications.

In this thesis, we work with the following probability space $(\Omega^*, \mathcal{F}, P)$:

Sample space. We let the sample space be the set of all mark strings Ω^*.

Event space. We let the event space $\mathcal{F}$ be the set of all subsets of Ω^* as the event space.

Probability function. We require the probability measure $P : \mathcal{F} \longrightarrow [0, 1]$ satisfy that

$$P(\mathbf{s}) = \sum_{\omega \in \mathbf{s}} p(\omega), \tag{7.4}$$

where $\mathbf{s} \in \mathcal{F}$, and $p : \Omega^* \longrightarrow [0, 1]$ is a probability distribution over Ω^*.

NFST machine weights as event probabilities. With the probability space $(\Omega^*, \mathcal{F}, P)$ introduced above, feature-neuralized NFSTs of the form $N_{\text{feature}}[\tau, p]$ can be regarded as event probabilities — specifically, $N_{\text{feature}}[\tau, p]$ is the probability that a mark string (distributed according to p) is recognized by τ_Ω. This suggests that if we parametrize the mark string scoring function (denoted as G in §7.2.4) as a normalized distribution over mark strings, a reasonable optimization objective will be to increase the probability that a randomly sampled mark string is recognized by 'good' MFSTs (*i.e.*, MFSTs whose mark projections

contain reasonable good string transduction annotations), and to decrease the probability that a randomly sampled mark string is recognized by 'bad' MFSTs (*i.e.*, MFSTs whose mark projections do not contain many useful string transduction annotations). Moreover, since the mark string scoring function p is now a *normalized* distribution over mark strings,[17] the promotion of good mark strings will simultaneously mean the demotion of bad mark strings: so one does not need to explicitly decrease the probability of bad MFSTs during training.

There may not be a distribution over input-output string pairs. The way we interpret NFST machine weights above implies that an NFST (τ, G) may not define a distribution over input-output pairs $(\mathbf{x}, \mathbf{y}) \in \Sigma^* \times \Delta^*$: for example, if τ accepts every $(\mathbf{x}, \mathbf{y}) \in \Sigma^* \times \Delta^*$, then the partition function $\sum_{(\mathbf{x},\mathbf{y})\in\Sigma^*\times\Delta^*} N[\tau, G](\mathbf{x}, \mathbf{y})$ may diverge. (τ, G) in this case defines a uniformly weighted relation between Σ^* and Δ^*. However, even when there is not an injective mapping from mark strings to input-ouput string pairs, G still defines a distribution over mark strings, where each observed $(\mathbf{x}, \mathbf{y})$ can be seen as a partial observation of a mark string. That is, while the the mark string is not fully observed, it is observed to be in the set $(\mathbf{x}\circ\tau\circ\mathbf{y})_\Omega$. This partial observation contributes a summand of $\log N[\tau, G](\mathbf{x}, \mathbf{y})$ to the log-likelihood. But since in this scenario the mark strings $\in (\mathbf{x}\circ\tau\circ\mathbf{y})_\Omega$ do not encode information about $(\mathbf{x}, \mathbf{y})$, we cannot decode the input-output pair using these mark strings alone. Therefore, MFSTs where that such mapping exists (*i.e.*, an injective mapping from

[17] Note that languages of strings that are not recognized by any MFSTs are also contained in the sample space, and may have positive probabilities under p.

mark strings to the input-ouput string pairs they encode) will be useful in such use cases.

We have discussed how machine weights of feature neuralized NFSTs can be seen as event probabilities, if their mark string scoring function is a normalized distribution over Ω^*. But how about feature neuralized NFSTs whose mark string scoring functions are *not* normalized distributions over Ω^*?

NFST machine weights can be seen as *unnormalized* probabilities, when the mark string scoring function is not a distribution over Ω^*. For example, suppose mark string scoring function G does not define a normalized distribution over Ω^*, we can simply define $p(\boldsymbol{\omega}) = {}^{G(\omega)}\!/_{Z}$, where $Z = \sum_{\omega' \in \Omega^*} G(\boldsymbol{\omega})$. And since p is a normalized distribution over Ω^*, and that for any τ, $N_{\text{feature}}[\tau, G] = N_{\text{feature}}[\tau, p] \cdot Z$, we can treat $N_{\text{feature}}[\tau, G]$ as an unnormalized probability (that a mark string is recognized by τ_Ω). However, we will show in §7.3 that certain computational challenges arise during parameter estimation, if we do not parametrize G as a normalized distribution.

7.3 Expressiveness of NFSTs

NFSTs is a powerful model family. In this section, we argue that due to the formalism's power, we must limit ourselves to less expressive parametric families of mark string scoring functions, if we would like to train and evaluate them as machine learning models. Specifically, we will show that if we choose to parametrize the mark string scoring function G as an unnormalized probability distribution over strings (*e.g.*, the EC-complete parametric

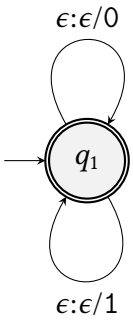
families introduced in §2.3.2), then the machine weights of feature neuralized NFSTs are generally uncomputable.

We will further demonstrate that it is possible restrict ourselves to parametric families where we can train NFSTs as machine learning models relatively fast (*e.g.*, with the limitations mentioned in §7.2.4.3), while still ensuring the induced model family is still much more powerful than common neural sequence models.

EC **weighted languages are too powerful as mark string scoring functions.** EC weighted languages (§2.2.3) can represent unnormalized distributions over strings. In the following, we show that assuming the mark string scoring function $G \in \text{EC}$, the feature neuralized NFST weight $N_{\text{feature}}[\tau, G]$ is generally uncomputable:

Proposition 7.3.1. *There exists an MFST τ, and $G \in \text{EC}$, such that assuming ZFC is consistent, there is no algorithm that provably computes $N_{\text{feature}}[\tau, G]$.*

MFST 7.3.4: MFST used in proof of Proposition 7.3.1



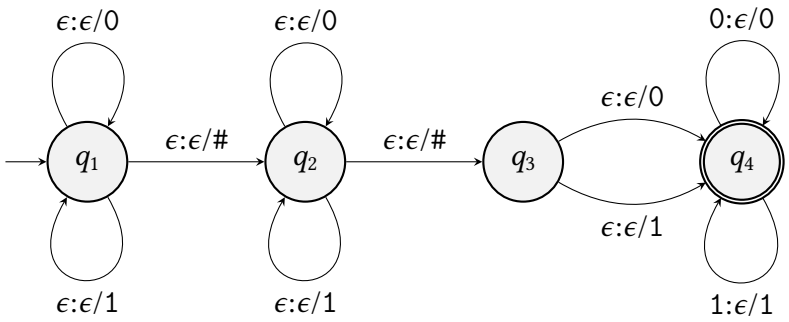
Proof. Let τ be an MFST that transduces ϵ into ϵ, with the mark strings $\in (0|1)^*$ (MFST 7.3.4).

And let G be a function of strings $\boldsymbol{\omega} \in \Omega^* = \mathbb{B}^*$, where $G(\boldsymbol{\omega}) = 1$ if $\tilde{b}(\boldsymbol{\omega}) = 2$, where $\tilde{b}$ is the weighted language introduced in Theorem 5.2.2, and $G(\boldsymbol{\omega}) = 0$ otherwise. Since $\tilde{b} \in \text{EC}$, we can construct $G \in \text{EC}$. By the same argument presented in our proof of Theorem 5.2.2, we know that we could have proven, or disproven ZFC, if there were any provably correct algorithm that could compute $N_{\text{feature}}[\tau, G] = \sum_{\boldsymbol{\omega} \in \mathbb{B}^*} G(\boldsymbol{\omega})$. □

ELNCP **mark string scoring functions make expressive NFSTs.** We have shown that EC weighted languages are too expressive as mark string scoring functions. However, we identify ELNCP as a good balance between tractability and expressiveness. With ELNCP mark string scoring functions, NFSTs can be used to model any decision problem ∈ NP:[18]

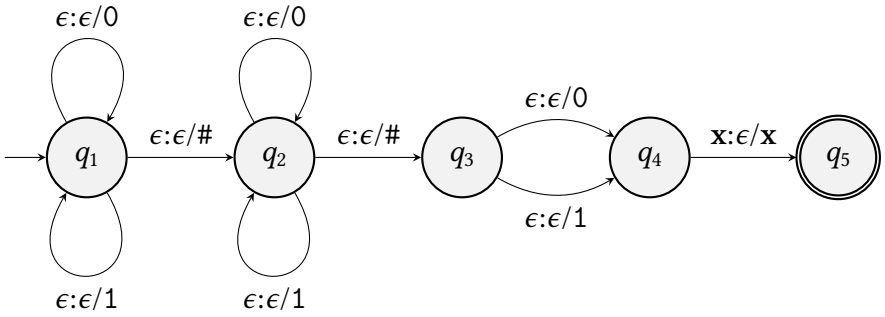
Proposition 7.3.2. *Given any language $L \in \text{NP}$, there exists an MFST τ, and $G \in \text{ELNCP}$, such that $N_{\text{feature}}[\tau, G](\mathbf{x}, \epsilon) > 0 \iff \mathbf{x} \in L$.*

MFST 7.3.5: τ in proof of Proposition 7.3.2



[18] Proposition 7.3.2 can be trivially strengthened to show that NFSTs with ELNCP mark scoring functions can be used to model any decision problem ∈ NP/poly. We choose to present results for the more familiar NP case.

MFST 7.3.6: $\mathbf{x} \circ \tau \circ \epsilon$ in proof of Proposition 7.3.2



Proof. Let $L \in \text{NP}$ be an unweighted language. By Lemma 3.3.4 we know that there exists an ELNCP weighted language p, which has a support L over all strings of the form $\mathbf{a}\#\mathbf{b}\#c\mathbf{d}$, where $\mathbf{a} \in \mathbb{B}^*, \mathbf{b} \in \mathbb{B}^*, c \in \mathbb{B}, \mathbf{d} \in \mathbb{B}^*$. L is the mark projection of τ (shown as MFST 7.3.5).

Let $L_{\mathbf{x}} \subseteq L$ be an unweighted language over strings of the form $\mathbf{a}\#\mathbf{b}\#c\mathbf{x}$. $L_{\mathbf{x}}$ is captured by the regular language $(0 \mid 1)^*\#(0 \mid 1)^*\#(0 \mid 1)\mathbf{x}$, which is the mark projection of $(\mathbf{x} \circ \tau \circ \epsilon)$ (MFST 7.3.6). It follows that $N_{\text{feature}}[\tau, p](\mathbf{x}, \epsilon) = N_{\text{feature}}[\mathbf{x} \circ \tau \circ \epsilon, p] = \sum_{\omega \in (\mathbf{x} \circ \tau \circ \epsilon)_\Omega} p(\omega) = \sum_{\omega \in L_{\mathbf{x}}} p(\omega) > 0$ iff $\mathbf{x} \in L$. □

Proposition 7.3.2 implies that even when we restrict ourselves to ELNCP mark scoring functions, NFSTs are still powerful enough to escape the limitations of neural autoregressive sequence models (which cannot even model all languages $\in$ P as their supports). They also surpass the expressiveness of energy-based sequence models (which cannot model any language $\in$ NP as their supports, if NP $\not\subseteq$ P/poly).

7.4 Inference and parameter estimation

The log NFST machine weight $\log N[\tau, G]$, and its gradients with regard to a parametric $G_{\boldsymbol{\theta}}$: $\nabla_{\boldsymbol{\theta}} \log N[\tau, G_{\boldsymbol{\theta}}]$, are key quantities that we need to estimate during inference and parameter estimation.

7.4.1 Inference

We have an estimator of $\log N[\tau, G]$ based on importance sampling:

$$\begin{aligned}
\log N[\tau, G] &\triangleq \log \sum_{\boldsymbol{\omega} \in \Omega^*} G(\boldsymbol{\omega}) \\
&= \log \sum_{\boldsymbol{\omega} \in \Omega^*} {}^{G(\boldsymbol{\omega})}\!/_{q(\boldsymbol{\omega})} q(\boldsymbol{\omega}) \\
&= \log \mathop{\mathbb{E}}_{\boldsymbol{\omega} \sim q} [\frac{G(\boldsymbol{\omega})}{q(\boldsymbol{\omega})}] \\
&\approx \log \sum_{m=1}^{M} \frac{G(\boldsymbol{\omega}^{(m)})}{q(\boldsymbol{\omega}^{(m)})} - \log M, \qquad (7.5)
\end{aligned}$$

where $M \geq 1, M \in \mathbb{N}$, and $\boldsymbol{\omega}^{(1)} \ldots \boldsymbol{\omega}^{(M)}$ are samples drawn from q. Equation (7.5) is a consistent, but *biased* estimator of $\log N[\tau, G]$: that is, $\lim_{M \to \infty} \log \sum_{m=1}^{M} {}^{G(\boldsymbol{\omega}^{(m)})}\!/_{q(\boldsymbol{\omega}^{(m)})} - \log M = \log N[\tau, G]$, but in general $\mathbb{E}[\log \sum_{m=1}^{M} {}^{G(\boldsymbol{\omega}^{(m)})}\!/_{q(\boldsymbol{\omega}^{(m)})} - \log M] \neq \log N[\tau, G]$. In fact, by Jensen's inequality we know $\mathbb{E}[\log \sum_{m=1}^{M} {}^{G(\boldsymbol{\omega}^{(m)})}\!/_{q(\boldsymbol{\omega}^{(m)})} - \log M] \leq \log N[\tau, G]$.

So, how good is the estimate of equation (7.5)? In general, by Proposition 7.3.1 we will

not have any guarantee on the quality of this estimator — or any other estimators that can be computed in finite time, when $G_\theta \in \text{EC}$ (*i.e.*, when G_θ comes from an EC-complete family). However, when $G_\theta \in \text{ELN}$, the quantity $\log N[\tau, G]$ is still computable by Theorem 5.6.1, and we may still approximate it effectively using equation (7.5), as long as we have a 'good' proposal distribution over $(\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega$. While it is generally difficult to ensure the goodness of our proposal distribution, empirically we find that particle smoothing — the amortized inference scheme we introduced in § 6 — results in good performance in downstream tasks.

7.4.1.1 Conditional distribution over output strings

$N[\tau, G](\mathbf{x}, \mathbf{y})$ can generally be regarded as an unnormalized *joint* probability of input-output pair $(\mathbf{x}, \mathbf{y})$ (§7.2.4.3). However, we are usually more interested in evaluating *conditional* probabilities $p(\mathbf{y} \mid \mathbf{x})$, namely the probability of an output string $\mathbf{y}$ given an input string $\mathbf{x}$. We may also hope to sample from such distributions.

Unfortunately, such conditional probabilities derived from unnormalized joint probabilities are generally difficult: evaluating $\log p(\mathbf{y} \mid \mathbf{x}) = \log {N[\tau,G](\mathbf{x},\mathbf{y})}/{\sum_{\mathbf{y}' \in \Delta^*} N[\tau,G](\mathbf{x},\mathbf{y}')}$ requires evaluating the intractable log partition function $\log Z_{\mathbf{x}} \triangleq \log \sum_{\mathbf{y}' \in \Delta^*} N[\tau, G](\mathbf{x}, \mathbf{y}')$. While we can approximate this quantity using importance sampling, we would need a proposal distribution q_Y over Δ^*; and each drawn sample $\mathbf{y}' \sim q_Y$ results in a different NFST $N[\tau, G](\mathbf{x}, \mathbf{y}') = N[\mathbf{x} \circ \tau \circ \mathbf{y}']$, which further needs to be approximated using another importance sampling scheme.

On the other hand, for MFST's τ where there is a surjective function from τ_Ω to τ_Δ

(defined in §7.2.1), it is possible to estimate $\log Z_{\mathbf{x}} = \log N[\tau_{\mathbf{x}\circ\tau}, G]$, since every mark string in $(\tau_{\mathbf{x}\circ\tau})_\Omega$ uniquely identifies with an output string.[19] This way, we would not need a two-level importance sampling scheme.

7.4.2 Parameter estimation

Given an MFST τ and parametric mark string scoring function G_θ, we generally would like to find good parameters $\hat{\theta} \in \Theta$ such that the NFST $(\tau, G_{\hat{\theta}})$ well explains the string transductions of some training dataset of size N: $(\mathbf{x}_1, \mathbf{y}_1) \ldots (\mathbf{x}_N, \mathbf{y}_N)$.

A natural estimator choice is the maximum likelihood estimator:[20]

$$\hat{\theta}_D = \operatorname*{argmax}_\theta \left\{ \sum_{(\mathbf{x},\mathbf{y})\in D} [\log N[\tau, G_\theta](\mathbf{x}, \mathbf{y}) - \log \sum_{(\mathbf{x}',\mathbf{y}')\in\Sigma^*\times\Delta^*} N[\tau, G_\theta](\mathbf{x}', \mathbf{y}')] \right\}, \tag{7.6}$$

where $D = \{(\mathbf{x}_n, \mathbf{y}_n) : 1 \le n \le N\}$ are N samples (typically drawn from the true distribution p^*).

However gradient-based optimization of equation (7.6) requires computing gradients of the log partition function $\nabla_\theta[\log \sum_{(\mathbf{x}',\mathbf{y}')\in\Sigma^*\times\Delta^*} N[\tau, G_\theta](\mathbf{x}', \mathbf{y}')]$, which can be very costly.

[19] Note that this condition is not necessary when $N = N_{\text{literal}}$.

[20] We implicitly assume that such an estimator is applicable, *i.e.*, the likelihood function does not diverge, throughout the discussion in this section. Such property can be guaranteed under certain sequence model parametrizations (*e.g.*. when confined to LN or ELN, which were first introduced in §2.2.4).

Alternatively, we may wish to ignore the partition function,[21] and try to find

$$\hat{\boldsymbol{\theta}}'_D = \operatorname*{argmax}_{\boldsymbol{\theta}} \sum_{(\mathbf{x},\mathbf{y})\in D} \log N[\tau, G_{\boldsymbol{\theta}}](\mathbf{x},\mathbf{y}). \tag{7.7}$$

We can show that $\hat{\boldsymbol{\theta}}_D = \hat{\boldsymbol{\theta}}'_D$ as long as the parametric sequence model family $\boldsymbol{\Theta}$ defines properly normalized probabilities, and that each mark string of τ identifies with at most one $(\mathbf{x},\mathbf{y})$ pair:

Lemma 7.4.1. *Let τ be an MFST, $\boldsymbol{D} = \{(\mathbf{x}_n, \mathbf{y}_n) : (\mathbf{x}_n \circ \tau \circ \mathbf{y}_n)_\Omega \neq \{\}, 1 \leq n \leq N\}$ be N string pairs, and $\boldsymbol{\Theta} = \{\boldsymbol{\theta} : \sum_{\boldsymbol{\omega}\in\Omega^*} G_{\boldsymbol{\theta}}(\boldsymbol{\omega}) = 1\}$ be the set of mark string scoring functions. Suppose that there exists a function $f : \tau_\Omega \longrightarrow \Sigma^* \times \Delta^*$ such that $f(\boldsymbol{\omega}) = (\mathbf{x},\mathbf{y}) \iff \boldsymbol{\omega} \in (\mathbf{x}\circ\tau\circ\mathbf{y})_\Omega$, then $\hat{\boldsymbol{\theta}}_D = \hat{\boldsymbol{\theta}}'_D$.*

Proof. If $\sum_{(\mathbf{x}',\mathbf{y}')\in\Sigma^*\times\Delta^*} N[\tau, G_{\hat{\boldsymbol{\theta}}'_D}](\mathbf{x}',\mathbf{y}') = 1$,[22] then

$$\begin{aligned}
\hat{\boldsymbol{\theta}}_D &= \operatorname*{argmax}_{\boldsymbol{\theta}} \left\{ \sum_{(\mathbf{x},\mathbf{y})\in D} [\log N[\tau, G_{\boldsymbol{\theta}}](\mathbf{x},\mathbf{y}) - \log \sum_{(\mathbf{x}',\mathbf{y}')\in\Sigma^*\times\Delta^*} N[\tau, G_{\boldsymbol{\theta}}](\mathbf{x}',\mathbf{y}')] \right\} \\
&= \operatorname*{argmax}_{\boldsymbol{\theta}} \left\{ \sum_{(\mathbf{x},\mathbf{y})\in D} [\log N[\tau, G_{\boldsymbol{\theta}}](\mathbf{x},\mathbf{y}) - 0] \right\} \\
&= \operatorname*{argmax}_{\boldsymbol{\theta}} \sum_{(\mathbf{x},\mathbf{y})\in D} [\log N[\tau, G_{\boldsymbol{\theta}}](\mathbf{x},\mathbf{y}) \\
&= \hat{\boldsymbol{\theta}}'_D.
\end{aligned}$$

[21] Note that the partition function may not be 1 since $G_{\boldsymbol{\theta}}$ may assign non-zero probabilities to mark strings that are not accepted by any finite-state machine derived from τ. Moreover, the partition function may not exist (*e.g.*, diverge), when multiple $(\mathbf{x},\mathbf{y})$ pairs use the same mark string. We will discuss how to deal with the divergence problem in the next paragraph.

[22] We repeat that given our assumptions, $\sum_{(\mathbf{x}',\mathbf{y}')\in\Sigma^*\times\Delta^*} N[\tau, G_{\hat{\boldsymbol{\theta}}'_D}](\mathbf{x}',\mathbf{y}')$ may not be equal to $\sum_{\boldsymbol{\omega}\in\Omega^*} G_{\boldsymbol{\theta}}(\boldsymbol{\omega})$. See footnote 21.

Note that $\forall \mathcal{D}, \sum_{(\mathbf{x}',\mathbf{y}')\in\Sigma^*\times\Delta^*} N[\tau, G_{\hat{\theta}'_D}](\mathbf{x}',\mathbf{y}') = 1$. We prove this by contradiction: Begin by assuming to the contrary that there exists $\mathcal{D}'$ such that $\sum_{(\mathbf{x}',\mathbf{y}')\in\Sigma^*\times\Delta^*} N[\tau, G_{\hat{\theta}'_{D'}}](\mathbf{x}',\mathbf{y}') \neq 1$. By our assumption that each mark string $\in \tau_\Omega$ identifies with at most one $(\mathbf{x},\mathbf{y})$ pair, we know $\sum_{(\mathbf{x}',\mathbf{y}')\in\Sigma^*\times\Delta^*} N[\tau, G_{\hat{\theta}'_{D'}}](\mathbf{x}',\mathbf{y}') < 1$. Therefore, $\exists \boldsymbol{\omega}' \in \Omega^*$, such that $G_{\hat{\theta}'}(\boldsymbol{\omega}') > 0$, and $\forall (\mathbf{x},\mathbf{y}) \in \Sigma^* \times \Delta^*, \boldsymbol{\omega}' \notin (\mathbf{x}\circ\tau\circ\mathbf{y})_\Omega$.

Let $(\mathbf{x}',\mathbf{y}') \in \mathcal{D}$ be a string pair. There exists $\boldsymbol{\omega}'' \in (\mathbf{x}_1\circ\tau\circ\mathbf{y}')_\Omega$ by our assumption. We will construct a new mark string scoring function that allocates the probability mass that $G_{\hat{\theta}'_{D'}}$ assigned to $\boldsymbol{\omega}'$ onto the probability mass assigned to $\boldsymbol{\omega}''$. Specifically, let $G_{\hat{\theta}''_{D'}}$ be a mark string scoring function where

$$G_{\hat{\theta}''_{D'}}(\boldsymbol{\omega}) = \begin{cases} G_{\hat{\theta}'_{D'}}(\boldsymbol{\omega}) + G_{\hat{\theta}'_{D'}}(\boldsymbol{\omega}') & \text{if } \boldsymbol{\omega} = \boldsymbol{\omega}'' \\ 0 & \text{if } \boldsymbol{\omega} = \boldsymbol{\omega}' \\ G_{\hat{\theta}'_{D'}}(\boldsymbol{\omega}) & \text{otherwise.} \end{cases}$$

Since $\sum_{\boldsymbol{\omega}\in\Omega^*} G_{\hat{\theta}''_{D'}}(\boldsymbol{\omega}) = \sum_{\boldsymbol{\omega}\in\Omega^*} G_{\hat{\theta}'_{D'}}(\boldsymbol{\omega}) = 1$, $\hat{\theta}''_{D'} \in \Theta$.

Finally, we have

$$\begin{aligned} \sum_{(\mathbf{x},\mathbf{y})\in D'} \log N[\tau, G_{\hat{\theta}''_{D'}}](\mathbf{x},\mathbf{y}) &= \sum_{(\mathbf{x},\mathbf{y})\in D'} \log N[\tau, G_{\hat{\theta}'_{D'}}](\mathbf{x},\mathbf{y}) + G_{\hat{\theta}'_{D'}}(\boldsymbol{\omega}') \\ &> \sum_{(\mathbf{x},\mathbf{y})\in D'} \log N[\tau, G_{\hat{\theta}'_{D'}}](\mathbf{x},\mathbf{y}), \end{aligned}$$

which contradicts our definition of $\hat{\theta}'_{D'}$ (equation (7.7)). □

When does Lemma 7.4.1 hold? Lemma 7.4.1 implies that we can opt for the cheaper estimator in equation (7.7), as long as:

- it is possible to extract from mark strings $\in \tau_\Omega$ the input/output strings they transduce; and that
- we work with a scoring function family that yields normalized string probabilities.

The first condition holds when we adopt a topology like that of MFST 7.2.1, where each input and output symbol is kept in the mark string. The second condition holds when $\boldsymbol{\Theta} \subseteq$ ELNCP.

7.4.2.1 Gradient-based learning

Lemma 7.4.1 suggests that a good training objective would be to minimize the log marginal likelihood. Namely, we seek to minimize

$$\mathcal{L}(\boldsymbol{\theta}) = - \mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim p^*} [\log N[\tau, G_{\boldsymbol{\theta}}](\mathbf{x}, \mathbf{y})]. \tag{7.8}$$

However, the gradients of equation (7.8)

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) &= - \mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim p^*} [\nabla_{\boldsymbol{\theta}} [\log N[\tau, G_{\boldsymbol{\theta}}](\mathbf{x}, \mathbf{y})]] \\
&= \mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim p^*} [\nabla_{\boldsymbol{\theta}} [\log N[\mathbf{x} \circ \tau \circ \mathbf{y}, G_{\boldsymbol{\theta}}]]] \\
&= \mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim p^*} [\nabla_{\boldsymbol{\theta}} \left[\log \sum_{\boldsymbol{\omega} \in (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega} G_{\boldsymbol{\theta}}(\boldsymbol{\omega}) \right]]
\end{aligned} \tag{7.9}$$

requires a sum over generally infinitely many mark strings $\in (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega$, which poses a problem for automatic differentiation procedures, which generally assume finite computation graphs.

First attempt: biased gradients of the true loss. It is possible to rewrite the term $\nabla_{\boldsymbol{\theta}} \left[\log \sum_{\boldsymbol{\omega} \in (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega} G_{\boldsymbol{\theta}}(\boldsymbol{\omega})\right]$ in equation (7.9) as

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} \left[\log \sum_{\boldsymbol{\omega} \in (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega} G_{\boldsymbol{\theta}}(\boldsymbol{\omega})\right] &= \mathop{\mathbb{E}}_{\boldsymbol{\omega} \sim p_{\boldsymbol{\theta}, |\mathbf{x} \circ \tau \circ \mathbf{y}_\Omega}} \left[\nabla_{\boldsymbol{\theta}} \left[\log G_{\boldsymbol{\theta}}(\boldsymbol{\omega})\right]\right] \\ &= \mathop{\mathbb{E}}_{\boldsymbol{\omega} \sim q_{\mathbf{x},\mathbf{y}}} \left[\frac{p_{\boldsymbol{\theta}, |\mathbf{x} \circ \tau \circ \mathbf{y}_\Omega}(\boldsymbol{\omega})}{q_{\mathbf{x},\mathbf{y}}(\boldsymbol{\omega})} \nabla_{\boldsymbol{\theta}} \left[\log G_{\boldsymbol{\theta}}(\boldsymbol{\omega})\right]\right], \end{aligned} \tag{7.10}$$

where

$$p_{\boldsymbol{\theta}, |\mathbf{x} \circ \tau \circ \mathbf{y}_\Omega}(\boldsymbol{\omega}) = \frac{G_{\boldsymbol{\theta}}(\boldsymbol{\omega})}{\sum_{\boldsymbol{\omega}' \in (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega} G_{\boldsymbol{\theta}}(\boldsymbol{\omega}')}, \tag{7.11}$$

and $q_{\mathbf{x},\mathbf{y}}$ is any distribution that has a support that is no smaller than $p_{\boldsymbol{\theta}, |\mathbf{x} \circ \tau \circ \mathbf{y}_\Omega}$.

The term $\nabla_{\boldsymbol{\theta}} \left[\log G_{\boldsymbol{\theta}}(\boldsymbol{\omega})\right]$ in equation (7.10) can be derived mechanically assuming $G_{\boldsymbol{\theta}}(\boldsymbol{\omega})$ is differentiable with regard to $\boldsymbol{\theta}$, and can be computed in finite time. However, since the term $\sum_{\boldsymbol{\omega}' \in (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega} G_{\boldsymbol{\theta}}(\boldsymbol{\omega}')$ is generally intractable, we still cannot compute equation (7.11) exactly. Therefore, we would not be able to approximate equation (7.10) using unnormalized importance sampling. We could resort to normalized importance sampling to approximate equation (7.10):

$$\mathop{\mathbb{E}}_{\boldsymbol{\omega}\sim q_{\mathbf{x},\mathbf{y}}}\left[\frac{p_{\boldsymbol{\theta},|\mathbf{x}\circ\tau\circ\mathbf{y}_\Omega}(\boldsymbol{\omega})}{q_{\mathbf{x},\mathbf{y}}(\boldsymbol{\omega})}\nabla_{\boldsymbol{\theta}}\left[\log G_{\boldsymbol{\theta}}(\boldsymbol{\omega})\right]\right] \approx \frac{1}{ZM}\sum_{m=1}^{M}\frac{G_{\boldsymbol{\theta}}(\boldsymbol{\omega}^{(m)})}{q_{\mathbf{x},\mathbf{y}}(\boldsymbol{\omega})}\nabla_{\boldsymbol{\theta}}\left[\log G_{\boldsymbol{\theta}}(\boldsymbol{\omega})\right], \tag{7.12}$$

where $\boldsymbol{\omega}^{(1)} \ldots \boldsymbol{\omega}^{(M)}$ are drawn from $q_{\mathbf{x},\mathbf{y}}$. Equation (7.12) is a biased (albeit consistent) estimate. Therefore, for any finite M, the approximation in equation (7.12) yields a biased estimate of equation (7.10). While the bias issue might somewhat be mitigated using a large value for M, the danger that parameter updates under SGD optimization may be ruined with an M that is too small strains compute budget.

Second attempt: unbiased gradients of an loss upper bound. Our second attempt follows the familiar variational inference recipe (Mnih and Gregor, 2014; Kingma and Welling, 2014). Instead of biased estimates of $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$, in this work we consider an *unbiased* estimate of $\nabla_{\boldsymbol{\theta}}\mathcal{L}'(\boldsymbol{\theta})$, where $\forall\boldsymbol{\theta}\in\Theta, \mathcal{L}'(\boldsymbol{\theta}) \geq \mathcal{L}(\boldsymbol{\theta})$ — that is, $\mathcal{L}'$ is an upper bound of $\mathcal{L}$. One such upper bound can be derived using Jensen's inequality:

$$
\begin{aligned}
\mathcal{L}(\boldsymbol{\theta}) &= -\mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim p^*}[\log N[\tau, G_{\boldsymbol{\theta}}](\mathbf{x},\mathbf{y})] \\
&= -\mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim p^*}[\log \sum_{\boldsymbol{\omega}\in(\mathbf{x}\circ\tau\circ\mathbf{y})_\Omega} G_{\boldsymbol{\theta}}(\boldsymbol{\omega})] \\
&= -\mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim p^*}[\log \mathop{\mathbb{E}}_{\boldsymbol{\omega}\sim q_{\mathbf{x},\mathbf{y}}}[{}^{G_{\boldsymbol{\theta}}(\boldsymbol{\omega})}\!/_{q_{\mathbf{x},\mathbf{y}}(\boldsymbol{\omega})}]] \\
&\leq -\mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim p^*}[\mathop{\mathbb{E}}_{\boldsymbol{\omega}\sim q_{\mathbf{x},\mathbf{y}}}[\log {}^{G_{\boldsymbol{\theta}}(\boldsymbol{\omega})}\!/_{q_{\mathbf{x},\mathbf{y}}(\boldsymbol{\omega})}]] \qquad (7.13) \\
&= \mathcal{L}'(\boldsymbol{\theta}).
\end{aligned}
$$

We can easily push the gradients of $\mathcal{L}'(\boldsymbol{\theta})$ through:

$$
\begin{aligned}
\nabla_{\boldsymbol{\theta}}\mathcal{L}'(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}}\left[-\mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim p^*}[\mathop{\mathbb{E}}_{\boldsymbol{\omega}\sim q_{\mathbf{x},\mathbf{y}}}[\log {}^{G_{\boldsymbol{\theta}}(\boldsymbol{\omega})}\!/_{q_{\mathbf{x},\mathbf{y}}(\boldsymbol{\omega})}]]\right] \\
&= -\mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim p^*}[\mathop{\mathbb{E}}_{\boldsymbol{\omega}\sim q_{\mathbf{x},\mathbf{y}}}[\nabla_{\boldsymbol{\theta}}[\log {}^{G_{\boldsymbol{\theta}}(\boldsymbol{\omega})}\!/_{q_{\mathbf{x},\mathbf{y}}(\boldsymbol{\omega})}]]], \qquad (7.14) \\
& \qquad (7.15)
\end{aligned}
$$

where the inner expectation $\mathbb{E}_{\boldsymbol{\omega}\sim q_{\mathbf{x},\mathbf{y}}}[\nabla_{\boldsymbol{\theta}}[\log {}^{G_{\boldsymbol{\theta}}(\boldsymbol{\omega})}\!/_{q_{\mathbf{x},\mathbf{y}}(\boldsymbol{\omega})}]]$ can be approximated with:

$$
\mathop{\mathbb{E}}_{\boldsymbol{\omega}\sim q_{\mathbf{x},\mathbf{y}}}[\nabla_{\boldsymbol{\theta}}[\log {}^{G_{\boldsymbol{\theta}}(\boldsymbol{\omega})}\!/_{q_{\mathbf{x},\mathbf{y}}(\boldsymbol{\omega})}]] \approx \frac{1}{M'}\sum_{m=1}^{M'} \log {}^{G_{\boldsymbol{\theta}}(\boldsymbol{\omega})}\!/_{q_{\mathbf{x},\mathbf{y}}(\boldsymbol{\omega}^{(m)})}, \qquad (7.16)
$$

where $\boldsymbol{\omega}^{(1)} \ldots \boldsymbol{\omega}^{(M')}$ are samples drawn from $q_{\mathbf{x},\mathbf{y}}$. Note that equation (7.16) is an unbiased

estimate (unlike equation (7.12)). Due to the linearity of expectations, we can also derive an unbiased estimate of $\nabla_\theta \mathcal{L}'(\theta)$.

Tighter upper bounds with importance weighting. A possible concern of optimizing for $\mathcal{L}'(\theta)$ in equation (7.14) is that the upper bound might be loose. To address this problem, we further tighten the upper bound $\mathcal{L}'$ using importance weighting estimators (IWAE, Burda, Grosse, and Salakhutdinov (2016)).

We define

$$\mathcal{L}'_K(\theta) = -\mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim p^*}\Big[\mathop{\mathbb{E}}_{\boldsymbol{\omega}^{(1)}\ldots\boldsymbol{\omega}^{(K)}\sim q_{\mathbf{x},\mathbf{y}}}\Big[\log \frac{1}{K}\sum_{k=1}^{K} {}^{G_\theta(\boldsymbol{\omega}^{(k)})}\!/_{q_{\mathbf{x},\mathbf{y}}(\boldsymbol{\omega}^{(k)})}\Big]\Big]. \tag{7.17}$$

By Burda, Grosse, and Salakhutdinov (2016, Theorem 1), we have

Proposition 7.4.2.

$$\forall K_1 \in \mathbb{N}, \forall K_2 \in \mathbb{N}, \mathcal{L}'_{K_1}(\theta) \geq \mathcal{L}'_{K_2}(\theta) \geq \mathcal{L}(\theta) \iff K_1 \leq K_2.$$

Proposition 7.4.2 implies that we can tighten an upper bound of $\mathcal{L}$: $\mathcal{L}'_K$ at the expense of compute (*i.e.*, to compute the term $\log \frac{1}{K}\sum_{k=1}^{K} {}^{G_\theta(\boldsymbol{\omega}^{(k)})}\!/_{q_{\mathbf{x},\mathbf{y}}(\boldsymbol{\omega}^{(k)})}$ in equation (7.17)). Note that unlike $\nabla_\theta \mathcal{L}(\theta)$ which has a biased estimate (equation (7.12)), for any $K \in \mathbb{N}$, $\nabla_\theta \mathcal{L}'_K(\theta)$ has

an unbiased estimate

$$\nabla_{\theta} \mathcal{L}'_K(\boldsymbol{\theta}) = - \mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim p^*} [\mathcal{L}'_{\mathbf{x},\mathbf{y},K}(\boldsymbol{\theta})], \tag{7.18}$$

where

$$\mathcal{L}'_{\mathbf{x},\mathbf{y},K}(\boldsymbol{\theta}) = \mathop{\mathbb{E}}_{\boldsymbol{\omega}\sim q_{\mathbf{x},\mathbf{y}}} [\nabla_{\theta} \left[\log \frac{1}{K} \sum_{k=1}^{K} {}^{G_{\theta}(\boldsymbol{\omega}^{(k)})}\!/_{q_{\mathbf{x},\mathbf{y}}(\boldsymbol{\omega}^{(k)})} \right]]. \tag{7.19}$$

7.5 Experiments

In this section we evaluate NFSTs over several datasets, on various metrics. Our overall goal is to answer the following questions:

Are NFSTs effective string transduction models? (§7.5.3) Being autoregressive latent-variable sequence models, theoretic results in § 3 and §7.3 suggest NFSTs are much more powerful than common string transduction paradigms, such as neural autoregressive sequence models (§2.1.3) and weighted finite-state machines (§2.1.2). But we would still like to know whether the difference in theoretical capacity translates into observable performance in real-life string transduction tasks.

Do topologies that are engineered using prior knowledge help? (§7.5.4) As we note in §7.1.1, we can induce inductive bias in NFSTs by designing specific marked finite-state transducers τ that encode domain knowledge. In this work, we compare the

empirical performance between NFSTs that adopt a τ that encodes prior knowledge about the task at hand, against τ's that are task-agnostic.

Are mark strings interpretable? (§7.5.5) Given NFST (τ, G_θ), a mark string $\boldsymbol{\omega} \in (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega$ identifies with a path of τ that transduces between string pair $(\mathbf{x}, \mathbf{y})$. $N[\tau, G_\theta](\mathbf{x}, \mathbf{y})$ thus weights paths in the machine $\mathbf{x} \circ \tau \circ \mathbf{y}$: the path that identifies with mark string $\boldsymbol{\omega}$ has weight $G_\theta(\boldsymbol{\omega})$. We would like to know if the NFST path weights are *interpretable*, in the sense that higher path weights correspond to more sensible transduction explanations, according to prior knowledge.

7.5.1 Datasets

We conduct experiments on the following datasets:

CMUDICT The CMU pronunciation dictionary (Weide, 2005) is an open-source machine-readable pronunciation dictionary for North American English. The original dataset transcribes word pronunciations using Arpabet. In this work, we adopt a version where pronunciations are further automatically transcribed into IPA symbols (Piperski, 2016). We also prune the dataset for short words and pronunciations.[23]

In this work, we conduct experiments on both transducing from orthographically spelled words to their IPA pronunciations, and also from IPA pronunciations to

[23] Specifically, we only keep words and pronunciations that have at most 20 English characters or IPA symbols.

words. Because of the highly irregular English orthography (Venezky, 2011), we expect that these tasks can illustrate the strengths of expressive model families.

SNIPS The Snips NLU dataset (Coucke et al., 2018) is a collection of voice commands for digital assistants, where each token is tagged with either a pre-defined slot type, or an 'outside' tag. Moreover, each voice command is also labeled for its intent.

The Snips dataset is commonly used to benchmark slot-filling systems. Due to the fact that many slot types are intent-specific (*i.e.*, they only co-occur with certain intents), we expect that systems that encode such prior knowledge will have an edge over systems that are agnostic to such information.

DAKSHINA The Dakshina dataset (Roark et al., 2020) is a corpus of 12 South Asian languages. In this work, we take their romanization lexica (*i.e.*, native script-romanization pairs), and experiment with transduction in either direction (*i.e.* from romanization to native scripts, and vice versa).

Much like grapheme-phoneme transduction in the CMUDICT experiments, we expect the romanization-native script transduction can often be explained by a monotonic alignment between the same word in a romanized form, and native script. In this work, we focus on two languages which are transcribed using Perso-Arabic scripts: Sindhi and Urdu. These two languages were shown to be the most difficult, as observed by Roark et al. (2020).

Because of compute constraints, we do not use the full CMUDICT and DAKSHINA datasets.

Rather, for these two datasets, we randomly sample $2,000/1,000/1,000$ string pairs (without replacement) as training/validation/test datasets.

7.5.2 Experimental setup

7.5.2.1 Amortized inference with particle smoothing

Both training and inferring from NFSTs require estimating NFST log weights: $\log N[\tau, G]$. As we note in §7.4.1, we seek to approximate this quantity (specifically, equation (7.5)) using importance sampling.

The variance of an importance sampling estimate depends heavily on the similarity between the proposal distribution q and the target distribution (which has unnormalized probabilities $G(\boldsymbol{\omega})$ in our case). We train q to approximate the target distribution following §6.5, in hope to minimize variance.

Similar to our treatment in § 6, we seek to minimize such variance by parametrizing q also as an expressive sequence model (which we denote as $q_{\boldsymbol{\phi}}$ where $\boldsymbol{\phi} \in \boldsymbol{\Phi}$ is a parameter vector), and also minimizing the KL-divergence between $q_{\boldsymbol{\phi}}$ and the target distribution $p_\tau(\boldsymbol{\omega}) \propto G(\boldsymbol{\omega})$. Also similar to the scenario encountered in § 6, the latent variables we seek to sample are strings *monotonically aligned* with some given string pairs.

Specifically, we need to sample mark strings $\boldsymbol{\omega}$ from a distribution with support $(\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega$[24] and unnormalized probabilities $p(\boldsymbol{\omega}) \propto G_\theta(\boldsymbol{\omega})$, where $\mathbf{x}$ and $\mathbf{y}$ are some given

[24] We know $\mathbf{y}$ because we rerank possible output candidates during decoding (§7.5.3.2).

input and output strings. In other words, we would like to sample from

$$p(\boldsymbol{\omega} \mid (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega) \triangleq \frac{G_\theta(\boldsymbol{\omega})}{\sum_{\boldsymbol{\omega}' \in (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega} G_\theta(\boldsymbol{\omega})} \tag{7.20}$$

The denominator of equation (7.20) is generally intractable because of the expressiveness of G_θ. The regular language $(\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega$[25] can have an infinite support, so we fall back on normalized importance sampling. That is, we draw M samples from a proposal distribution $q_{\mathbf{x},\mathbf{y}}$ over the language of $(\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega$, and weight each string $\boldsymbol{\omega}^{(m)}$ by $w_m \propto \tilde{p}(\boldsymbol{\omega}^{(m)})/q(\boldsymbol{\omega}^{(m)})$ where $\sum_{m=1}^{M} w_m = 1$. The resulting **weighted ensemble of particles** $\hat{p}$ forms a sampling-based approximation to $p(\boldsymbol{\omega} \mid (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega)$. To be precise:

$$\begin{aligned} p(\boldsymbol{\omega} \mid (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega) &= \mathop{\mathbb{E}}_{\boldsymbol{\omega}' \sim q_{\mathbf{x},\mathbf{y}}} \left[\frac{p(\boldsymbol{\omega}' \mid (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega)}{q_{\mathbf{x},\mathbf{y}}(\boldsymbol{\omega}')} \cdot \mathbb{I}(\boldsymbol{\omega} = \boldsymbol{\omega}') \right] \\ &\approx \underbrace{\sum_{m=1}^{M} w_m \cdot \mathbb{I}(\boldsymbol{\omega} = \boldsymbol{\omega}^{(m)})}_{\text{call this approximation } \hat{p}(\boldsymbol{\omega} \mid (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega)} \end{aligned} \tag{7.21}$$

$q_{\mathbf{x},\mathbf{y}}$ must depend on $\mathbf{x}$ and $\mathbf{y}$. In this chapter, we follow the **particle smoothing** approach (§ 6), thereby porting that approach to the FST setting. $q_{\mathbf{x},\mathbf{y}}$ is chosen to be a locally normalized distribution on τ',

$$q_{\mathbf{x},\mathbf{y}}(\boldsymbol{\omega}) = \prod_{t=1}^{|\boldsymbol{\omega}|} q_{\mathbf{x},\mathbf{y}}(\omega_t \mid \boldsymbol{\omega}_{<t}), \tag{7.22}$$

[25] We will use $(\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega$ to denote both the regular language it defines, and the regular language's equivalent deterministic finite-state acceptor, interchangeably.

where the $q_{\mathbf{x},\mathbf{y}}(\omega_t \mid \cdots)$ factor is defined to be

$$\frac{\exp(\tilde{g}_{\boldsymbol{\theta}}(\omega_t) + H_{\boldsymbol{\phi}}(\omega_1 \cdots \omega_t))}{\exp(\tilde{g}_{\boldsymbol{\theta}}(\omega') + H_{\boldsymbol{\phi}}(\omega_1 \cdots \omega_{t-1}\omega'))}$$

where the $\tilde{g}_{\boldsymbol{\theta}}$ notation is borrowed from equation (6.2) and ω' now ranges over all competing next symbols ω' that follow from the same state in $(\mathbf{x}\circ\tau\circ\mathbf{y})_\Omega$ (not τ) as ω_t does. Thus, $q_{\mathbf{x},\mathbf{y}}$ may be regarded as a policy for choosing among competing "actions" ω', just as in reinforcement learning. The **lookahead function** $H_{\boldsymbol{\phi}}(\vec{\boldsymbol{\omega}})$[26] is parameterized by $\boldsymbol{\phi}$ and attempts to evaluate whether the path *prefix* $\vec{\boldsymbol{\omega}}$ is compatible with *completing* a path $\boldsymbol{\omega}$ in $(\mathbf{x}\circ\tau\circ\mathbf{y})_\Omega$.

In future work, we are interested in devising a $H_{\boldsymbol{\phi}}$ function that would work well with general X, Y by considering the graph structure of $\tau' = X\circ\tau\circ Y$. In this chapter, however, we consider $\mathbf{x}$ and $\mathbf{y}$ alone, and only if they are single strings.

Let $\vec{\mathbf{h}}$ be the hidden state vector of the model after reading the marks on $\vec{\mathbf{a}}$. Let $\overleftarrow{\mathbf{x}}$ be the suffix of $\mathbf{x}$ that has *not* been consumed by $\vec{\boldsymbol{\omega}}$, and let $\overleftarrow{\mathbf{h}}^{\mathrm{x}}$ be the state of an auxiliary recurrent neural network after reading the symbols of $\overleftarrow{\mathbf{x}}$ in reverse order. Similarly, define $\vec{\mathbf{y}}$ and $\overleftarrow{\mathbf{h}}^{\mathrm{y}}$ likewise.

Note that $\vec{\mathbf{h}}$, $\overleftarrow{\mathbf{h}}^{\mathrm{x}}$, and $\overleftarrow{\mathbf{h}}^{\mathrm{y}}$ are the states of 3 different RNNs that respectively read strings in Ω^*, Σ^*, and Δ^*. The value $H_{\boldsymbol{\phi}}(\boldsymbol{\omega}) \in \mathbb{R}$ is computed by a feed-forward network whose input is the concatenation of these 3 states, or more precisely, the subset of these 3 states that are defined.[27]

[26] $H_{\boldsymbol{\phi}}$ is analogous the H term given in equation (6.11), but may have to summarize infinitely possible outcomes.

[27] In § 6 we define $H = C + \hat{H}$, and this design resembles their design for the C part. In retrospect, we should also have captured the $\hat{H}$ part by allowing the the auxiliary recurrent networks to add up scores

Following §6.5, we train $\boldsymbol{\phi}$ so that the proposal distribution $q_{\mathbf{x},\mathbf{y}}(\boldsymbol{\omega})$ tends to place a high probability on paths that are good—or at least appear to be good—under the model p. A step of training $q_{\mathbf{x},\mathbf{y}}$ consists of approximating $p(\boldsymbol{\omega} \mid (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega)$ by a particle ensemble $\hat{p}(\boldsymbol{\omega} \mid (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega)$, and then adjusting $\boldsymbol{\phi}$ to decrease $\mathrm{KL}(\hat{p} \mid\mid q_{\mathbf{x},\mathbf{y}})$ (as in stochastic gradient descent). In other words, we sample M paths from the current proposal distribution and then try to increase the proposal probability of all of them, but especially those that were given a high importance weight w_m by the actual model.

7.5.2.2 Parameter estimation

We train following Algorithm 2.

7.5.2.3 Hyperparameters

We fix dropout rate to 0.2. We make use of the Adam optimizer throughout, with learning rate fixed to 10^{-3}. We adopt an early-stopping training scheme, with a patience of 20 epochs. And we have $K = 8$ for $\mathcal{L}'_K$ (equation (7.17)), except for ablation studies (§7.5.3.4).

7.5.2.4 Design details of the MFST τ's topology

An MFST τ decides the relation an NFST (in which τ is a component, that is, $(\tau, G_{\boldsymbol{\theta}})$) recognizes. As we said in §7.1.1, τ can be engineered to design the relation an NFST recognized. For some applications — such as slot-filling — we can readily engineer a relation

for the symbols of $\overleftarrow{\mathbf{x}}, \overleftarrow{\mathbf{y}}$ as it reads them (much as (6.2) adds up arc scores). In our FST setting, $\overleftarrow{\mathbf{x}}, \overleftarrow{\mathbf{y}}$ may be slightly different for different ω', in contrast to § 6 where $\hat{H}$ was constant.

Algorithm 2: Training procedure for $G_{\boldsymbol{\theta}}$.

Input:

- τ is an MFST
- $\mathcal{D}_{\text{train}} = \{(\mathbf{x}_1, \mathbf{y}_1) \dots (\mathbf{x}_{|\mathcal{D}_{\text{train}}|}, \mathbf{y}_{|\mathcal{D}_{\text{train}}|})\}$ is the training dataset
- AdamOptimizer : $\boldsymbol{\Theta} \times \boldsymbol{\Theta} \longrightarrow \boldsymbol{\Theta}$ takes as arguments parameters and their gradients, and outputs updated parameters
- $\boldsymbol{\theta}_0 \in \boldsymbol{\Theta}$ are the initial parameters of $G_{\boldsymbol{\theta}}$
- M is a given sample size
- maxEpoch $\in \mathbb{N}$ is the number of epochs to train for

Output: Trained parameters $\boldsymbol{\theta}$

```
foreach epoch ∈ [1 ... maxEpochs] do
    foreach (x, y) ∈ shuffle(D) do
        τ' ← x∘τ∘y;
        Construct distribution q_{x,y}(·) according to equation (7.22);
        u ←  importance sampling estimate of equation (7.19) with M samples;
        θ ← Optimizer(θ, u);
        (Optional) update the parameters of q_{x,y};
    end
end
return θ
```

which does not accept invalid transductions. For example, Snips (MREs 7.5.9–7.5.13) recognizes each intent using its own machine (*e.g.*, PlayMusic), where only the slot types that belong to this unit are recognized. We let τ = Snips for all our snips experiments in this chapter.

MRE 7.5.9: Snips

(PlayMusic|BookRestaurant| ...)

MRE 7.5.10: PlayMusic

(Outside|Album|Artist|Track|...)*

MRE 7.5.11: Album

(ε:B/*B-album*)Word((ε:I/*I-album*)Word)*

MRE 7.5.12: Outside

(ε:O/*outside*)Word

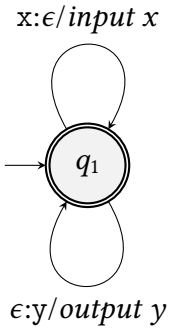
MRE 7.5.13: Word

(a:a/*word-a*|aardvark:aardvark/*word-aardvark*|...)

Likewise, 'latent' marks that indicate state transitions, but do not appear in either input or output projections, may also reflect domain knowledge. Some experiments in this work are designed to evaluate whether such injected prior knowledge is effective. Special τ's are specifically designed for these experiments. We also have an 'agnostic' design that has a single state, with arcs that are either of the form $x : \epsilon/\textit{input-x}$ or $\epsilon : y/\textit{output-y}$, where $x \in \Sigma$ and $y \in \Delta$, such that the MFST recognizes all relation $\in \Sigma^* \times \Delta^*$ (*e.g.*, AgnosticTransduction in MFST 7.5.7).[28]

[28] Such MFSTs are not entirely domain-agnostic since we do know the input and output alphabets Σ and Δ, which may contain symbols that are concatenations of other symbols in set. *E.g.*, the output alphabet of cmudict contains the IPA symbol tʃ, which is also a concatenation of two other symbols of the output alphabet t and ʃ.

MFST 7.5.7: AgnosticTransduction (many arcs omitted for clarity)



7.5.2.5 Implementation details of the mark string scoring function

G_θ is implemented as an ordinary left-to-right 2-layer LSTM language model over Ω^* (Hochreiter and Schmidhuber, 1997), with hidden dimension size 256 a final softmax activation function.

As described in §7.5.2.1, our NFSTs make use of neural particle smoothing network during both parameter estimation and inference to sample mark strings from the MFSA $\mathbf{x} \circ \tau \circ \mathbf{y}$. We parametrize a particle smoothing network to consist of 2-layer bi-directional LSTMs that encode $\mathbf{x}$ and $\mathbf{y}$ respectively, each with hidden dimension size 256.

7.5.3 Effectiveness of NFSTs

7.5.3.1 Metrics

To empirically support our claim that NFSTs are an effective sequence transduction paradigm, we compare their ability to weight string transductions (*i.e.*, input-output string pairs) against common seq2seq models, on the following metrics:

MRR (mean reciprocal rank) looks at a system's relative performance as a reranker. It rules out the confounding factors of search error (*i.e.*, the correct answer not included as a candidate).

Let N be the size of test input strings. MRR is defined as

$$\text{MRR} \triangleq \frac{1}{N} \sum_{n=1}^{N} \frac{1}{\text{rank}_n}, \tag{7.23}$$

where rank_n is the position of the correct output string $\mathbf{y}^{(n)}$ for the n-th input string $\mathbf{x}^{(n)}$, within the n-th list of candidate output strings.[29]

In this work, we first take k-best lists from seq2seq models, and then union each of them with a respective correct output string, to form these candidate output lists. For NFSTs, we use the rank of $N[\tau, G_\theta](\mathbf{x}^{(n)}, \mathbf{y}^{(n)})$ as rank_n within the n-th candidate output list in equation (7.23). For seq2seq baselines, we use $p(\mathbf{y}^{(n)} \mid \mathbf{x}^{(n)})$ as rank_n.

[29] In this work, $\mathbf{y}^{(n)}$ always appear in the candidate list. Nonetheless, common definitions of MRR define $\text{rank}_n = \infty$ when $\mathbf{y}^{(n)}$ does not appear in the list.

EM (exact match) is related to the MRR metric in that EM also looks at reranked candidate lists. Given N list of candidate outputs, ordered by probability of correctness, for N input strings, EM is defined as

$$\text{EM} \triangleq \frac{1}{N} \sum_{n=1}^{N} \mathbb{I}(\text{rank}_n = 1). \tag{7.24}$$

In other words, EM looks at whether the system under evaluation ranks correct output strings from candidate lists first. We still take k-best list from seq2seq models; but unlike the MRR metric, we do *not* union them with the correct output strings. Thus the EM metric provides a sense of how NFSTs would perform in real life as a reranker (without assigning partial credit) — the candidate lists presumably come from computationally cheap models (*e.g.*, seq2seq models); and their search error would be a confounding factor.

In addition to EM, we also evaluate on Slot F1 scores for the SNIPS dataset.

7.5.3.2 Baseline systems

We compare against seq2seq models. Specifically, we adopt encoder-decoder baseline systems of Gorman et al. (2020). All hyperparameters follow their settings, unless otherwise specified. These models are standard encoder-decoder LSTM models with attention, where both the encoder and decoder networks have a hidden size of 512, and an embedding size of 128. The encoder is additionally bidirectional. As for model selection, we do a grid

search of dropout rates $\in (0.1, 0.2, 0.3)$, and batch sizes $\in (256, 512, 1024)$. We choose the best configurations using validation metrics.

Effects of reranking. We use baseline seq2seq models to provide k-best lists for NFST systems. To see whether the exact match metric benefits from such an ensemble setup, we additionally evaluate seq2seq models as rerankers, where the k-best lists are obtained from separately trained seq2seq models. Those seq2seq models are selected using grid search from the same set of hyperparameters of the baseline seq2seq models, except for the random seed[30] We then report these additional exact match numbers for the CMUDICT dataset.

For the SNIPS dataset, we additionally consider published results by E et al. (2019) and Wu et al. (2020). E et al. (2019) is a carefully and specifically designed bidirectional LSTM network that processes the input over multiple passes, for the joint slot filling and intention detection task. They considered slot filling as a tagging task, and had a final CRF layer for decoding label sequences. Wu et al. (2020) is also a neural network with a tailored multiple pass design, but is Transformer-based.

7.5.3.3 Results

Empirically, we find that NFSTs compare very favorably against seq2seq models on every metric and every task.

[30]We used a random seed 1917 for baseline seq2seq models, and 1918 for 'spare' seq2seq models introduced in this paragraph.

	Valid			Test		
	MRR	EM	F1	MRR	EM	F1
NFST	0.937	0.889	93.6	0.949	0.908	93.7
Seq2seq	0.825	0.763	88.5	0.829	0.764	89.0
Hakkani-Tür et al. (2016)	—	—	—	—	—	87.3
E et al. (2019)	—	—	—	—	—	92.2
Wu et al. (2020)	—	—	—	—	—	93.9

Table 7.1: SNIPS results.

Snips. We see in Table 7.1 that on the SNIPS dataset, NFSTs outperform the Seq2seq baseline[31] by a large margin on every metric (with $p < 0.01$ in all cases, under the permutation test). The difference in MRR clearly shows that NFSTs are competitive as rerankers against seq2seq models. Moreover, NFSTs' superior performance on F1 and EM directly suggests that the correct output string often resides in the k-best list of a seq2seq model; and reranking seq2seq output strings using NFSTs is a viable way of boosting a string transduction system's performance.

NFST results are also competitive against existing systems in the literature (E et al., 2019; Hakkani-Tür et al., 2016). While the NFSTs achieve an F1 score 0.2 points lower than Wu et al. (2020), we note that the difference is not statistically significant ($p = 0.76$). The difference between our results and that of E et al. (2019) is statistically significant ($p < 0.01$), however.

[31] While the Seq2seq baseline looks weak when compared against other competitive runs, we note that it does outperform published Bi-LSTM Seq2seq results (Hakkani-Tür et al., 2016).

	Valid		Test	
	MRR	EM	MRR	EM
NFST	0.674	0.533	0.647	0.49
Seq2seq	0.496	0.404	0.472	0.382
Seq2seq reranking another seq2seq	—	0.411	—	0.383

Table 7.2: CMUDICT G2P results.

	Valid		Test	
	MRR	EM	MRR	EM
NFST	0.531	0.367	0.535	0.364
Seq2seq	0.306	0.224	0.303	0.208
Seq2seq reranking another seq2seq	—	0.218	—	0.212

Table 7.3: CMUDICT P2G results.

Cmudict. We evaluate on two transduction directions on the CMUDICT dataset: phoneme-to-grapheme (P2G) and grapheme-to-phoneme (G2P). We observe that NFSTs greatly outperform Seq2seq models on both tasks (with $p < 0.01$ on every metric). We also observe that while we get some statistically insignificant ($p > 0.05$) improvement when we used the seq2seq models to rerank other seq2seq models' outputs, the magnitude is not nearly as large as the improvement we get from switching to NFSTs.

Dakshina. We evaluate on romanization for both Urdu and Sindhi. We observe that for both languages, domain-agnostic NFSTs consistently outperform Seq2seq models on both MRR and EM (again with $p < 0.01$).

	Valid		Test	
	MRR	EM	MRR	EM
NFST	0.439	0.270	0.499	0.349
Seq2seq	0.378	0.243	0.415	0.262

Table 7.4: DAKSHINA transliteration results — Urdu.

	Valid		Test	
	MRR	EM	MRR	EM
NFST	0.463	0.297	0.453	0.299
Seq2seq	0.419	0.276	0.400	0.259

Table 7.5: DAKSHINA transliteration results — Sindhi.

7.5.3.4 Ablation studies

We also do the following ablation studies on the CMUDICT dataset:

Removal of recurrent connection. Under this configuration, we remove the recurrent LSTM component in the design of G. Therefore G is effectively a bigram model: $G(\boldsymbol{\omega}) = \prod_t p(\omega_t \mid \omega_{t-1})$. We would like to know how much performance do we lose with this simplification.

The results are listed in Tables 7.6 and 7.7. We can see that the performance drops significantly with the removal of the LSTM component.

Looser $\mathcal{L}'_K(\theta)$ upper bound. Recall that by Proposition 7.4.2 we can improve upon the variational upper bound $\mathcal{L}'$ of the true loss function $\mathcal{L}$ (equation (7.14)) using importance reweighting: $\mathcal{L}'_K$ (equation (7.17)); and that with $K \longrightarrow \infty$, $\mathcal{L}'_K \longrightarrow L$. We would like to know

	Valid		Test	
	MRR	EM	MRR	EM
NFST — no recurrent connection	0.156	0.035	0.156	0.034
NFST	0.674	0.533	0.647	0.49

Table 7.6: CMUDICT: no recurrent connections (G2P).

	Valid		Test	
	MRR	EM	MRR	EM
NFST — no recurrent connection	0.149	0.030	0.148	0.026
NFST	0.531	0.367	0.535	0.364

Table 7.7: CMUDICT: no recurrent connections (P2G).

how performace reacts when we vary K.

We experimented with $K \in \{1, 2, 4, 8\}$, with $K = 8$ being the default hyperparameter (*i.e.*, all experiments run outside this ablation study section have $K = 8$.)

Results from this ablation study are in Tables 7.8 and 7.9. They do not seem to suggest that optimizing for the tighter IWAE bound in equation (7.17) translates to better performance (than optimizing for the standard variational bound equation (7.14)). While we do achieve the best accuracy with $K = 8$ on the P2G task (Table 7.9), the trend is the opposite in Table 7.8.

	Valid		Test	
	MRR	EM	MRR	EM
NFST: $K = 1$	0.678	0.538	0.657	0.503
NFST: $K = 2$	0.672	0.525	0.650	0.496
NFST: $K = 4$	0.667	0.522	0.650	0.500
NFST: $K = 8$	0.674	0.533	0.647	0.49

Table 7.8: CMUDICT: varying K (G2P).

	Valid		Test	
	MRR	EM	MRR	EM
NFST: $K = 1$	0.519	0.360	0.524	0.347
NFST: $K = 2$	0.528	0.363	0.531	0.361
NFST: $K = 4$	0.502	0.328	0.517	0.342
NFST: $K = 8$	0.531	0.367	0.535	0.364

Table 7.9: CMUDICT: varying K (P2G).

7.5.4 Effects of knowledge-loaded topologies

We focus on the CMUDICT dataset to test the effects of knowledge-loaded topologies. In particular, we test whether finite-state topologies that encode our phonological knowledge about English have any quantitative or qualitative effects on NFSTs we learn. To this end, we design an MRE SYLLABLES (MRE 7.5.15) that seeks to model the English syllable structure in a phoneme string: SYLLABLES allows optional onset and coda units; but a nucleus is mandatory. We also allow an onset (MRE 7.5.16) and a coda (MRE 7.5.17) to consist of multiple consonant phonemes, while a nucleus (MRE 7.5.18) is a single vowel or consonant phoneme. As for STRESS, they appear at the start of a syllable, and may consist of a primary stress marking (‘ˈ’), a secondary stress marking (‘ˌ’), or no stress marking at all. Finally, we classify phonemes in our CMUDICT dataset into CONSONANTS (MFST 7.5.8) and VOWELS, and prepend them with special marks to signify their classifications respectively.

SYLLABLES is an MFST that transduces a phoneme sequence $\mathbf{y}$ into itself if it conforms to our syllable constraints, and does not accept $\mathbf{y}$ otherwise.[32] SYLLABLES is then com-

[32] In other words, SYLLABLES is a marked finite-state *acceptor*. As we noted in footnote 8, we opt to write MREs as tailored towards to MFSTs in this work, which unfortunately adds clutter to the notation of SYL-

	Valid		Test	
	MRR	EM	MRR	EM
NFST w/ SyllabicTransduction	0.617	0.471	0.603	0.447
NFST w/ AgnosticTransduction	0.674	0.533	0.647	0.49
Seq2seq	0.496	0.404	0.472	0.382

Table 7.10: SyllabicTransduction G2P results.

	Valid		Test	
	MRR	EM	MRR	EM
NFST w/ SyllabicTransduction	0.475	0.314	0.490	0.330
NFST w/ AgnosticTransduction	0.531	0.367	0.535	0.364
Seq2seq	0.306	0.224	0.303	0.208

Table 7.11: SyllabicTransduction P2G results.

posed against AgnosticTransduction (MFST 7.5.7) to form SyllabicTransduction (MRE 7.5.14).

MRE 7.5.14 encodes more knowledge of what we know about English phonology than other machines that we experimented with. We neuralize it[33], and conduct experiments following setups in §7.5.3.

The results are in Tables 7.10 and 7.11. The differences between SyllabicTransduction and AgnosticTransduction runs are statistically significant, as well as the differences between Seq2seq and SyllabicTransduction (all with $p < 0.01$). We conclude that the domain knowledge we encode in MRE 7.5.14 and its component MREs actually *hurt* empirical performance. While it can be frustrating that added domain knowledge[34] does

lables.

[33]That is, let τ = SyllabicTransduction when we compute the NFST machine weight in equation (7.1).

[34]Of course, we did *not* exhaust the space of English-phonology-knowledge-encoding MFST topologies in our experiments. We leave a more thorough study as future work.

not lead to better accuracy numbers, such negative results are not entirely surprising (Shi et al., 2018a; Post et al., 2019; Sutton, 2019). We also note that SyllabicTransduction still outperforms Seq2seq by a large margin on all metrics. More importantly, such added knowledge *does* have an effect: we will see in §7.5.5 that the induced mark strings under SyllabicTransduction show a clear pattern of phoneme-grapheme correspondence; while AgnosticTransduction does not appear to have learned the notion of phoneme units.

MRE 7.5.14: SyllabicTransduction

Syllables∘AgnosticTransduction

MRE 7.5.15: Syllables

$(\epsilon : \epsilon/\textit{syllable-start}\,\textsc{Stress}.\textsc{Onset}.\textsc{Nucleus}.\textsc{Coda}.\epsilon : \epsilon/\textit{syllable-end})^*$

MRE 7.5.16: Onset

$(\epsilon : \epsilon/\textit{onset}.\textsc{Consonant})^*$

MRE 7.5.17: Coda

$(\epsilon : \epsilon/\textit{coda}.\textsc{Consonant})^*$

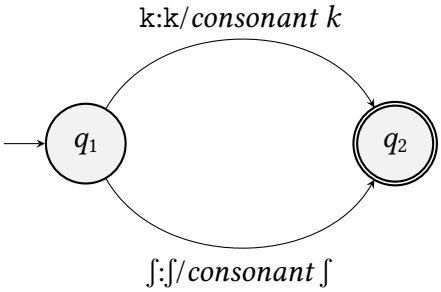
MRE 7.5.18: NUCLEUS

ϵ : ϵ/*nucleus.*(CONSONANT|VOWEL)

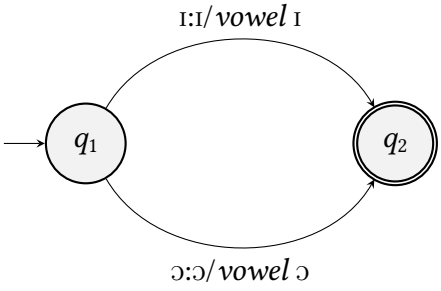
MRE 7.5.19: STRESS

(ϵ : ϵ/ϵ|ˈ : ˈ/*primary-stress*|ˌ : ˌ/*secondary-stress*)

MFST 7.5.8: CONSONANT (many arcs omitted)



MFST 7.5.9: VOWEL (many arcs omitted)



7.5.5 Interpretability

Empirical results in §7.5.3.3 show that NFSTs excel at assigning probability mass to input-output string pairs. As we have said in §7.2.4.1, for feature neuralized NFSTs, the machine weight of string pair $(\mathbf{x}, \mathbf{y})$ is the sum of weights of mark strings in the regular language $(\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega \subseteq \tau_\Omega$. We would like to know how an NFST (τ, G) assigns weight among mark strings within $(\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega$. We have two questions:

Emergence of patterns? There are many ways of transducing between $\mathbf{x}$ and $\mathbf{y}$ under a permissive topology (*e.g.*, AGNOSTICTRANSDUCTION as defined in MFST 7.5.7). Do all mark strings $\in (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega$ contribute equally to the weight sum $\sum_{\omega \in (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega} G_\theta(\omega)$? Or do some of these strings dominate the weight sum? If so, is there a pattern of these dominant mark strings?

Effects of inductive bias? Our attempt at imbuing τ with what we know about syllable structures did not bring about increased empirical performance in §7.5.4. However, we wonder if these topologies still encourage mark strings that conform to known (English) phonological patterns to dominate the mark string weight sum.

To answer these two questions, we look at samples from posterior distributions over mark strings, conditioning on input-output string pairs $(\mathbf{x}, \mathbf{y})$ — namely $p(\omega \mid (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega)$ (equation (7.20)). We randomly pick 5 string pairs from the validation set (that is, corresponding phoneme-grapheme string pairs), and approximately sample from the posterior distributions

$p(\boldsymbol{\omega} \mid (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega)$ following the methodology described in §7.5.2.1, but with dampened proposal distributions (with temperature = 3.) to encourage sample diverage, and with a much larger particle size = 512 for more courage. Most probable paths, along with their estimated normalized probabilities, are listed in Table 7.12 (where τ = AgnosticTransduction) and Table 7.13 (where τ = SyllabicTransduction). In Table 7.13 we do not list the original mark strings: those are interleaving strings of mark strings from two different machines, and would be hard to interpret. We separate subsequences of these mark strings into two parts: those originate from AgnosticTransduction (MFST 7.5.7) and those from Syllabic (MRE 7.5.14), and list them separately.

One can immediately see from these two tables that our approximated posterior distributions $\hat{P}(\boldsymbol{\omega} \mid (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega)$ are very peaked: all string transductions have most of their masses put on a single mark string. In Table 7.12 we find that the mark strings are interleavings of counterpart phonemes and graphemes. But the interleaving order is not strict — for example, in the *auntie* transduction, the most probable mark string went from phoneme-grapheme to grapheme-phoneme, then back to phoneme-grapheme. There does not appear to be a consistent monotonic alignment scheme between phonemes and graphemes across all words, either. In fact, it is not clear whether the model operates on the notion of grapheme-phoneme alignment — for example, in the *Atchison* transduction the model thinks the most likely transduction splits up a single phoneme tʃ into disjoint segments, even while knowledge of output alphabet is provided (footnote 28). An interpretation is that the ordering of marks in high-probability mark strings maps to a highly probable generative

Phoneme / grapheme strings	Mark strings	$\hat{P}(\boldsymbol{\omega} \mid (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega)$
/ˈɔnti/ auntie	ˈauɔnnttiie	95.7%
	ˈaɔnunttiie	4.1%
	ˈɔaunnttiie	.2%
/ˈdʌbli/ doubly	ˈddouʌbblliy	96.1%
	ˈddoʌubblliy	3.2%
	ˈddʌoubblliy	.6%
/ˈætʃɪsən/ Atchison	ˈaættʃchiɪssoənn	99.0%
	ˈaættʃchiɪssəonn	.6%
	ˈaættʃchiɪssoənn	.3%
/juːˈmænɪtiːz/ humanities	hjuuːˈmmaænniɪttiiːezs	80.9%
	jhuuːˈmmaænniɪttiiːezs	18.9%
/ˈnɛmənˌtoʊdz/ nematodes	ˈnneɛmmaəˌttooʊddezs	99.9%

Table 7.12: Most probable mark strings from $(\mathbf{x} \circ \textsc{AgnosticTransduction} \circ \mathbf{y})_\Omega$ under our approximated approximate posterior distribution. Only mark strings with estimated probability > 0.1% are shown.

story of a string pair $(\mathbf{x}, \mathbf{y})$ under the distribution $G_{\boldsymbol{\theta}}$: while $G_{\boldsymbol{\theta}}$ evidently favors strings where corresponding phoneme and grapheme clusters alternate, there is no additional incentive for aligning atomic phonemes.

On the other hand, we see in Table 7.13 that the distributions are even more peaked. Moreover, the AgnosticTransduction subsequences of the most probable mark strings follow a strict $(\textit{phoneme.grapheme})^*$ pattern. We speculate that sequences of phoneme units are required to model syllable structures. Even though the model hasn't managed to model English syllables perfectly, it still encourage the alignment against corresponding graphemes.

Phoneme / grapheme strings	Mark strings under AGNOSTICTRANSDUCTION	Mark strings under SYLLABIC	$\hat{P}(\boldsymbol{\omega} \mid (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega)$
/ˈɔnti/ auntie	ˈɔaunnttiie	syllable-start primary-stress nucleus vowel ɔ coda consonant n coda consonant t syllable-end syllable-start syllable-no-stress nucleus vowel i syllable-end	100.0%
/ˈdʌbli/ doubly	ˈddʌoubblliy	syllable-start primary-stress onset consonant d nucleus vowel ʌ coda consonant b coda consonant l syllable-end syllable-start syllable-no-stress nucleus vowel i syllable-end	99.7%
	ˈddoʌubblliy	syllable-start primary-stress onset consonant d nucleus vowel ʌ coda consonant b coda consonant l syllable-end syllable-start syllable-no-stress nucleus vowel i syllable-end	.3%

Phoneme / grapheme strings	Mark strings under AGNOSTICTRANSDUCTION	Mark strings under SYLLABIC	$\hat{P}(\boldsymbol{\omega} \mid (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega)$
/ˈætʃɪsən/ Atchison	ˈæatʃtchɪissəonn	syllable-start primary-stress nucleus vowel æ coda consonant tʃ syllable-end syllable-start syllable-no-stress nucleus vowel ɪ coda consonant s syllable-end syllable-start syllable-no-stress nucleus vowel ə coda consonant n syllable-end	99.8%
/juːˈmænɪtiːz/ humanities	jhuuːˈmmaænniɪttiiːezs	syllable-start syllable-no-stress onset consonant j nucleus vowel u primary-stress syllable-end syllable-start primary-stress onset consonant m nucleus vowel æ coda consonant n syllable-end syllable-start syllable-no-stress nucleus vowel ɪ coda consonant t syllable-end syllable-start syllable-no-stress nucleus vowel i ː coda consonant z syllable-end	99.9%

Phoneme / grapheme strings	Mark strings under AGNOSTICTRANSDUCTION	Mark strings under SYLLABIC	$\hat{P}(\boldsymbol{\omega} \mid (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega)$
/ˈnɛmənˌtoʊdz/ nematodes	ˈnneɛemmməaˌttoʊoddzes	syllable-start primary-stress syllable-onset coda n syllable-nucleus vowel ɛ syllable-coda coda m syllable-end syllable-start syllable-no-stress syllable-nucleus vowel ə syllable-end syllable-start secondary-stress syllable-onset coda t syllable-nucleus vowel o ʊ syllable-coda coda d syllable-coda coda z syllable-end	98.2%
	ˈnneɛemmməaˌttoʊoddezs	syllable-start primary-stress syllable-onset coda n syllable-nucleus vowel ɛ syllable-coda coda m syllable-end syllable-start syllable-no-stress syllable-nucleus vowel ə syllable-end syllable-start secondary-stress syllable-onset coda t syllable-nucleus vowel o ʊ syllable-coda coda d syllable-coda coda z syllable-end	1.8%

Phoneme / grapheme strings	Mark strings under AGNOSTICTRANSDUCTION	Mark strings under SYLLABIC	$\hat{P}(\boldsymbol{\omega} \mid (\mathbf{x} \circ \tau \circ \mathbf{y})_\Omega)$

Table 7.13: Most probable mark strings from $(\mathbf{x} \circ \text{SYLLABICTRANSDUCTION} \circ \mathbf{y})_\Omega$ under our approximated approximate posterior distribution. Only mark strings with estimated probability > 0.1% are shown.

7.6 Related work

In this chapter we have introduced string-set semirings (Definition 7.2.3). String-set semirings are **structured semirings**, whose $\otimes$ and $\oplus$ operations can be leveraged to mechanically derive quantities of larger finite-state automata (in our case, the mark projection of an MFST). Other well-known structured semirings include expectation semirings (Eisner, 2001; Li and Eisner, 2009) and lexicographic semirings (Roark, Sproat, and Shafran, 2011; Sproat et al., 2014). Separately, the idea of featurizing finite-state transductions with local features was also present in Wu et al. (2014).

There has been work relating finite-state methods and neural architectures. For example, Schwartz, Thomson, and Smith (2018) and Peng et al. (2018) have shown the equivalence between some neural models and WFSAs. The most important differences of our work is that in addition to classifying strings, NFSTs can also transduce strings. Moreover, NFSTs also allow free topology of FST design, and breaks the Markovian assumption.

We note that Aharoni and Goldberg (2017) and Deng et al. (2018) are also similar to our work; in that they also marginalize over latent alignments, although they do not enforce

the monotonicity constraint. Work that discusses globally normalized sequence models are relevant to our work. In this paper, we discuss a training strategy that bounds the partition function; other ways to train a globally normalized model (not necessarily probabilistic) include Wiseman and Rush (2016) and Andor et al. (2016). On the other hand, our NFSTs bear resemblance to RNNGs (Dyer et al., 2016), which was also autoregressive, and also employed importance sampling for training.

7.7 Conclusion and limitations

We have introduced neural finite-state transducers. We have discussed two alternative definitions of their machine weights, and their theoretical expressiveness: we emphasized that when we couple NFSTs with autoregressive mark string scoring functions (§2.2.4) G_{θ} (*i.e.*, $\Theta \in$ ELNCP), they form a very expressive model family (§7.3). We also stressed that we did not give up on grammar engineerability (*i.e.*, we can freely design the finite-state grammar of relations that we wish to recognize, or not recognize, for NFSTs), compared to neural autoregressive sequence models.

Empirically, we have shown that NFSTs compete favorably with state-of-the-art neural sequence models on transduction tasks. At the same time, they also can also offer interpretable mark strings, when the right inductive bias is engineered in their MFST components.

However, as we discussed in §7.2.4.1, feature neuralization gives up the path weight

property. Therefore, feature neuralized NFSTs do not form an algebraic structure under the MFST regular operations. While their MFST components (§7.2.1) — which decide what relations a grammar should recognize — are still closed under rational algebraic operations (§7.2.3) and composition,[35] the resultant NFSTs are not. In other words, while a composite grammar's *support* (*i.e.*, which $(\mathbf{x}, \mathbf{y})$ pairs will have weight 0) can be engineered/changed by familiar operations à la WFSTs (*i.e.*, concatenation, union, composition, and closure), we cannot analogously engineer the composite grammar's *weights* this way. The lack of compositionality makes it impossible to reuse independently trained modules in different tasks, *e.g.*, using a separately trained n-gram language model for speech recognition (Mohri, Pereira, and Riley, 2008).

The decision to limit ourselves to $G_\theta \in$ ELNCP in this chapter can also cause an (unnecessary) tension between model expressiveness and interpretability. Recall that any marked finite-state machine τ will have a regular mark projection τ_Ω, which can be captured by an ELN (and therefore ELNCP) weighted language. However, some applications, when modeled using the topology of τ, may define a mark string weighted language $p \in$ EC, but support$(p) \subsetneq \tau_\Omega$. By conclusions of § 3 we know p may not be in ELN, or even ELNCP. While by Proposition 7.3.2 we know we can have any $L \in$ NP as our output string distribution support, it required an topology that inserts 'dummy' latent variables to the start of every mark string. In other words, to make an NFST expressive, a practitioner may have to:

- design the topology of a marked finite-state machine to encode a regular language,

[35] However recall that the while MFSTs are also closed under our defined composition operation, it is not algebraic (§7.2.2.1).

where each mark string is prepended with some 'stochastic noise' symbols. Such design may not meaningfully encode prior knowledge, but caters to parametrization peculiarities of autoregressive models. Furthermore, mark strings from such regular language may contain more symbols than needed, which exacerbates the computational burden. Alternatively, a practitioner may instead:

- parametrize $G_{\boldsymbol{\theta}}$ not as an autoregressive model, to escape the 'wrong latent variable order' problem. Of course, the mark string scoring function in an NFST does not have to be $\in$ ELNCP, let alone ELN, as long as parameter estimation is not required (*e.g.*, when the user can manually specify $\boldsymbol{\theta}$ for $\boldsymbol{\theta} \in \boldsymbol{\Theta}, \boldsymbol{\Theta} \subseteq$ EC). But in practice we usually need to learn $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ from data, subjecting ourselves under the uncomputability results of § 5.

We will revisit these limitations in § 8.

Chapter 8

Neuralization of regular expressions

8.1 Introduction

In this chapter, we introduce **neural regular expressions** (NREs) as an extension to NFSTs.

8.1.1 Why do we need to extend NFSTs?

As we said in §7.7, we introduce NREs to address the two distinct but related problems of NFSTs. We review these problems below:

It is not clear how we can combine NFSTs together the way we do WFSTs. While NFSTs are a strict generalization of WFSTs, there does not seem to be a straightforward generalization of the familiar regular and composition operations on NFSTs, which preserve

the WFST machine weight semantics. For example, the union operator '|': while the expression $M_1|M_2$ is well-defined when M_1 and M_2 are both FSTs, it is not clear from our NFST definitions (§7.2) what the expression $N[\tau_1, G_1]|N[\tau_2, G_2]$ means — and if it is indeed a valid expression, should the resultant be an NFST.

One reason why there does not appear to be a straightforward generalization of the WFST operations is that an NFST has a scoring function component, in addition to a finite-state one (§7.2). While the finite-state component of an NFST (*i.e.*, MFST) does have a well-defined set of regular operations (*i.e.*, MREs introduced in §7.2.2), the weighted language component (*i.e.*, the mark string scoring function) does not.

Without the WFST machine weight semantics, we would not be able to combine NFSTs in the manner that people combined WFSTs. Consider the slot filling example of §1.2.1, where gazetteers of product names (*e.g.* RE 1.2.2) can be separately engineered, and incorporated into a bigger composite machine later on. Losing the ability to form NLP pipelines this way is certainly undesirable.

Monotonic string transduction annotations may require non-autoregressive mark string scoring functions. Recall that the finite-state machine component of an NFST — namely an MFST — is a compact representation of string transductions, which also annotates each transduction path with a mark string. Since these annotations are *monotonically aligned* to the string transductions they annotate, there may be no good autoregressive model (*e.g.*, ELNCP) that can capture them, despite the fact that the NFST itself is a very powerful formalism. As we noted in §3.4.2, autoregressive latent variable sequence models

are sensitive to the ordering of latent variables in a string. And as a result, the monotonically aligned annotations, when seen as latent variables, may not appear at optimal positions, such that autoregressive mark string scoring functions can capture them perfectly. We will show a concrete example in §8.4. While it is certainly possible to design specialized mark string scoring functions that work with monotonically aligned annotations, those will require specialized parameter estimation procedures. However, as we have argued in §7.3, general non-autoregressive families of mark string scoring functions will result in uncomputable NFST machine weights, which in turn make training difficult.

To summarize, the NFSTs we introduced in § 7 cannot be combined together easily as WFSTs. And the monotonic alignment of mark strings forces a practitioner to either give up expressiveness, or design their own mark string scoring function family.

8.1.2 What are NREs?

Neural regular expressions are an algebraic structure with **neural rational operations** (NROs) over **partially neuralized finite-state transducers**[1] (pNFSTs). pNFSTs (which we formally define in §8.2) generalize both marked and neuralized finite-state transducers (§ 7), by making both mark strings and their scoring functions compositional. A major difference between pNFSTs and NFSTs is that mark strings under pNFSTs encode *parses*. To score a mark string $\boldsymbol{\omega}$ under pNFST M, $\boldsymbol{\omega}$ is first parsed into a parse (which we denote as $t = \downarrow_M(\boldsymbol{\omega})$), which contains instructions for scoring certain subsequences of $\boldsymbol{\omega}$, which

[1] We call them *partially* neuralized because a pNFST can have parts (which themselves are pNFSTs) that are neuralized (*i.e.*, associated with a non-trivial scoring function), while other parts aren't.

are subsequently aggregated with M's own scoring of $\boldsymbol{\omega}$, and returned as the weight of $\boldsymbol{\omega}$ under M (which we denote as $[\![t]\!]_M$).

Therefore, under pNFSTs, a mark string contains *out-of-band* information that explicitly instructs how the mark string itself should be parsed; and also how such a parse should be scored. More specifically, its subtrees will be scored by composite functions, which share intermediate computation results while scoring these different subsequences, to help capture structures and long-term dependencies. Crucially, a pNFST evaluates weights of subsequences recursively, following the structure of the parse: this allows subsequences of a mark string to be scored *not* in a left-to-right order. Effectively, this allows a *lookahead* when it comes to scoring the prefix, which is impossible under NFSTs (that have ELNCP $G_{\boldsymbol{\theta}}$'s) — pNFSTs can weight a mark string's left context based on its right context, without the need of dummy latent variables as a mark string prefix.

By capitalizing the expressiveness of pNFSTs, NROs generalize familiar rational operations on FSTs — namely union, concatenation, closure — as well as the relational operation composition. As a concrete example, the NRO composition of two pNFSTs $M_1 = (\tau_1, G_1)$ and $M_2 = (\tau_2, G_2)$ results in a pNFST $M = M_1 \circ M_2$, under which a mark string $\boldsymbol{\omega}$ has weight $G_1(\boldsymbol{\omega}_1) \cdot G_2(\boldsymbol{\omega}_2)$, where $\boldsymbol{\omega}_1$ and $\boldsymbol{\omega}_2$ are subsequences of $\boldsymbol{\omega}$ that come through τ_1 and τ_2 respectively — just like the WFST composition. We implement these NROs through the use of 'out-of-band' mark symbols in a mark string, which are used to construct a parse tree that how different subsequences should be differently scored.

8.1.3 Chapter outline

This chapter is structured as follows: we formally describe our partially neuralized finite-state transducer extension in §8.2. We also describe how counterparts to familiar WFST operations can be implemented as neural regular expressions in §8.3. In the rest of this chapter, we show how two limitations of NFSTs outlined in §7.7 can be addressed by NREs: in §8.4 we describe an example transduction task, which no NFSTs with a 'natural' MFST topology can capture with autoregressive scoring functions, but can be handled by NREs with the same MFST topology. Finally, in §8.5 we demonstrate how separately trained NFSTs can be combined together into a larger NRE, to solve a cold-start slot-filling problem.

8.2 Partially neuralized finite-state machines

In this section, we introduce partially neuralized finite-state machines (pNFSTs). In §8.2.1 we formally define them. In §8.2.2 we discuss some concerns that arise in the context of autoregressive sequence model families. Finally, in §8.2.3 we list two example pNFSTs.

8.2.1 Definition

Here we describe the formal definitions of pNFSTs. First we define **parses**:

Definition 8.2.1. *$C \subsetneq \Omega$ is a finite non-literal alphabet.*

Definition 8.2.2. *A parse $t \in \mathcal{T}$ is a tree associated with mark alphabet Ω and non-literal alphabet C, where*

- *terminal nodes are strings $\in \Omega^*$, which we also call **literals**.*
- *non-terminal nodes are single symbols $\in C$, which we also call **non-literals**.*

Formally, we define a partially neuralized finite-state machine to be a 3-tuple $M = (T_M, \downarrow_M, [\![]\!]_M)$:

Definition 8.2.3.

- *T_M is the marked finite-state transducer associated with M. And*
- *parse function $\downarrow_M : \Omega^* \longrightarrow \mathcal{T}$ that produces a parse from mark string $\boldsymbol{\omega} \in \Omega^*$. And*
- *parse scoring function $[\![]\!]_M : \mathcal{T} \longrightarrow \mathbb{R}_{\geq 0}$ scores a parse tree.*

Machine weights of pNFSTs. The two interpretations of NFST machine weights (§7.2.4.1) still apply for pNFSTs. Under the feature neuralization interpretation, we define the machine weight of pNFST M to be

$$[\![M]\!] \triangleq \sum_{\boldsymbol{\omega} \in T_M} [\![\downarrow_M (\boldsymbol{\omega})]\!]_M. \tag{8.1}$$

And

$$[\![M]\!](\mathbf{x}, \mathbf{y}) \triangleq \sum_{\boldsymbol{\omega} \in \mathbf{x} \circ T_M \circ \mathbf{y}} [\![\downarrow_M (\boldsymbol{\omega})]\!]_M. \tag{8.2}$$

On the other hand, under the literal neuralization interpretation, we define

$$[\![M]\!] \triangleq \sum_{\boldsymbol{\omega} \in T_M} [\![\downarrow_M (\boldsymbol{\omega})]\!]_M f(\boldsymbol{\omega}). \tag{8.3}$$

And

$$[\![M]\!](\mathbf{x}, \mathbf{y}) \triangleq \sum_{\boldsymbol{\omega} \in \mathbf{x} \circ T_M \circ \mathbf{y}} [\![\downarrow_M (\boldsymbol{\omega})]\!]_M f(\boldsymbol{\omega}). \tag{8.4}$$

where $f(\boldsymbol{\omega})$ denotes the number of paths in M that have a mark projection of $\boldsymbol{\omega}$. In this chapter we do not explicitly denote which interpretation we are using, since they coincide for all finite-state machines we manipulate.

To repeat, the main difference between NFSTs and pNFSTs is that a pNFST scores a parse[2] that is built from mark strings. In §8.3 we will see how familiar rational operations can be implemented using this new abstraction. Since MFSTs (§7.2.1) and NFSTs can both be seen as pNFSTs that have simple parses and parse scoring functions, type conversion is also straightforward. Below we introduce two possible constructions. Both of these constructions introduce a new non-literal $\omega_M \in C$ to help identify these newly built pNFSTs M in further processing.

- **Marked finite-state machines**: given marked FST τ, we build $M = (T_M, \downarrow_M, [\![]\!]_M)$ to be an equivalent pNFST $(T_M, \downarrow_M, [\![]\!]_M)$ where

[2] More accurately, pNFSTs score *traversals of parses* that is *structured*. In this chapter we do not always make an explicit distinction between the two, since we work with parse traversals that unambiguously identify with their parses (among the training and test instances).

  - the non-literal $\omega_M \in C$ is a unique identifier of M,
  - $T_M = (\varepsilon : \varepsilon / \omega_M)\tau$,
  - $\downarrow_M$ is the parse building function of M, where $\downarrow_M(\boldsymbol{\omega})$ has a root non-literal node $\omega_M \in C$, which has a single literal child $\boldsymbol{\omega} \in \Omega^*$; and
  - $[\![]\!]_M$ is the parse scoring function of M, where $[\![t]\!]_M = 1, \forall t \in \mathcal{T}$.

- **Neural finite-state machines**: an NFST $N[\tau, G]$ has an equivalent pNFST $M = (T_M, \downarrow_M, [\![]\!]_M)$ where
  - $\omega_M \in C$ is a unique identifier of (τ, G),
  - $T_M = (\varepsilon : \varepsilon / \omega_M)\tau$,
  - $\downarrow_M$ is the parse building function of M, where $\downarrow_M(\boldsymbol{\omega})$ is a tree rooted in non-literal ω_M, which has a single child $\boldsymbol{\omega}_{1:}$,[3] and
  - $[\![]\!]_M$ is the parse scoring function of M, where $[\![t]\!]_M = G(\boldsymbol{\omega}')$, and $\boldsymbol{\omega}'$ is the single child literal of parse t.

Partial neuralization operator. We described one way of converting an NFST to a pNFST above, by seeing mark strings as (flat) parses. In this chapter, we further generalize the feature neuralization operator (§7.2.4.1) to operate on pNFSTs. More formally, we define a new **partial neuralization**[4] operator:

[3] In this thesis, we use the notation $\boldsymbol{\omega}_{1:}$ to denote the suffix of $\boldsymbol{\omega} = \omega_0\omega_1 ...: \omega_1\omega_2 ...$

[4] Note that while feature and literal neuralization operators are technically machine weight functions of NFSTs, the newly introduced $N_{\text{partial}}[]$ are truly operators that build new pNFSTs. The machine weights of pNFSTs are separately defined, as the $[\![]\!]_M$ operator we described above.

Definition 8.2.4. *The partial neuralization operator* $N_{\text{partial}}[]$ *takes two arguments* (M, G)*, where* M *is a pNFST, and* $G : \mathcal{T} \longrightarrow \mathbb{R}_{\geq 0}$*. We define* $M' = N_{\text{partial}}[M, G] = (T_{M'}, \downarrow_{M'}, [\![]\!]_{M'})$ *to be the following NFST:*

- $\omega_{M'} \in C$ *is a unique identifier of* (M, G)*,*
- $T_{M'} = (\varepsilon : \varepsilon / \omega_{M'}) T_M$*,*
- *the parse building function* $\downarrow_{M'}$*, where* $\downarrow_{M'}(\boldsymbol{\omega})$ *is a tree rooted in the unique identifier* $\omega_{M'}$*, which has a single child* $\downarrow_M(\boldsymbol{\omega})$*, and*
- *the parse scoring function* $[\![]\!]_{M'}$*, where* $[\![t]\!]_{M'} = [\![t']\!]_M \cdot G(t)$*, and* t' *being the subtree that is the child of the root non-literal* $\omega_{M'}$*.*

In this chapter, we use the shorthand notation $N_{\text{partial}}[M, G] = N[M, G]$. An algorithm that builds neuralized pNFSTs is listed in Algorithm 3. Note that while feature and literal neuralization operators map NFSTs to real machine weights, the parse neuralization operator takes a pNFST and a scoring function as input, and produces a new pNFST as output.

8.2.2 Traversal scoring under partial neuralization

We may see the pNFST formalism as a parametric family of NFSTs that define mark string scoring functions in specific ways: instead of scoring mark strings $\boldsymbol{\omega}$ under a single opaque mark string scoring function G, mark string weights under pNFSTs are $\prod_t G_t(t)$, namely

Algorithm 3: An algorithm for building the pNFST $N[M, G]$.

Input:

- M is a pNFST
- $(G : \mathcal{T} \longrightarrow \mathbb{R})$ is a parse scoring function

Output: A pNFST $M' = N[M, G]$

$\omega_{M'} \longleftarrow$ some unique identifier of (M, G) ;
Assert $\omega_{M'} \in C$;
$T_{M'} \longleftarrow (\varepsilon : \varepsilon / \omega_G) T_M$;
return $(T_{M'}, \downarrow_{M'}, [\![]\!]_{M'})$
Function $\downarrow_{M'}(\boldsymbol{\omega})$**:**
    Assert the first symbol of $\boldsymbol{\omega}$: $\omega_0 = \omega_{M'}$;
    $t \longleftarrow$ a tree with root $\omega_{M'}$; and the subtree $\downarrow_M (\boldsymbol{\omega}_{1:})$ as its child ;
    return t ;
Function $[\![]\!]_{M'}(t)$**:**
    Assert the root of t is a non-literal $\omega_{M'}$;
    Assert the root of t has a single subtree t' ;
    return $[\![t']\!]_M \cdot G(t)$;

products of scores of $\boldsymbol{\omega}$'s subsequence parses t, under some parse scoring function G_t. Now a question is: how do we parametrize these G_t's? One way is to parametrize them as functions of parse *traversals*, which can be regarded as strings. For example, we can traverse a parse in a depth-first left-to-right order. Such traversal strings may contain symbols from both Ω and C. Therefore, G_t can be regarded a weighted language, with a support a subset of $(\Omega \cup C)^*$. In this chapter, we parametrize parse scoring functions If we further specify them as ELNCP mark string scoring functions, we can recover the behavior of NFSTs introduced in § 7. Below we discuss some issues concerning traversal order and the parse scoring functions that score them:

Multiple neuralization and inter-pNFST dependency. We can recursively neuralize a pNFST multiple times. Furthermore, as we will see in §8.3, it is also possible to combine multiple pNFSTs into a larger pNFST. How should we parametrize the parse scoring function of a composite pNFST? If every pNFST component in the composite pNFST makes use of traversal-scoring in their parse scoring functions, we should arrange the traversals in a way that honors the definition of parse scoring functions in Definition 8.2.4: specifically, the parse score of inner pNFSTs should not depend on the outer context that falls outside of the neuralize scope (since the autoregressive model has not seen outer context yet). On the other hand, the score of a parse that encompass these subtrees can legitimately be functions of these subtrees (and their traversals). Speaking loosely, an autoregressive model of parse traversals should generate traversals among neuralized components first, followed by unneuralized ones.

Shared computation. Under the pNFST formalism introduced in this chapter, each neuralization operation gets its own scoring function argument: for example, the pNFST $N[N[M, G_1], G_2]$ undergoes two neuralizations, and two different functions G_1 and G_2 (which in this chapter are parameterized as sequence models of parse traversals) are invoked. However, in practice we may want to share computation between different scoring functions that operate on overlapping components in a parse. This can usually be done by sharing some neural network layers between scoring functions. For example, G_2 in the example above may make use of layer embeddings of G_1. Such kind of computation sharing

is supported in many modern computation graph packages, such as PyTorch.

Probability interpretation of parse traversals. We previously noted that the machine weight of a vanilla NFST can be interpreted as the probability that a randomly sampled mark string is accepted by the mark projection of the NFST's mark projection (§7.2.4.3). In a similar spirit, we would like to (hopefully) interpret a pNFST's machine weight (*e.g.*, equations (8.1) and (8.2)) as the probability of some parse traversal, distributed according to some (likely parametric and trained) sequence distributions.

However, the pNFSTs we describe in this chapter are formed under assumptions that are quite different from that of NFSTs in § 7: in § 7 we generally assumed that all NFSTs are induced using the very same mark string scoring function (which itself may be parametric and trained), differing only in their topologies (*i.e.*, MFST components). However, in this chapter, a pNFST may be formed by combining other pNFSTs. There is generally no guarantee that these pNFSTs will have the same parse scoring function, or the same parse function. As a result, there is no guarantee that the same mark string from two different pNFSTs will result in the same parse. Nor is there a guarantee that under two different pNFSTs, the same parse will have identical scores.

There is another problem stemming from transforming parses (and hence mark strings) into traversals, before scoring: if infinitely many different parses from the same pNFST can have the same parse traversal, then the pNFST will have a divergent machine weight.

To circumvent those problems, we only work with pNFSTs that additionally satisfy the two following properties:

- Different mark strings always result in different parse traversals.[5]
- The same mark string always results in the same parse, regardless of pNFST.
- The same parse always results in the same score, regardless of pNFST.

In other words, we assume that we can induce 'universal' parse function and parse scoring function per the task we train and evaluate on, which allow us to train and decode from pNFSTs the way we did NFSTs. This way, given an pNFST, we can still regard its machine weight to be the probability that a string drawn from the distribution defined by the induced parse scoring function is 'accepted' by the pNFST, in the sense that the string matches one of the traversal strings under the pNFST.

8.2.2.1 Two variants of traversal-scoring parse scoring functions

As we said earlier, in this chapter, we parametrize G in the partial neuralization operation to score parse traversals. We need all parses' traversal orders to be unambiguous, for G to be a function. To this end, we define a traversal priority function of both literals and non-literals:

Definition 8.2.5. $W : (C \cup \Omega^*) \longrightarrow \mathbb{N}$ *maps a non-literal or literal to its* ***traversal priority****.*

[5] Strictly speaking, this property is not required if we can guarantee that pNFST machine weights (that we consider) never diverge. However, if one is to decode input-output pairs from parse traversals, this property is a necessary condition (see a related discussion on recovering input-output string pairs from mark strings in §7.2.4.3).

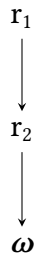


Figure 8.1: Example parse.

In this chapter, when the definition of a traversal priority function is omitted, we arbitrarily define $W(\omega) = 1, \forall \omega \in (C \cup \Omega^*)$. We can now consider the two parse scoring variants. At a high level, they differ in how they treat parse subtrees that correspond to previously neuralized pNFSTs: should we still traverse inside these subtrees (since this is the default traversal behavior), or should we skip these subtrees (since they have already been traversed)?

Double scoring. Under the first option, we first visit each root non-literal introduced by the partial neuralization operator. We visit them in an order that honors both the neuralization dependency (so that the parse of an inner pNFST always gets traversed before an outer one), and the traversal priority function (so that a pNFST parse with a higher priority root non-literal is traversed first). Upon visiting a pNFST parse's root non-literal, we do a preorder depth-first traversal in the parse, visiting siblings in the descending traversal priority order.

We use Figure 8.1 as an example pNFST parse. Assume that both r_1 and r_2 are root non-literals introduced by the partial neuralization operator, which partially neuralizes with G_1 and G_2 respectively. To derive a parse traversal, we will first visit

r_2, since root depends on it. Then, doing traversal in the subtree rooted in r_2, we will visit the literal $\boldsymbol{\omega}$. After r_2 we will visit its parent root, and start another DFS traversal from there. The complete traversal in Figure 8.1 would be $r_2\ \boldsymbol{\omega}\ r_1\ r_2\ \boldsymbol{\omega}$, where the literal $\boldsymbol{\omega}$ will be scored by the inner pNFST's parse scoring function, while first node and the last 3 nodes $r_1\ r_2\ \boldsymbol{\omega}$ will be scored by the outer pNFST's parse scoring function.

Let's now discuss the parse scores of the inner and outer pNFSTs. The inner pNFST is agnostic of its surrounding context. Therefore, its parse scoring function evaluates the literal $\boldsymbol{\omega}$ without any conext: namely $G_2(\boldsymbol{\omega})$. As for G_1, it has to account for all other symbols (nodes) in the traversal. Unlike G_2, G_1 gets to freely look at symbols that are being scored by other parse scoring functions, namely G_2. Since we parametrize both G_1 and G_2 as autoregressive models, we denote context as conditional probability. The overall parse score of Figure 8.1 is thus $G_1(r_2) \cdot G_2(\boldsymbol{\omega}) \cdot G_1(r_1\ r_2\ \boldsymbol{\omega} \mid r_2\ \boldsymbol{\omega})$.

We reiterate that these autoregressive models of parse traversals are *not* generative models of parses. They are generally leaky; they generally assign non-zero probability to invalid parse traversals, not dissimilar to how a mark string scoring function may have support outside the mark projection of an MFST it is coupled with (§ 7).

No double scoring. The no-double-scoring variant differs from the double-scoring variant we just introduced above in that it does not traverse a subtree more than once. For example, the complete traversal in Figure 8.1 under a no-double-scoring variant

would be $\mathrm{r}_2\ \boldsymbol{\omega}$ root, where the first 2 nodes $\mathrm{r}_2\ \boldsymbol{\omega}$ will be scored by the inner pNFST's parse scoring function, while the last node root will be scored by the outer pNFST's parse scoring function.

The parse score of Figure 8.1 under a no-double-scoring variant can be similarly derived as $G_1(\mathrm{r}_2) \cdot G_2(\boldsymbol{\omega}) \cdot G_1(\mathrm{r}_1 \mid \mathrm{r}_2\ \boldsymbol{\omega})$.

In this work, we consider both double-scoring and no-double-scoring variants of G: double-scoring in §8.4, and no-double-scoring in §8.5.

8.2.2.2 Parameter estimation and inference

The weight of mark string $\boldsymbol{\omega}$ of pNFST M, whose parse scoring function scores traversals, takes the form of $[\![\boldsymbol{\omega}]\!]_M = \prod_t G(t) = \prod_i p_\theta(\omega'_i \mid \boldsymbol{\omega}'_{<i})$, where $\boldsymbol{\omega}' \in \Omega^* \cup C$. By parametrizing p_θ as a normalized distribution (*i.e.*, $\sum_{\boldsymbol{\omega}' \in \in (\Omega \cup C)^*} p_\theta(\boldsymbol{\omega}) = 1$, parameters of pNFSTs can be estimated as described in §7.4, following Lemma 7.4.1. Moreover, we continue to do amortized inference with particle smoothing, following §7.5.2.1.

8.2.3 pNFST in action

To illustrate what pNFSTs add to vanilla NFSTs, let's consider the following MRE that transduces `(abc)*` back to themselves as both output and mark strings:

MRE 8.2.20: An MRE

`(a:a/ab:b/bc:c/c)*`

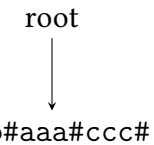


Figure 8.2: $\downarrow_M$ (abcabcabc)

The mark projection of MRE 8.2.20 is the regular language $\{(\texttt{abc})^n \mid n \in \mathbb{N} \cup \{0\}\}$. While it is impossible to design an MRE whose mark projection is non-regular, we will describe how to build a pNFST M upon MRE 8.2.20 whose mark string scoring function scores *non-regular* languages. We define $M = (T_M, \downarrow_M, [\![]\!]_M)$ as follows:

- T_M is defined to be the MFST in MRE 8.2.20,
- $\downarrow_M$ is defined as a function that takes as input string of the form $(\texttt{abc})^n$, $n \in \mathbb{N} \cup \{0\}$, and outputs a parse rooted in the node root that is one level high, and has a singleton literal leaf node of the form $\texttt{b}^n\texttt{\#a}^n\texttt{\#c}^n\texttt{\#}$. For example, on input $\boldsymbol{\omega} = \texttt{abcabcabc}$, $\downarrow_M (\boldsymbol{\omega})$ produces the tree in Figure 8.2.
- Given all our parses admit exactly one traversal order, the design of the traversal priority function (Definition 8.2.5) is not a big concern. We arbitrarily let $W(\boldsymbol{\omega}) = 1, \forall \boldsymbol{\omega} \in C \cup \Omega^*$.
- $[\![]\!]_M$, the parse scoring function, is defined to score parse traversals (as in §8.2.2). Given our simple parses, we arbitrarily design all traversals to be of the form root$\boldsymbol{\omega}$. Finally, we parametrize the traversal scoring function to be ELN.

Under the pNFST M, each mark string from the mark projection of MRE 8.2.20, which is of the form $(\texttt{abc})^n$, will effectively be rearranged to the form $\texttt{b}^n\texttt{a}^n\texttt{c}^n$, preceded by the

root non-literal, then scored. Note that the arranged mark strings now form a *non-regular* language — all the `b` symbols are moved to the beginning, regardless of their original positions (from the machine $T_{M\Omega}$).

Non-regularness through parse structure. We showed above a design of pNFST that effectively make an NFST's mark language non-regular, through $\downarrow_M$ that reorders mark symbols. Parses of M are shallow: they are all one-level deep parses, where the only leaf node of a parse is a shuffled version of the input mark string $\boldsymbol{\omega}$. Below we will also describe an alternative design of pNFST M' whose parses are more complicated. We define $M' = (T_{M'}, \downarrow_{M'}, [\![]\!]_{M'})$ as follows:

- $T_{M'}$ is defined as the MRE in MRE 8.2.21, which is very similarly defined as in MRE 8.2.20, but has additional marks that indicate positions in a parse tree.

  MRE 8.2.21: MRE 8.2.20 with annotations on the structure.

  `((a:a/`*a subtree-a*`(b:b/`*b subtree-b*`))c:c/`*c subtree-c*`)*`

- $\downarrow_{M'}$ is defined as a function that builds up a parse from a mark string $\boldsymbol{\omega}$ in the following way: given an input mark string $\boldsymbol{\omega}$, the resultant parse will be structured as in Figure 8.3, where $\boldsymbol{\omega}_a$, $\boldsymbol{\omega}_b$, and $\boldsymbol{\omega}_c$ are literals, and the rest of nodes are non-literals. Moreover, $\boldsymbol{\omega}_a$ will be a concatenation of mark symbols in $\boldsymbol{\omega}$ that are immediately followed by *subtree-a*, with an additional # suffix, and similarly for $\boldsymbol{\omega}_b$ and $\boldsymbol{\omega}_c$. For example, suppose $\boldsymbol{\omega}$ = `abcabcabc`, $\downarrow_{M'}(\boldsymbol{\omega})$ would be the parse in Figure 8.4.

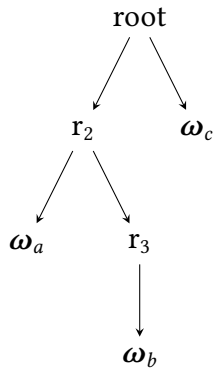


Figure 8.3: Structure of parses produced by $\downarrow_{M'}$.

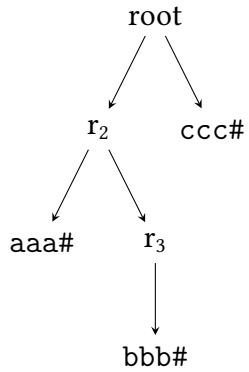


Figure 8.4: $\downarrow_{M'}$ (abcabcabc).

- We define the priority function W in a way that r_3 is traversed before literal strings. Quite arbitrarily, we define:
  - $W(\mathrm{r}_3) = 10$
  - $W(\boldsymbol{\omega}) = 1, \forall \boldsymbol{\omega} \in (C \cup \Omega^*) - \{\mathrm{r}_3\}$.
- $[\![]\!]_{M'}$ first does a preorder depth-first traversal in a given parse t, and returns the traversal's score. Using the parse listed in Figure 8.4 as our example here: a preorder depth-first traversal would yield the following string: root r_2 r_3 `bbb#` `aaa#` `ccc#`.[6] Such strings form a non-regular language (as in the case of M).

Again, both pNFSTs that we have shown above (M and M') enables lookahead for autoregressive mark string scoring functions, by manipulating mark strings from an MFST's mark projection, before scoring them using sequence models. These two pNFSTs differ in how they manipulate the mark strings, though: M directly rearranged the mark symbols on a mark string; while M' achieved the same goal, by building up a parse whose traversal yields the desired mark symbol order.

M' on the other hand took a more roundabout path. The parse building function $\downarrow_{M'}$ did not know to reordering the `b` symbols before `a`'s. But rather, $\downarrow_{M'}$ put mark symbols in their places in a parse. And the symbol reordering is implicitly done during the parse

[6] In general we need more than a preorder traversal to reconstruct a tree. For example, if we also have an inorder traversal we will be able to reconstruct a parse. Given only a preorder traversal we would need more knowledge of the parse's structure to ensure that the parse traversal identifies with a unique parse (*e.g.*, the knowledge that r_2 has two children and that r_3 has a single child). Nonetheless, even if we cannot ensure that a parse traversal uniquely identifies with a parse, it *might* still not pose as a problem, as long as we can ensure that a pNFST's machine weight does not diverge through other means (see the discussion in §8.2.2).

tree traversal. While M' seemed to have built up unnecessarily complicated parses to do a very simple string manipulation task (when compared to M), we will show later in §8.3 that complicated parse-building function (*e.g.*, $\downarrow_{M'}$ above) can be pragmatically built from smaller components. We will also show that such parses allow us to simulate familiar regular operations of WFSTs.

8.3 Neural rational operators

In this section, we define neural rational operators (NROs) that generalize regular expression operators. More formally, we would like to implement (as algorithms) *rational operations*, as well as composition (Mohri, 2009) for pNFSTs. The properties of these rational operations are defined below:

Definition 8.3.1. ***Concatenation*** *Let M_1 and M_2 be pNFSTs. the concatenation $M_1.M_2$ is defined by*

$$\forall(\mathbf{x},\mathbf{y}) \in \Sigma^* \times \Delta^*, [\![M_1.M_2]\!](\mathbf{x},\mathbf{y}) \triangleq \sum_{\mathbf{x}=\mathbf{x}_1\mathbf{x}_2,\mathbf{y}=\mathbf{y}_1\mathbf{y}_2} [\![M_1]\!](\mathbf{x}_1,\mathbf{y}_1)[\![M_2]\!](\mathbf{x}_2,\mathbf{y}_2).$$

Union *Let M_1 and M_2 be pNFSTs. $M_1|M_2$ is defined by*

$$\forall(\mathbf{x},\mathbf{y}) \in \Sigma^* \times \Delta^*, [\![M_1|M_2]\!](\mathbf{x},\mathbf{y}) \triangleq [\![M_1]\!](\mathbf{x},\mathbf{y}) + [\![M_2]\!](\mathbf{x},\mathbf{y}).$$

Closure *Let M be a pNFST. M^**

$$[\![M^*]\!](\mathbf{x}, \mathbf{y}) \triangleq \sum_{n=0}^{\infty} [\![M^n]\!](\mathbf{x}, \mathbf{y}),$$

where $M^n = \overbrace{M. \ldots .M}^{n}$.

Composition *Let M_1 and M_2 be pNFSTs, where T_{M_1} has input and output alphabets Σ and Γ, and T_{M_2} input and output alphabets Γ and Δ. We say*

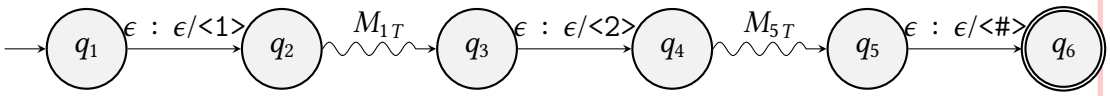
$$[\![M_1 \circ M_2]\!](\mathbf{x}, \mathbf{y}) \triangleq \sum_{\mathbf{z} \in \Gamma^*} \left[[\![M_1]\!](\mathbf{x}, \mathbf{z}) \cdot [\![M_2]\!](\mathbf{z}, \mathbf{y}) \right].$$

For weighted FSTs, these rational operations can be implemented by manipulating finite-state topology. But in our case, our NROs build new pNFSTs that have new finite-state topologies, new parsing functions, and new scoring functions. As a result, just like the redefined neuralization operator (Algorithm 3), there is an explicit 'compile phase' when NROs are being applied, where we parametrize T_M, $\downarrow_M$, and $[\![]\!]_M$.[7] We will describe algorithms that build these NROs in the compile phase.

8.3.1 Concatenation

[7] Alternatively, these functions can be presented as curried higher-order functions, where arguments that are functions of other pNFSTs are not explicitly precomputed. We opt for the 'compile/run' two phase presentation because it more closely mirrors our current implementation, where fixed computation graphs are 'compiled' before we evaluate them.

MFST 8.3.10: Topology of T_M in Algorithm 4



The concatenation NRO (denoted with '.') concatenates two pNFSTs M_1 and M_2 (Definition 8.3.1). Let $M = M_1.M_2$, there exist many algorithms that build an pNFST M that satisfies the concatenation property of Definition 8.3.1. In this thesis we mere give an example algorithm `BuildConcat`, such that $M = \texttt{BuildConcat}(M_1, M_2) = M_1.M_2$ in Algorithm 4. At a high level, we build T_M, $\downarrow_M$, and $[\![]\!]_M$ as follows:

- T_M is a concatenation of the two marked FSTs T_{M_1} and T_{M_2}, separated by three special marked symbols, as depicted in MFST 8.3.10; and
- the parse building function $\downarrow_M$ takes as input a mark string in $T_{M\Omega}$, where every string is of the form <1>**a**<2>**b**#. $\downarrow_M$ returns a parse with non-literal `Concat` as its root, and $\downarrow_{M_1}$ (**a**) and $\downarrow_{M_2}$ (**b**) as left and right subtrees respectively; and
- the parse scoring function $[\![]\!]_M$ is a function that takes a parse whose root node is `Concat` as input, and returns $[\![t_1]\!]_{M_1} \cdot [\![t_2]\!]_{M_2}$, where t_1 and t_2 are the left and right subtrees of the root node `Concat` respectively.

We can show that `BuildConcat` behaves as expected:

Proposition 8.3.1. *$\texttt{BuildConcat}(M_1, M_2) = M_1.M_2$ (as defined in Definition 8.3.1).*

Proof. Let $M = \texttt{BuildConcat}(M_1, M_2)$. By equation (8.2) we have

$$[\![M]\!](\mathbf{x}, \mathbf{y}) = \sum_{\boldsymbol{\omega} \in \mathbf{x} \circ T_M \circ \mathbf{y}} [\![\downarrow_M (\boldsymbol{\omega})]\!]_M.$$

We observe that there is a bijective mapping between $T_{M_1\Omega} \times T_{M_2\Omega}$ and $T_{M\Omega}$: $\forall \boldsymbol{\omega} \in T_{M\Omega}$, $\boldsymbol{\omega}$ is of the form <1>$\boldsymbol{\omega}_1$<2>$\boldsymbol{\omega}_2$# where $\boldsymbol{\omega}_1 \in T_{M_1\Omega}$ and $\boldsymbol{\omega}_2 \in T_{M_2\Omega}$. Therefore, we further have

$$[\![M]\!](\mathbf{x}, \mathbf{y}) = \sum_{\boldsymbol{\omega}_1 \in \mathbf{x}_1 \circ T_{M_1} \circ \mathbf{y}_1, \boldsymbol{\omega}_2 \in \mathbf{x}_2 \circ T_{M_2} \circ \mathbf{y}_2} [\![\downarrow_M (\texttt{<1>}\boldsymbol{\omega}_1\texttt{<2>}\boldsymbol{\omega}_2\texttt{\#})]\!]_M,$$

where $\mathbf{x}_1.\mathbf{x}_2 = \mathbf{x}$ and $\mathbf{y}_1.\mathbf{y}_2 = \mathbf{y}$. Furthermore, by the definitions of $\downarrow_M$ and $[\![]\!]_M$, we have

$$\begin{aligned}[\![M]\!](\mathbf{x}, \mathbf{y}) &= \sum_{\boldsymbol{\omega}_1 \in \mathbf{x}_1 \circ T_{M_1} \circ \mathbf{y}_1, \boldsymbol{\omega}_2 \in \mathbf{x}_2 \circ T_{M_2} \circ \mathbf{y}_2} [\![\downarrow_{M_1} (\boldsymbol{\omega}_1)]\!]_{M_1} [\![\downarrow_{M_2} (\boldsymbol{\omega}_2)]\!]_{M_2} \\ &= [\![M_1.M_2]\!](\mathbf{x}, \mathbf{y}).\end{aligned}$$

□

8.3.2 Union

The **union** NRO (denoted with '|') takes the union of two pNFSTs M_1 and M_2. As in §8.3.1, we give an example algorithm `BuildUnion` (Algorithm 5), such that $M = \texttt{BuildUnion}(M_1, M_2) = M_1|M_2$. Again, the high level-sketch is that we build T_M, $\downarrow_M$, and $[\![]\!]_M$ as follows:

Algorithm 4: `BuildConcat`(M_1, M_2).

Input:

- M_1 and M_2 are pNFSTs

Output: A pNFST $M = M_1.M_2$

Find three symbols $\in \Omega$ that have not been used. If this impossible, raise an exception. Otherwise, refer to these three symbols as <1>, <2>, #. Mark them as special symbols $\in \Omega$;

Parametrize functions $\downarrow_M$ and $[\![]\!]_M$ using M_1, M_2, and {<1>, <2>, #} ;

return $(T_{M'}(), \downarrow_M, [\![]\!]_M)$;

Function $T_M()$**:**

- $q_1 \longleftarrow$ a marked FST initial state ;
- $q_6 \longleftarrow$ a marked FST accepting state ;
- $q_{\{2,3,4,5\}} \longleftarrow$ marked FST states ;
- Add arcs between q_1 and q_2, and q_3 and q_4, and q_5 and q_6 as depicted in MFST 8.3.10 ;
- Add an arc $\epsilon : \epsilon/\epsilon$ between q_2 and the start state of M_{1T} ;
- **foreach** *final state* q_F *of* M_{1T} **do**
  - Add an arc $\epsilon : \epsilon/\epsilon$ between q_F and q_3 ;
- **end**
- Add an arc $\epsilon : \epsilon/\epsilon$ between q_4 and the start state of M_{2T} ;
- **foreach** *final state* q_F *of* M_{2T} **do**
  - Add an arc $\epsilon : \epsilon/\epsilon$ between q_F and q_5 ;
- **end**
- $M_T \longleftarrow$ a marked FST consisting of all states and arcs of M_{1T}, M_{2T}, $q_{1...6}$, and all arcs connected to $q_{1...6}$, with q_1 as the initial state, and q_6 as the sole accepting state ;

Function $\downarrow_M(\boldsymbol{\omega})$**:**

- Assert {<1>, <2>, #} all occur exactly once in $\boldsymbol{\omega}$;
- Assert $\boldsymbol{\omega}$ starts with symbol <1> and ends with symbol # ;
- **a** $\longleftarrow$ substring of $\boldsymbol{\omega}$ that occur between <1> and <2> ;
- **b** $\longleftarrow$ substring of $\boldsymbol{\omega}$ that occur between <2> and # ;
- $R \longleftarrow$ (`Concat` $M_{1\downarrow}$(**a**) $M_{2\downarrow}$(**b**)) ;
- **return** R ;

Function $[\![]\!]_M(t)$**:**

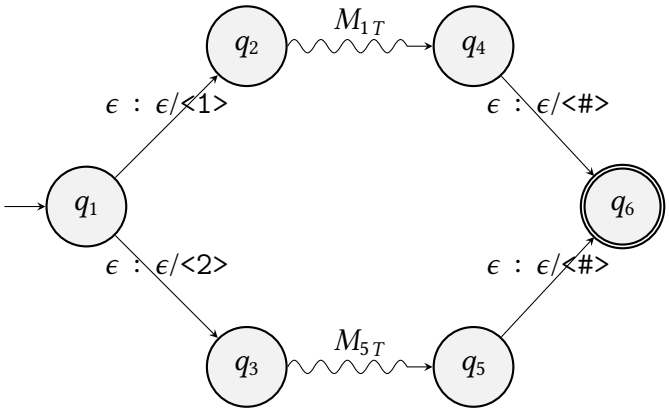
- Assert the root of t is `Concat` ;
- Assert the root of t has two children. Let the subtree rooted at the left child be t_1, and the subtree rooted at the right child be t_2 ;
- **return** $[\![t_1]\!]_{M_1} \cdot [\![t_2]\!]_{M_2}$;

- T_M is a union of the two marked FSTs T'_{M_1} and T'_{M_2}, where T'_{M_1} is obtained by adding two arcs to the marked FST T_{M_1}; and T'_{M_2} is similarly obtained from T_{M_2}, as illustrated in MFST 8.3.11.

- The parse building function $\downarrow_M$ takes as input a mark string $\boldsymbol{\omega}$ in $T_{M\Omega}$, where every string is of the form <1>$\boldsymbol{\omega}'$# or <2>$\boldsymbol{\omega}'$#. $\downarrow_M$ returns a parse with root node `Union` and two children. The left child is the first symbol of $\boldsymbol{\omega}$ (*i.e.*, either <1> or <2>). And the right child is a subtree: $\downarrow_{M_1}(\boldsymbol{\omega}')$.

- The parse scoring function $[\![]\!]_M$ takes as input a parse with `Union` as its root node, with exactly two children. If the root node `Union` has a left child <1>, return $[\![t']\!]_{M_1}$. Otherwise (*i.e.*, the root node's left child is <2>), return $[\![t']\!]_{M_2}$.

MFST 8.3.11: Topology of T_M in Algorithm 5



Algorithm 5: BuildUnion(M_1, M_2).

Input:

- M_1 and M_2 are pNFSTs

Output: A pNFST $M = M_1.M_2$

Find two symbols $\in \Omega$ that have not been used. If this impossible, raise an exception. Otherwise, refer to these three symbols as <1> and <2>. Mark them as special symbols $\in \Omega$;

Parametrize functions $\downarrow_M$ and $[\![]\!]_M$ using M_1, M_2, and {<1>, <2>, #} ;

return $(T_{M'}(), \downarrow_M, [\![]\!]_M)$;

Function $T_M()$**:**

- $q_1 \leftarrow$ a marked FST initial state ;
- $q_6 \leftarrow$ a marked FST accepting state ;
- $q_{\{2,3,4,5\}} \leftarrow$ marked FST states ;
- Add arcs between q_1 and q_2, and q_1 and q_3, q_4 and q_6, and q_5 and q_6, as depicted in MFST 8.3.11 ;
- Add an arc $\epsilon : \epsilon/\epsilon$ between q_2 and the start state of M_{1T} ;
- Add an arc $\epsilon : \epsilon/\epsilon$ between q_3 and the start state of M_{2T} ;
- **foreach** *final state* q_F *of* M_{1T} **do**
  - Add an arc $\epsilon : \epsilon/\epsilon$ between q_F and q_4 ;
- **end**
- **foreach** *final state* q_F *of* M_{2T} **do**
  - Add an arc $\epsilon : \epsilon/\epsilon$ between q_F and q_5 ;
- **end**
- $M_T \leftarrow$ a marked FST consisting of all states and arcs of M_{1T}, M_{2T}, $q_{1...6}$, and all arcs connected to $q_{1...6}$, with q_1 as the initial state, and q_6 as the sole accepting state ;

Function $\downarrow_M(\omega)$**:**

- Assert <#> occurs exactly once in ω ;
- Assert either <1> occurs exactly once in ω, or <2> occurs exactly once in ω ;
- Assert ω starts with either <1> or <2>;
- Assert ω ends with <#>;
- $\omega' \leftarrow$ substring of ω that occur after either <1> or <2> and before <#>;
- **if** ω *starts with* <1> **then**
  - $R \leftarrow$ (Union <1>$\downarrow_{M_1}(\omega')$) ;
- **else**
  - $R \leftarrow$ (Union <2>$\downarrow_{M_2}(\omega')$) ;
- **return** R ;

Function $[\![]\!]_M(t)$**:**

- $r \leftarrow$ root of t ;
- Assert r = Union ;
- Assert r has two children.;
- Assert left child of r is either <1> or <2>.;
- $t' \leftarrow$ right subtree of r ;
- **if** *left child of* r *is* <1> **then**
  - **return** $[\![t']\!]_{M_1}$;
- **else**
  - **return** $[\![t']\!]_{M_2}$;

Proposition 8.3.2. *`BuildUnion`*$(M_1, M_2) = M_1|M_2$ *(as defined in Definition 8.3.1).*

8.3.3 Composition

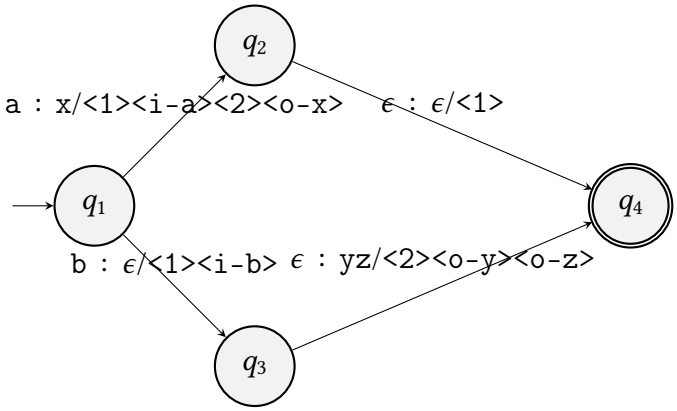
We define the **composition** NRO (denoted with '$\circ$') to compose two pNFSTs M_1 and M_2. As in §§8.3.1 and 8.3.2, we exhibit an example algorithm `BuildComposition`, such that M = `BuildComposition`$(M_1, M_2) = M_1 \circ M_2$. Similar to Algorithms 4 and 5, `BuildComposition` builds T_M, $\downarrow_M$, and $[\![]\!]_M$:

- The construction of T_M is more involved in the `BuildComposition` case. We first obtain MFSTs T'_{M_1} and T'_{M_2} from T_{M_1} and T_{M_2}, where each arc is prepended with special symbols <1> and <2> respectively. An example of T'_{M_1} is in MFST 8.3.12.

- $\downarrow_M$ takes as input a mark string $\boldsymbol{\omega}$ in $T_{M\Omega}$, where every string is of the pattern $(a\boldsymbol{\omega}')^*$, $a \in \{$<1>, <2>$\}$, and $\boldsymbol{\omega}'$ is a string $\in \Omega^*$ that does not contain either <1> or <2>. $\downarrow_M$ collects all subsequences $\boldsymbol{\omega}'$, and concatenate them into two strings: $\boldsymbol{\omega}_1$ from subsequences that were prefixed by <1>, and $\boldsymbol{\omega}_2$ by <2>. Finally, $\downarrow_M$ returns a tree with root node `Composition`, with $\downarrow_{M_1}(\boldsymbol{\omega}_1)$ and $\downarrow_{M_2}(\boldsymbol{\omega}_2)$ as left and right trees, respectively.

- $[\![]\!]_M$ takes as input an parse t with `Composition` as its root node, with exactly two subtrees: the left tree t_1 and the right tree t_2. $[\![t]\!]_M$ returns the scalar $[\![t_1]\!]_{M_1} \cdot [\![t_2]\!]_{M_2}$.

MFST 8.3.12: Example topology of T'_{M_1} in Algorithm 6.



Algorithm 6: `BuildComposition`(M_1, M_2).

Input:

- M_1 and M_2 are pNFSTs

Output: A pNFST $M = M_1.M_2$

Find two symbols $\in \Omega$ that have not been used. If this impossible, raise an exception. Otherwise, refer to these three symbols as <1> and <2>. Mark them as special symbols $\in \Omega$;

Parametrize functions $\downarrow_M$ and $[\![]\!]_M$ using M_1, M_2, and {<1>, <2>} ;

return $(T_{M'}(), \downarrow_M, [\![]\!]_M)$;

Function $T_M()$**:**

  $T'_{M_1} \leftarrow$ modified T_{M_1}, with mark symbol <1> prepending each arc (in the manner of MFST 8.3.12) ;

  $T'_{M_2} \leftarrow$ modified T_{M_2}, with mark symbol <2> prepending each arc;

  return $T_{M_1} \circ T_{M_2}$;

Function $\downarrow_M(\omega)$**:**

  $\omega_1 \leftarrow$ concatenation of longest subsequences of ω, each of which starts with <1>, and does not contain <2> ;

  $\omega_2 \leftarrow$ concatenation of longest subsequences of ω, each of which starts with <2>, and does not contain <1> ;

  return (`Composition` $\downarrow_{M_1} (\omega_1) \downarrow_{M_2} (\omega_2)$) ;

Function $[\![]\!]_M(t)$**:**

  $r \leftarrow$ root of t ;

  Assert r = `Composition` ;

  Assert r has exactly two children. ;

  $t_1 \leftarrow$ left subtree of r ;

  $t_2 \leftarrow$ right subtree of r ;

  return $[\![t_1]\!]_{M_1} \cdot [\![t_2]\!]_{M_2}$

Proposition 8.3.3. *`BuildComposition`$(M_1, M_2) = M_1 \circ M_2$ (as defined in Definition 8.3.1).*

8.3.4 Kleene closure

The **Kleene closure** NRO (denoted with '*') takes a pNFST M_0 as input. We give an example `BuildClosure`, such that M = `BuildClosure`$(M_0) = {M_0}^*$. Our construction below bears

similarity to that of `BuildConcat` (§8.3.1): here we also make use of a special mark symbol to indicate string boundaries.

As with our other NROs, `BuildClosure` (Algorithm 7) define T_M, $\downarrow_M$, and $[\![]\!]_M$:

- $T_M = (T_{M_0}.S)^*$ is a Kleene closure of the MFST $T_{M_0}.S$, where the MFST S acts as a sentinel. Each mark string $\boldsymbol{\omega} \in (T_M)_\Omega$ is either an empty string, or of form $\boldsymbol{\omega}_1 \# \dots \boldsymbol{\omega}_K \#$, where $\forall k \in [1, K]$, $\boldsymbol{\omega}_k \in (T_{M_0})_\Omega$, and $s \in \Omega$ is a 'guard' mark only used in this context.

- $\downarrow_M$ takes as input a mark string $\boldsymbol{\omega}$, where either $\boldsymbol{\omega} = \epsilon$, or $\boldsymbol{\omega} = \boldsymbol{\omega}_1 \# \dots \boldsymbol{\omega}_K \# \in (T_M)_\Omega$. $\forall \boldsymbol{\omega} \neq \epsilon$, $\downarrow_M (\boldsymbol{\omega})$ is a K-ary tree with root node `Closure`, with $K \downarrow M_0(\boldsymbol{\omega}_k), \forall k \in [0..K]$, as the subtrees.

- $[\![]\!]_M$ takes as input a parse t with `Closure` as its root node. When t is a singleton tree, $[\![]\!]_M(t) = 1$. Otherwise $[\![]\!]_M(t) = \prod_{k=1}^{K} [\![]\!]_{M_0}(t_k)$, where t_k is the k-th subtree of t.

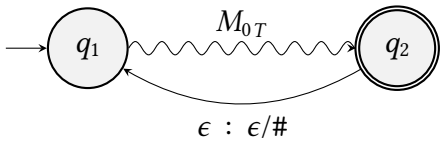
Algorithm 7: `BuildClosure`(M_0).

Input:

- M_0 is a pNFST

Output: A pNFST $M = M_0^*$
Find a symbol $\in C$ that have not been used. If this impossible, raise an exception. Otherwise, refer to this symbols as # ;
Parametrize functions $\downarrow_M$ and $[\![]\!]_M$ using M_0 and # ;
return $(T_M(), \downarrow_M, [\![]\!]_M)$;
Function T_M `()`**:**
    $q_1 \leftarrow$ a marked FST initial state ;
    $q_2 \leftarrow$ a marked FST accepting state ;
    Add arcs between q_1 and q_2 as depicted in MFST 8.3.13 ;
Function $\downarrow_M (\boldsymbol{\omega})$**:**
    $\boldsymbol{\omega}_1 \dots \boldsymbol{\omega}_K \leftarrow$ substrings of $\boldsymbol{\omega}$ separated by # ;
    foreach $k \in [1 \dots K]$ **do**
        $t_k \leftarrow \downarrow_{M_0} (\boldsymbol{\omega}_k)$;
    end
    $R \leftarrow$ (`Closure` $t_1 \dots t_k$) ;
    return R ;
Function $[\![]\!]_M (t)$**:**
    Assert the root of t is `Closure` ;
    $t_1 \dots t_K \leftarrow$ subtrees of t ;
    return $\prod_{k=1}^{K} [\![t_k]\!]_{M_0}$;

MFST 8.3.13: Topology of T_M in Algorithm 7



Proposition 8.3.4. *`BuildClosure`*$(M_0) = M_0^*$ *(as defined in Definition 8.3.1).*

8.4 Capacity of NREs with autoregressive G's

In §7.7 we said that a major limitation of NFSTs with autoregressive scoring functions — (τ, G) — is that the MFST component τ may not be very interpretable, if we are to capture certain 'hard' string pair distributions. On the other hand, pNFSTs can capture more difficult distributions, even with MFSTs that have interpretable topologies. The additional power is due to the flexibility of processing parses in a pNFST in a different order than their corresponding subsequences. Such flexibility allows 'easy' subsequences in a mark string to be processed first, which is not possible under an autoregressive scoring function that scores a mark string in a left-to-right fashion.

In this section, we use 3-SAT as an example, to make the point that some MFST topologies do *not* have any good corresponding mark string scoring functions, if we limit ourselves to autoregressive model families (*i.e.*, ELNCP weighted languages); but such problem goes away under pNFSTs. Under this example task, we want to transduce a Boolean CNF formula

$\boldsymbol{\phi}$ of variables $\{A_n : 1 \leq n \leq N, N \in \mathbb{N}\}$, into a string of satisfying variable assignments $\mathbf{a} \in \mathbb{B}^*$. For example, let $\boldsymbol{\phi}$ = `(A1 or ~A2 or A3) (A1 or ~A4)`, a satisfying variable assignment would be `1101` — namely $A_1 = 1, A_2 = 1, A_3 = 0, A_4 = 1$.

It is straightforward to design an MRE (namely MRE 8.4.22, with components from MREs 8.4.23–8.4.29) whose relation is a superset of valid transductions:

MRE 8.4.22: SatRE

Clause*Assignment*

MRE 8.4.23: Assignment

(ε:0/*0*|ε:1/*1*)

MRE 8.4.24: Clause

OneLit|TwoLit|ThreeLit

MRE 8.4.25: NegVar

(~:ε/*negate*|ε:ε/ε)Var

MRE 8.4.26: Var

(A1:ε/*A1*|A2:ε/*A2*...)

MRE 8.4.27: OneLit

(: ε/*start-clause* NegVar) : ε/*end-clause*

MRE 8.4.28: TwoLit

(: ε/*start-clause* NegVar or : ε/*or* NegVar) : ε/*end-clause*

MRE 8.4.29: ThreeLit

(: ε/*start-clause* NegVar or : ε/*or* NegVar or : ε/*or* NegVar) : ε/*end-clause*

However, there is no $G \in$ ELNCP such that (SatRE, G) captures this transduction task. More formally:

Proposition 8.4.1. *There is no* $G \in$ ELNCP *such that* $\forall \mathbf{x} \in \Sigma^*, \forall \mathbf{y} \in \Delta^*, N[\text{SatRE}, G](\boldsymbol{\phi}, \mathbf{a}) > 0$ *if and only if* $\boldsymbol{\phi}$ *is a formula satisfied by variable assignment* $\mathbf{a}$*, assuming* NP $\not\subseteq$ P/poly.

Proof. Mark strings $\boldsymbol{\omega} \in (\boldsymbol{\phi} \circ \text{SatRE} \circ \mathbf{a})_\Omega$ have form $\boldsymbol{\phi}\mathbf{a}$. We know from Theorem 3.2.2 that there is no $G \in$ ELNCP such that $G(\boldsymbol{\phi}\mathbf{a}) > 0 \iff \mathbf{a}$ satisfies $\boldsymbol{\phi}$. □

A possible fix is to add dummy marks at the start of mark strings, as latent variables that hold a hypothesis of variable assignments, which G can *look behind* at, to rule out both invalid formulae and variable assignments that appear later. Indeed we used such an MFST construction (MFST 7.3.5) to show how NFSTs can capture all NP-complete problems (Proposition 7.3.2). However, such a technique introduces marks that do not annotate the

string transduction in a meaningful way: they are only placeholders for random bits. Such a technique also increases the number of latent variables in a mark string, which may in turn increases the compute requirement of approximate inference at a given level.

Alternatively, we consider NRE 8.4.8, which is an NRE that has an MRE component largely identical to MRE 8.4.22, with the only difference that we replace the MRE AssignMent (MRE 8.4.23) with NRE 8.4.7:

NRE 8.4.7: AssignmentNRE: pNFST replacement of Assignment (MRE 8.4.23)

$N[(\varepsilon:0/0|\varepsilon:1/1), G_0]$

And NRE 8.4.8 is itself neuralized:

NRE 8.4.8: SatNRE: pNFST replacement of SatRE (MRE 8.4.22)

$N[\text{Clause}^*\text{AssignmentNRE}, G_1]$

Proposition 8.4.2. *There exists* $p : \Omega^* \longrightarrow \mathbb{R}_{>0} \in$ ELN, *a polytime algorithm* transform $:$ $\Omega^* \longrightarrow \Omega^*$, *and parse scoring functions* G_0, G_1 *such that*

- $\forall \boldsymbol{\omega} \in \Omega^*, [\![\downarrow_{SatNRE} (\boldsymbol{\omega})]\!]_{SatNRE} = p(\text{transform}(\boldsymbol{\omega}))$*; and*
- $\forall \mathbf{x} \in \Sigma^*, \forall \mathbf{y} \in \Delta^*, [\![SatNRE]\!](\boldsymbol{\phi}, \mathbf{a}) > 0$ *if and only if* $\boldsymbol{\phi}$ *is a formula satisfied by variable assignment* $\mathbf{a}$.

Proof. Intuitively, the weight of every mark string $\boldsymbol{\omega}$ in the pNFST SatNRE is a product of two parses' weights, under G_0 and G_1 respectively. Under the construction of NREs 8.4.7

and 8.4.8, G_0 is applied first to score a parse t_0 under AssignmentNRE, followed by the application of G_1 on its parse $t_1 = \downarrow_{\text{SatNRE}}(\boldsymbol{\omega})$. Also note that t_0 is a subtree of t_1 by our definition of the neuralization operation (§8.2), and can be extracted from t_1 by a traversal from the root of t_0.

Now we parametrize the parse scoring functions as $G_0(t_0) = p(\boldsymbol{\omega}')$, and $G_1(t_1) = p(\boldsymbol{\omega}'')$, where $\boldsymbol{\omega}'$ is the mark subsequence obtained from a left-to-right traversal of t_0, and $\boldsymbol{\omega}''$ is the mark subsequence obtained from a left-to-right traversal of t_1. In other words, we adopt for the double-scoring variant of partial neuralization (§8.2.2). There is a bijective mapping between $(\boldsymbol{\omega}', \boldsymbol{\omega}'')$ and their corresponding $\boldsymbol{\omega} \in T_{\text{SatNRE}\Omega}$. We define the function that extracts $\boldsymbol{\omega}'.\boldsymbol{\omega}''$ from $\boldsymbol{\omega}$ as transform. transform can be implemented as an algorithm that traverses through t_1 twice, to extract $\boldsymbol{\omega}'$ and $\boldsymbol{\omega}''$ respectively. Since each traversal takes time linear in the tree's size ($O(|V| + |E|)$), transform is a polytime algorithm.

We first argue $\forall \boldsymbol{\phi} \in \Sigma^*, \forall \mathbf{a} \in \Delta^*, [\![\text{SatNRE}]\!](\boldsymbol{\phi}, \mathbf{a}) > 0 \iff \exists \boldsymbol{\omega} \in (\boldsymbol{\phi} \circ T_{\text{SatNRE}} \circ \mathbf{a})_\Omega$, such that $p(\text{transform}(\boldsymbol{\omega}))$. To see this, we rewrite

$$
\begin{aligned}
[\![\text{SatNRE}]\!](\boldsymbol{\phi}, \mathbf{a}) &= \sum_{\boldsymbol{\omega} \in \boldsymbol{\phi} \circ T_{\text{SatNRE}} \circ \mathbf{a}_\Omega} [\![\downarrow_{\text{SatNRE}}(\boldsymbol{\omega})]\!]_{\text{SatNRE}} \\
&= \sum_{\boldsymbol{\omega} \in \boldsymbol{\phi} \circ T_{\text{SatNRE}} \circ \mathbf{a}_\Omega} [\![\downarrow_{\text{SatNRE}}(\boldsymbol{\omega})]\!]_{\text{SatNRE}} \\
&= \sum_{\boldsymbol{\omega} \in \boldsymbol{\phi} \circ T_{\text{SatNRE}} \circ \mathbf{a}_\Omega} p(\boldsymbol{\omega}' \boldsymbol{\omega}'').
\end{aligned}
$$

We then argue $p \in \text{ELN}$. The construction of p is very similar to that of the proof

of Theorem 3.3.1: p has support over strings of the form $\omega'\omega''$. ω' is a sequence of self-consistent variable assignments; and it is efficient to decide whether their prefix probability > 0, just by checking its self-consistency. As for the conditional distribution $p(\cdot \mid \omega')$, the construction largely follows our proof of Theorem 3.3.1. □

8.5 Tackling the cold start problem in slot-filling tasks using NREs

The compositional nature of NREs also allows us to *decompose* a transduction task into multiple ones: given a transduction task in the form of an NRE, we may be able to train its pNFST components on their respective subtasks separately. Afterwards, we put the trained components together using NROs (§8.3), which define a new weighted string relation as functions of the components' relations.

The *decompositional* view is particularly helpful when we have a transduction task that has little training data, but can be regarded as a combination of several smaller tasks, where there is much training data. Adding new intents to digital assistants is one such example, as we sketched in § 1: we may not have enough utterances to train a good NFST to recognize the new intent; but we may have existing corpora to recognize slot types of the new intent.

In this section, we experiment on the SNIPS dataset (first introduced in §7.5.1), and specifically choose 'PlayMusic' as our 'new' intent. The PlayMusic intent has slot types such as `album`, `artist`, `track`, `playlist`, etc. The standard SNIPS dataset contains 2,000

utterances of PlayMusic instructions. In this section we simulate a *cold start* situation — that is, we hold out a majority of these PlayMusic utterances in the training dataset. Without signal from training data, we observe that the performance drops significantly across all skills, and particularly so for PlayMusic.

However, there are datasets that provide ample data for training NFSTs that recognize slot types of PlayMusic. For example, to recognize the slot types `album`, `artist`, and `track`, we can scrape the Spotify Million Playlist Dataset (Chen et al., 2018a) for appropriate data. We can subsequently train NFSTs that recognize these slot types. Finally, these NFSTs are combined in an NRE to recognize PlayMusic intents.

In the following subsections, we will describe how we construct an decomposable NRE in §8.5.1. We will also describe how we parametrize and train these components. Finally, we discuss the experiment results.

8.5.1 Decomposable SNIPS NRE

We define DecomposableSnipsNRE (NRE 8.5.9) as a neuralization of a union of several intent-specific NREs, which implement BIO tagging through the use of NREs such as NREs 8.5.10–8.5.13 and MREs 7.5.12 and 7.5.13. We also neuralize all slot types (*i.e.*, Word sequences that transduce into `B` and `I` tags) and annotate with indicating marks.

The pNFST defined in DecomposableSnipsNRE will first score parses that match any slot types (*e.g.*, under G'_θ in AlbumNRE). After all slot types have been scored, the parse of an entire mark string from $(T_{\text{DecomposableSnipsNRE}})_\Omega$ will be scored under G_θ.

NRE 8.5.9: DecomposableSnipsNRE

$N[(\text{PlayMusicNRE}|\text{BookRestaurant}|\ldots), G_\theta]$

NRE 8.5.10: PlayMusicNRE

$(\text{Outside}|\text{AlbumNRE}|\text{ArtistNRE}|\text{TrackNRE}|\ldots)^*$

NRE 8.5.11: AlbumNRE

$N[(\varepsilon : \texttt{B}/\textit{B-album})\text{Word}((\varepsilon : \texttt{I}/\textit{I-album})\text{Word})^*, G'_\theta]$

NRE 8.5.12: ArtistNRE

$N[(\varepsilon : \texttt{B}/\textit{B-artist})\text{Word}((\varepsilon : \texttt{I}/\textit{I-artist})\text{Word})^*, G''_\theta]$

NRE 8.5.13: TrackNRE

$N[(\varepsilon : \texttt{B}/\textit{B-track})\text{Word}((\varepsilon : \texttt{I}/\textit{I-track})\text{Word})^*, G'''_\theta]$

8.5.2 Experiment setup

We compare between two configurations:[8]

[8] Our experiments in this chapter are designed to test the hypothesis that the compositionality of NREs are useful in the cold-start scenario. But we note that there are other ways to make use of both datasets, for an even better performance. For example, it may be possible to simultaneously train on both SNIPS and external datasets, using some multitask loss function. We leave a more thorough study for future work.

Entry type	Training set size	Validation set size	Example
Track	375687	117627	`Rooba Rooba - From "Orange"`
Album	176515	61295	`Girls With Guitars`
Artist	87253	31464	`Cherry Pie`

Table 8.1: Statistics and example entries of track, album and artist entries extracted from Chen et al. (2018a).

End-to-end In this setup, we train DecomposableSnipsNRE in an end-to-end fashion.

Compositional In this setup, some NRE components in DecomposableSnipsNRE are separately trained. Specifically, we train AlbumNRE, ArtistNRE, TrackNRE on collections of corresponding entries extracted from Chen et al. (2018a). These NREs are trained using the hyperparameters listed in §7.5.2.3, with separate validation sets also extracted from Chen et al. (2018a). Some statistics of the training and validation datasets for training these NREs, as well as some example entries, are listed in Table 8.1.

We simulate a *cold-start* scenario, on **modified** snips datasets (§7.5.1): in these configurations we remove the majority of utterances under the `PlayMusic` intent from the training dataset, keeping only 10% to 50% of them in the training set. We keep the other intents intact. The validation and test sets are also unchanged. We additionally compare our results with the end-to-end configuration trained on all data, as well as vanilla NFST (§7.5) trained on both the 10% split, and the full dataset.

	PlayMusic-only F1		Overall F1	
% of retained PlayMusic utterances in training set	Compositional	End-to-end trained	Compositional	End-to-end trained
NRE — 10%	84.4	78.4	92.3	91.5
NRE — 20%	89.4	80.5	92.9	91.4
NRE — 30%	88.6	80.6	92.6	91.3
NRE — 40%	89.5	83.2	92.6	91.6
NRE — 50%	87.6	81.2	92.2	91.3
NRE — 100%	—	85.2	—	92.3
Vanilla NFST — 10%	—	80.6	—	92.3
Vanilla NFST — 100%	—	91.1	—	93.7

Table 8.2: Cold-start results

8.5.3 Results

The experiment results are in Table 8.2.[9] We first note that end-to-end trained NREs compare worse than end-to-end trained NFSTs on 10% and 100% splits: this is expected because under NRE 8.5.9, strings recognized by some slot types are generated independently, and not conditioned on other subsequences in NRE 8.5.9. Because of the conditional independence, mark string scoring functions in NREs may not be as expressive as that of (vanilla) NFSTs. On the other hand, such conditional independence allows composition of separately trained NREs. And such functionality certainly helps when we can train some individual NRE components on much larger external datasets: with as few as 20% amount of the original PlayMusic utterances in the training set, the compositional NRE (with the

[9] The pairwise differences between compositional and end-to-end trained runs are all statistically significant ($p < .05$) under the permutation test.

help of pretrained components) achieves performance slightly worse than an NFST trained on the full dataset.

8.6 Conclusion and future work

In this chapter, we have introduced neural regular expressions (NREs), a generalization of both NFSTs and (weighted) regular expressions, to address some limitations of NFSTs pointed out in §7.7. With the help of partially neuralized finite-state transducers, mark strings in an NRE may be scored in a non-autoregressive fashion, even with autoregressive mark string scoring functions. Such added expressiveness makes NREs with autoregressive mark string scoring functions strictly more powerful than their NFST counterparts (§8.4). It also leads to the possibility of combining separately trained NREs into a larger unit, for a new task (§8.5).

Under NREs, (autoregressive) mark string scoring functions are effectively sequence distributions that are conditioned on trees. In this thesis we do not fully explore architecture designs that fully exploit such structural information. We also have not considered architecture designs that naturally incorporate conditioned-upon information, *e.g.*, Transformers. We expect that a more extensive search of architectures will lead to a considerable increase in performance, which we leave as future work.

Chapter 9

Conclusions

9.1 Contributions

This thesis has made both theoretical and methodological contributions to the problem of sequence modeling.

Theoretical contributions. In § 3 we have studied the difference in expressiveness of various parametrizations of sequence models, and created a expressiveness hierarchy of sequence models. In particular, we found that autoregressive sequence models, despite their popularity in text generation applications, are limited in expressiveness. Moreover, such expressiveness problem cannot be mitigated assuming these models have compact parameters. In § 5 we further argued that expressiveness is not the whole story: energy-based sequence models that are sufficiently expressive suffer from parameter estimation

problems. In particular, likelihood-based model selection is undecidable for hard attention Transformers that have at least 5 layers. To summarize, our theoretical results suggest autoregressive latent-variable sequence models strike a good balance between tractability and expressive, which in turn motivate methods proposed in §§ 7 and 8.

Methodological contributions. In § 4 we have described a method for training bounded-length energy-based sequence models, and observed marginal but significant improvement over autoregressive baselines. In § 6 we have described neural particle smoothing, an amortized inference scheme for sampling from conditional sequence distributions. Through empirical results, we show that by learning to approximate lookahead functions, we can achieve good approximation with fewer samples than vanilla importance sampling. In § 7 we have described neuralized finite-state machines, and proposed neural finite-state transducers (NFSTs) as an effective sequence transduction paradigm for tasks that show monotonic alignments. We have proposed a method to effectively train NFSTs. We also devised a method to decode from them. Empirically we have shown that NFSTs compare very favorably against seq2seq models. In § 8, we further described neuralized regular expressions (NREs), which generalize NFSTs to allow for modularity.

9.2 Future work

This work initiates further research questions. We conclude from §§ 3 and 5 that autoregressive latent-variable sequence models as an appealing choice, because of their expressiveness

and tractability. However, these models have no known exact inference methods, and must resort to approximate methods, such as neural particle smoothing (§ 6). On the other hand, there are latent-variable sequence models that admit *exact* inference (Kim, 2021; Rastogi, Cotterell, and Eisner, 2016; Libovický and Fraser, 2021; Kong, Dyer, and Smith, 2016). These models belong in the class ELN and are therefore not expressive, under our classification in § 3. However, many of these models perform really well empirically — *why*? Specifically, we would like to answer the following questions:

Why do these 'weak' models work well in practice? The theoretical analysis in § 3 is mostly a worst-case result. It is possible that many sequence modeling tasks in real life can be approximated well using ELN weighted languages. If this is true, can we formally characterize these sequence modeling tasks as a weighted language class?

Do we need the full expressiveness of latent variables? As a concrete example, the expressiveness of NFSTs come from the expressiveness of their mark string scoring functions: there is no Markov assumption. However such expressiveness comes at a great cost — even forced decoding cannot be done exactly; and we must always use approximate inference methods. Would it be a good trade-off, if we make the mark string scoring functions Markov (and weak), and use the spare compute budget to featurize input and output strings? A large scale empirical study that compares between slow-but-expressive and fast-but-weak sequence models on a variety of tasks could help answer this question.

Bibliography

Adleman, Leonard (1978). "Two theorems on random polynomial time". In: *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pp. 75–83.

Agarwal, Sushant, Nivasini Ananthakrishnan, Shai Ben-David, Tosca Lechner, and Ruth Urner (2020). "On Learnability wih Computable Learners". In: *Proceedings of the 31st International Conference on Algorithmic Learning Theory*. Ed. by Aryeh Kontorovich and Gergely Neu. Vol. 117. Proceedings of Machine Learning Research. PMLR, pp. 48–60. URL: https://proceedings.mlr.press/v117/agarwal20b.html.

Aharoni, Roee and Yoav Goldberg (2017). "Morphological Inflection Generation with Hard Monotonic Attention". In: *ACL*. URL: http://aclweb.org/anthology/P17-1183.

Allauzen, Cyril, Michael Riley, and Johan Schalkwyk (2010). "Filters for Efficient Composition of Weighted Finite-State Transducers". In: *CIAA*.

Allauzen, Cyril, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri (2007). "OpenFst: A General and Efficient Weighted Finite-State Transducer Library". In: *CIAA*.

Amos, Brandon and J. Zico Kolter (2017). "OptNet: Differentiable Optimization as a Layer in Neural Networks". In: *ICML*. URL: https://arxiv.org/abs/1703.00443.

BIBLIOGRAPHY

Andor, Daniel, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins (2016). "Globally Normalized Transition-Based Neural Networks". In: *ACL*. URL: https://www.aclweb.org/anthology/P16-1231.

Andrews, Nicholas, Mark Dredze, Benjamin Van Durme, and Jason Eisner (2017). "Bayesian Modeling of Lexical Resources for Low-Resource Settings". In: *ACL*. URL: http://cs.jhu.edu/~jason/papers/#andrews-et-al-2017.

Appelt, D., Jerry R. Hobbs, J. Bear, David J. Israel, and M. Tyson (1993). "FASTUS: A Finite-state Processor for Information Extraction from Real-world Text". In: *IJCAI*.

Arora, Sanjeev and Boaz Barak (2009). *Computational Complexity: a Modern Approach*. Cambridge University Press. URL: https://www.cambridge.org/tw/academic/subjects/computer-science/algorithmics-complexity-computer-algebra-and-computational-g/computational-complexity-modern-approach.

Bahdanau, Dzmitry, Philemon Brakel, Kelvin Xu, Anirudh Goyal, Ryan Lowe, Joelle Pineau, Aaron Courville, and Yoshua Bengio (2017). "An actor-critic algorithm for sequence prediction". In: *ICLR*. URL: https://arxiv.org/abs/1607.07086.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (Sept. 2015). "Neural Machine Translation by Jointly Learning to Align and Translate". In: *ICLR*. URL: https://arxiv.org/abs/1409.0473.

Bailly, Raphaël and François Denis (2011). "Absolute convergence of rational series is semi-decidable". In: *Information and Computation* 209.3, pp. 280–295.

BIBLIOGRAPHY

Bakhtin, Anton, Yuntian Deng, Sam Gross, Myle Ott, Marc'Aurelio Ranzato, and Arthur Szlam (2021). "Residual Energy-Based Models for Text Generation". In: *JMLR* 22.40, pp. 1–41. URL: http://jmlr.org/papers/v22/20-326.html.

Bangert, Julian, S. Bratus, Rebecca Shapiro, and Sean W. Smith (2013). "The Page-Fault Weird Machine: Lessons in Instruction-less Computation". In: *WOOT*.

Bengio, Yoshua and Paolo Frasconi (1996). "Input-Output HMMs for Sequence Processing". In: *IEEE Transactions on Neural Networks* 7.5, pp. 1231–1249. URL: http://ieeexplore.ieee.org/document/536317/.

Bhattamishra, Satwik, Arkil Patel, and Navin Goyal (Nov. 2020). "On the Computational Power of Transformers and Its Implications in Sequence Modeling". In: *Proceedings of the 24th Conference on Computational Natural Language Learning*. Online: Association for Computational Linguistics, pp. 455–475. URL: https://aclanthology.org/2020.conll-1.37.

Bishop, Christopher M and Nasser M Nasrabadi (2006). *Pattern recognition and machine learning*. Vol. 4. 4. Springer.

blixt (2020). *Re: Teaching GPT-3 to Identify Nonsense*. https://news.ycombinator.com/item?id=23990902. Online (accessed Oct 23, 2020). URL: https://news.ycombinator.com/item?id=23990902.

Bogdanov, Andrej and Luca Trevisan (Oct. 2006). "Average-Case Complexity". In: *Foundations and Trends in Theoretical Computer Science* 2.1, 1–106. URL: https://doi.org/10.1561/0400000004.

BIBLIOGRAPHY

Booth, T.L. and R.A. Thompson (1973). "Applying Probability Measures to Abstract Languages". In: *IEEE Transactions on Computers* C-22.5, pp. 442–450.

Boser, Bernhard E., Isabelle M. Guyon, and Vladimir N. Vapnik (1992). "A Training Algorithm for Optimal Margin Classifiers". In: *COLT*. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.21.3818.

Bouchard-Côté, Alexandre, Percy Liang, Thomas Griffiths, and Dan Klein (2007). "A Probabilistic Approach to Diachronic Phonology". In: *EMNLP-CoNLL*, pp. 887–896. URL: http://www.aclweb.org/anthology/D/D07/D07-1093.

Brown, Tom B., 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 (2020). "Language Models are Few-Shot Learners". In: arXiv: 2005.14165 [cs.CL].

Burda, Yuri, Roger B. Grosse, and Ruslan Salakhutdinov (2016). "Importance Weighted Autoencoders". In: *CoRR* abs/1509.00519.

Buys, Jan and Phil Blunsom (June 2018). "Neural Syntactic Generative Models with Exact Marginalization". In: *NAACL*. URL: https://www.aclweb.org/anthology/N18-1086.

Chandrasekaran, Venkat, Nathan Srebro, and Prahladh Harsha (2008). "Complexity of Inference in Graphical Models". In: *UAI*. URL: https://dl.acm.org/doi/10.5555/3023476.3023485.

Chen, Ching-Wei, Paul Lamere, Markus Schedl, and Hamed Zamani (2018a). "Recsys Challenge 2018: Automatic Music Playlist Continuation". In: *Proceedings of the 12th ACM Conference on Recommender Systems*. RecSys '18. Vancouver, British Columbia, Canada: Association for Computing Machinery, 527–528. URL: https://doi.org/10.1145/3240323.3240342.

Chen, Yining, Sorcha Gilroy, Andreas Maletti, Jonathan May, and Kevin Knight (June 2018b). "Recurrent Neural Networks as Weighted Language Recognizers". In: *NAACL*. URL: https://aclanthology.org/N18-1205.

Chi, Zhiyi and Stuart Geman (June 1998). "Estimation of Probabilistic Context-Free Grammars". In: *Computational Linguistics* 24.2, 299–305. URL: https://dl.acm.org/doi/10.5555/972732.972738.

Chiang, David and Darcey Riley (2020). "Factor Graph Grammars". In: *NeurIPS*.

Child, Rewon, Scott Gray, Alec Radford, and Ilya Sutskever (2019). "Generating long sequences with sparse transformers". In: *ArXiv* abs/1904.10509. URL: https://arxiv.org/abs/1904.10509.

Cho, Kyunghyun, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio (2014). "Learning Phrase Representations using RNN Encoder-

Decoder for Statistical Machine Translation". In: *EMNLP*. URL: https://arxiv.org/abs/1406.1078.

Chomsky, Noam (1957). *Syntactic Structures*. The Hague: Mouton and Co.

Clark, Elizabeth, Tal August, Sofia Serrano, Nikita Haduong, Suchin Gururangan, and Noah A. Smith (Aug. 2021). "All That's 'Human' Is Not Gold: Evaluating Human Evaluation of Generated Text". In: *ACL-IJCNLP*. URL: https://aclanthology.org/2021.acl-long.565.

Cook, Stephen A. (1971). "The Complexity of Theorem-Proving Procedures". In: *STOC*. URL: https://doi.org/10.1145/800157.805047.

Cotterell, Ryan, John Sylak-Glassman, and Christo Kirov (2017). "Neural Graphical Models over Strings for Principal Parts Morphological Paradigm Completion". In: *EACL*. URL: http://aclweb.org/anthology/E17-2120.

Coucke, Alice, Alaa Saade, Adrien Ball, Théodore Bluche, Alexandre Caulier, David Leroy, Clément Doumouro, Thibault Gisselbrecht, Francesco Caltagirone, Thibaut Lavril, et al. (2018). "Snips Voice Platform: an embedded Spoken Language Understanding system for private-by-design voice interfaces". In: *arXiv preprint arXiv:1805.10190*, pp. 12–16.

Deng, Yuntian, Yoon Kim, Justin Chiu, Demi Guo, and Alexander Rush (2018). "Latent Alignment and Variational Attention". In: *NeurIPS*. URL: http://papers.nips.cc/paper/8179-latent-alignment-and-variational-attention.pdf.

Douc, Randal and Olivier Cappé (2005). "Comparison of resampling schemes for particle filtering". In: *Image and Signal Processing and Analysis, 2005. ISPA 2005. Proceedings of*

the 4th International Symposium on. IEEE, pp. 64–69. URL: https://arxiv.org/abs/cs/0507025.

Doucet, Arnaud and Adam M. Johansen (2009). "A tutorial on particle filtering and smoothing: Fifteen years later". In: *Handbook of Nonlinear Filtering* 12.656-704, p. 3. URL: https://www.stats.ox.ac.uk/~doucet/doucet_johansen_tutorialPF2011.pdf.

Dreyer, Markus and Jason Eisner (2009). "Graphical Models over Multiple Strings". In: *EMNLP*. URL: https://aclanthology.org/D09-1011.

Dreyer, Markus, Jason R. Smith, and Jason Eisner (2008). "Latent-Variable Modeling of String Transductions with Finite-State Methods". In: *EMNLP*. URL: http://cs.jhu.edu/~jason/papers/#dreyer-smith-eisner-2008.

Dyer, Chris, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith (2015). "Transition-Based Dependency Parsing with Stack Long Short-Term Memory". In: *ACL*. URL: http://www.aclweb.org/anthology/P15-1033.

Dyer, Chris, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith (2016). "Recurrent Neural Network Grammars". In: *HLT-NAACL*. URL: https://aclweb.org/anthology/N16-1024.

E, Haihong, Peiqing Niu, Zhongfu Chen, and Meina Song (2019). "A Novel Bi-directional Interrelated Model for Joint Intent Detection and Slot Filling". In: *ACL*. URL: https://aclanthology.org/P19-1544.

Eddy, Sean R. (1996). "Hidden Markov models." In: *Current opinion in structural biology* 6 3, pp. 361–5.

Eisner, Jason (Aug. 2001). "Expectation Semirings: Flexible EM for Finite-State Transducers". In: *Proceedings of the ESSLLI Workshop on Finite-State Methods in Natural Language Processing (FSMNLP)*. Ed. by Gertjan van Noord. Extended abstract (5 pages). Helsinki. URL: http://cs.jhu.edu/~jason/papers/#eisner-2001-fsmnlp.

Fillmore, Charles J and Collin Baker (2010). "A frames approach to semantic analysis". In: *The Oxford handbook of linguistic analysis*.

Finkel, Jenny Rose, Alex Kleeman, and Christopher D. Manning (2008). "Efficient, Feature-based, Conditional Random Field Parsing". In: *ACL*. URL: https://aclanthology.org/P08-1109.

Finkel, Jenny Rose, Christopher D. Manning, and Andrew Y. Ng (2006). "Solving the Problem of Cascading Errors: Approximate Bayesian Inference for Linguistic Annotation Pipelines". In: *EMNLP*. URL: http://dl.acm.org/citation.cfm?id=1610075.1610162.

Galanis, Andreas, Daniel Stefankovic, and Eric Vigoda (2016). "Inapproximability of the Partition Function for the Antiferromagnetic Ising and Hard-Core Models". In: *Combinatorics, Probability and Computing* 25, pp. 500 –559.

Ghazvininejad, Marjan, Vladimir Karpukhin, Luke Zettlemoyer, and Omer Levy (2020). "Aligned Cross Entropy for Non-Autoregressive Machine Translation". In: *ICML*.

Glynn, Peter W. (1990). "Likelihood ratio gradient estimation for stochastic systems". In: *Communications of the ACM* 33.10, pp. 75–84. URL: http://web.stanford.edu/~glynn/papers/1990/G90a.html.

Gontier, Nicolas, Koustuv Sinha, Siva Reddy, and C. Pal (2020). "Measuring Systematic Generalization in Neural Proof Generation with Transformers". In: *NeurIPS*. URL: https://arxiv.org/abs/2009.14786.

Gorman, Kyle, Lucas F.E. Ashby, Aaron Goyzueta, Arya McCarthy, Shijie Wu, and Daniel You (July 2020). "The SIGMORPHON 2020 Shared Task on Multilingual Grapheme-to-Phoneme Conversion". In: *Proceedings of the 17th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*. Online: Association for Computational Linguistics, pp. 40–50. URL: https://aclanthology.org/2020.sigmorphon-1.2.

Graves, A. (2016). "Adaptive Computation Time for Recurrent Neural Networks". In: *ArXiv* abs/1603.08983. URL: https://arxiv.org/abs/1603.08983.

Grice, H. P. (1975). "Logic and Conversation". In: *Syntax and Semantics: Vol. 3: Speech Acts*. Ed. by Peter Cole and Jerry L. Morgan. New York: Academic Press, pp. 41–58. URL: http://www.ucl.ac.uk/ls/studypacks/Grice-Logic.pdf.

Grigore, Radu (2016). "Java Generics are Turing Complete". In: *CoRR* abs/1605.05274. arXiv: 1605.05274. URL: http://arxiv.org/abs/1605.05274.

Gu, Jiatao, James Bradbury, Caiming Xiong, Victor O. K. Li, and Richard Socher (2018). "Non-Autoregressive Neural Machine Translation". In: *ICLR*.

Gu, Shixiang, Zoubin Ghahramani, and Richard E. Turner (2015). "Neural Adaptive Sequential Monte Carlo". In: *NIPS*. URL: https://arxiv.org/abs/1506.03338.

BIBLIOGRAPHY

Hakkani-Tür, Dilek, Gökhan Tür, Asli Celikyilmaz, Yun-Nung Chen, Jianfeng Gao, Li Deng, and Ye-Yi Wang (2016). "Multi-domain joint semantic frame parsing using bi-directional rnn-lstm." In: *Interspeech*, pp. 715–719.

Hao, Yiding, Dana Angluin, and Roberta Frank (2022). "Formal Language Recognition by Hard Attention Transformers: Perspectives from Circuit Complexity". In: *ArXiv* abs/2204.06618.

Hart, Peter E., Nils J. Nilsson, and Bertram Raphael (1968). "A Formal Basis for the Heuristic Determination of Minimal Cost Paths". In: 4.2, pp. 100–107. URL: http://ai.stanford.edu/~nilsson/OnlinePubs-Nils/PublishedPapers/astar.pdf.

Hayes, Bruce (1995). *Metrical Stress Theory: Principles and Case Studies*. University of Chicago Press. URL: http://press.uchicago.edu/ucp/books/book/chicago/M/bo3621567.html.

Heinrich, Philippe and Jonas Kahn (2018). "Strong identifiability and optimal minimax rates for finite mixture estimation". In: *The Annals of Statistics*.

Hinton, Geoffrey E. (Aug. 2002). "Training Products of Experts by Minimizing Contrastive Divergence". In: *Neural Comput.* 14.8, 1771–1800. URL: https://doi.org/10.1162/089976602760128018.

Hochreiter, Sepp and Jürgen Schmidhuber (Nov. 1997). "Long Short-Term Memory". In: *Neural Comput.* 9.8, 1735–1780. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

Hokamp, Chris and Qun Liu (2017). "Lexically Constrained Decoding for Sequence Generation Using Grid Beam Search". In: *ACL*. URL: https://aclanthology.org/P17-1141.

Hoy, Matthew B (2018). "Alexa, Siri, Cortana, and more: an introduction to voice assistants". In: *Medical reference services quarterly* 37.1, pp. 81–88.

Hu, J Edward, Huda Khayrallah, Ryan Culkin, Patrick Xia, Tongfei Chen, Matt Post, and Benjamin Van Durme (2019). "Improved lexically constrained decoding for translation and monolingual rewriting". In: *ACL*.

Huang, Ting-Hao Kenneth, Yun-Nung (Vivian) Chen, and Jeffrey P. Bigham (2017). "Real-time On-Demand Crowd-powered Entity Extraction". In: *ArXiv* abs/1704.03627.

Huang, Y., A. Sethy, K. Audhkhasi, and B. Ramabhadran (2018). "Whole Sentence Neural Language Models". In: *ICASSP*. URL: https://ieeexplore.ieee.org/document/8461734.

Hulden, Mans (2015). "Grammar Design with Multi-tape Automata and Composition". In: *Proceedings of the 12th International Conference on Finite-State Methods and Natural Language Processing 2015 (FSMNLP 2015 Düsseldorf)*. URL: https://aclanthology.org/W15-4807.

Hulst, Harry van der (2010). *Recursion and Human Language*. Berlin, Boston: De Gruyter Mouton. URL: https://www.degruyter.com/view/title/34061.

IEEE (2018). "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7". In: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pp. 1–3951.

Ihler, Alexander T. and David A. McAllester (2009). “Particle Belief Propagation”. In: *AISTATS*. URL: http://proceedings.mlr.press/v5/ihler09a.html.

Impagliazzo, Russell and Ramamohan Paturi (1999). “The Complexity of K-SAT”. In: *COCO*. URL: https://dl.acm.org/doi/10.5555/792764.793393.

Indyk, Piotr and Rajeev Motwani (1998). “Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality”. In: *STOC*. URL: https://doi.org/10.1145/276698.276876.

Jang, Eric, Shixiang Gu, and Ben Poole (2017). “Categorical Reparameterization with Gumbel-Softmax”. In: *ICLR*. URL: https://arxiv.org/abs/1611.01144.

Jelinek, Frederick (1980). “Interpolated estimation of Markov source parameters from sparse data”. In: *Proc. Workshop on Pattern Recognition in Practice, 1980*.

Jerrum, Mark and Alistair Sinclair (1993). “Polynomial-time approximation algorithms for the Ising model”. In: *SIAM Journal on computing* 22.5, pp. 1087–1116.

Josh (2013). *Parsing C is literally undecidable*. URL: https://blog.reverberate.org/2013/08/parsing-c-is-literally-undecidable.html.

Kann, Katharina and Hinrich Schütze (2016). “Single-Model Encoder-Decoder with Explicit Morphological Representation for Reinflection”. In: *ACL*. URL: http://www.aclweb.org/anthology/P16-2090.

Karp, Richard M. and Richard J. Lipton (1980). “Some connections between nonuniform and uniform complexity classes”. In: *STOC*. URL: https://dl.acm.org/doi/10.1145/800141.804678.

Kempe, André, Jean-Marc Champarnaud, and Jason Eisner (2004). "A note on join and auto-intersection of n-ary rational relations". In: *Proc. Eindhoven FASTAR Days*. 04-40. Citeseer, pp. 64–78.

Khalifa, Muhammad, Hady Elsahar, and Marc Dymetman (2021). "A Distributional Approach to Controlled Text Generation". In: *ICLR*. URL: https://openreview.net/forum?id=jWkw45-9AbL.

Khandelwal, Urvashi, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis (2020). "Generalization through Memorization: Nearest Neighbor Language Models". In: *ICLR*. URL: https://arxiv.org/abs/1911.00172.

Kim, Yoon (2021). "Sequence-to-Sequence Learning with Latent Neural Grammars". In: *NeurIPS*.

Kingma, Diederik P. and Max Welling (2014). "Auto-Encoding Variational Bayes". In: *ICLR*. URL: http://arxiv.org/abs/1312.6114v10.

Klaas, Mike, Mark Briers, Nando de Freitas, Arnaud Doucet, Simon Maskell, and Dustin Lang (2006). "Fast Particle Smoothing: If I Had a Million Particles". In: *ICML*. URL: http://www.cs.ubc.ca/~nando/papers/fasttwo.pdf.

Knudsen, Bjarne and Michael M. Miyamoto (2003). "Sequence Alignments and Pair Hidden Markov Models Using Evolutionary History". In: *Journal of Molecular Biology* 333.2, pp. 453 –460. URL: http://www.sciencedirect.com/science/article/pii/S0022283603009987.

Kong, Lingpeng, Chris Dyer, and Noah A. Smith (2016). “Segmental Recurrent Neural Networks”. In: *ICLR*.

Krishnan, Rahul G., Uri Shalit, and David Sontag (2017). “Structured Inference Networks for Nonlinear State Space Models”. In: *AAAI*. URL: https://arxiv.org/abs/1609.09869.

Kschischang, Frank R, Brendan J Frey, and H-A Loeliger (2001). “Factor graphs and the sum-product algorithm”. In: *IEEE Transactions on information theory* 47.2, pp. 498–519.

Kuhn, T., H. Niemann, and E.G. Schukat-Talamazzini (1994). “Ergodic hidden Markov models and polygrams for language modeling”. In: *ICASSP*.

Kuich, Werner and Arto Salomaa (2012). *Semirings, automata, languages*. Vol. 5. Springer Science & Business Media.

Kwisthout, Johan H. P., Hans L. Bodlaender, and L. C. van der Gaag (2010). “The Necessity of Bounded Treewidth for Efficient Inference in Bayesian Networks”. In: *ECAI*. URL: http://www.cs.ru.nl/~johank/ECAI-623.pdf.

Ladner, Richard E. (Jan. 1975). “The Circuit Value Problem is log Space Complete for P”. In: *SIGACT News* 7.1, 18–20. URL: https://doi.org/10.1145/990518.990519.

Lafferty, John D., Andrew McCallum, and Fernando Pereira (2001). “Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data”. In: *ICML*.

Lathrop, Richard H. (1996). “On the Learnability of the Uncomputable”. In: *ICML*.

LeCun, Yann, Sumit Chopra, Raia Hadsell, Marc’Aurelio Ranzato, and Fu-Jie Huang (2006). “A Tutorial on Energy-Based Learning”. In: *Predicting Structured Data*. Ed. by G. Bakir,

T. Hofman, B. Schölkopf, A. Smola, and B. Taskar. MIT Press. URL: http://yann.lecun.com/exdb/publis/pdf/lecun-06.pdf.

Lee, Jason, Elman Mansimov, and Kyunghyun Cho (2018). "Deterministic Non-Autoregressive Neural Sequence Modeling by Iterative Refinement". In: *EMNLP*.

Lehmann, Erich L and George Casella (2006). *Theory of point estimation*. Springer. URL: https://link.springer.com/book/10.1007%2Fb98854.

Levin, Leonid A. (1986). "Average Case Complete Problems". In: *SIAM Journal on Computing* 15, pp. 285–286. URL: https://epubs.siam.org/doi/10.1137/0215020.

Lewis, Patrick, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. (2020). "Retrieval-augmented generation for knowledge-intensive NLP tasks". In: *Advances in Neural Information Processing Systems*.

Li, Zhifei and Jason Eisner (2009). "First- and Second-Order Expectation Semirings with Applications to Minimum-Risk Training on Translation Forests". In: *EMNLP*. URL: https://aclanthology.org/D09-1005.

Libovickỳ, Jindřich and Alexander Fraser (2021). "Neural String Edit Distance". In: *arXiv preprint arXiv:2104.08388*.

Lienart, Thibaut, Yee Whye Teh, and Arnaud Doucet (2015). "Expectation Particle Belief Propagation". In: *NIPS*. URL: http://papers.nips.cc/paper/5674-expectation-particle-belief-propagation.pdf.

Lin, Chu-Cheng and Jason Eisner (June 2018). "Neural Particle Smoothing for Sampling from Conditional Sequence Models". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*. New Orleans, pp. 929–941. URL: http://cs.jhu.edu/~jason/papers/#lin-eisner-2018-naacl.

Lin, Chu-Cheng, Aaron Jaech, Xin Li, Matthew R. Gormley, and Jason Eisner (2021). "Limitations of Autoregressive Models and Their Alternatives". In: *NAACL*. URL: https://aclanthology.org/2021.naacl-main.405.

Lin, Chu-Cheng and Arya D. McCarthy (2022). "On the Uncomputability of Partition Functions in Energy-Based Sequence Models". In: *ICLR*. URL: https://openreview.net/forum?id=SsPCtEY6yCl.

Lin, Chu-Cheng, Hao Zhu, Matthew R. Gormley, and Jason Eisner (2019). "Neural Finite-State Transducers: Beyond Rational Relations". In: *NAACL*. URL: https://www.aclweb.org/anthology/N19-1024.

Little, Roderick J. A. and Donald B. Rubin (1987). *Statistical Analysis with Missing Data*. New York: J. Wiley & Sons. URL: https://www.wiley.com/en-us/Statistical+Analysis+with+Missing+Data.

Luby, Michael and Eric Vigoda (1999). "Fast convergence of the Glauber dynamics for sampling independent sets". In: *Random Structures & Algorithms* 15.3-4, pp. 229–241.

Ma, Zhuang and Michael Collins (2018). "Noise Contrastive Estimation and Negative Sampling for Conditional Models: Consistency and Statistical Efficiency". In: *EMNLP*. URL: https://www.aclweb.org/anthology/D18-1405.pdf.

Maddison, Chris J., Andriy Mnih, and Yee Whye Teh (2017). "The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables". In: *ICLR*. URL: https://arxiv.org/abs/1611.00712.

Mahaney, Stephen R. (1982). "Sparse complete sets for NP: Solution of a conjecture of Berman and Hartmanis". In: *Journal of Computer and System Sciences* 25.2, pp. 130–143. URL: https://www.sciencedirect.com/science/article/pii/0022000082900022.

Marcus, Mitchell P., Mary Ann Marcinkiewicz, and Beatrice Santorini (June 1993). "Building a Large Annotated Corpus of English: The Penn Treebank". In: *Computational Linguistics* 19.2, pp. 313–330. URL: http://dl.acm.org/citation.cfm?id=972470.972475.

Martín, Ernesto San and Fernando Quintana (2002). "Consistency and identifiability revisited". In: *Brazilian Journal of Probability and Statistics*, pp. 99–106.

McAllester, David A., Michael Collins, and Fernando C Pereira (2004). "Case-factor diagrams for structured probabilistic modeling". In: *J. Comput. Syst. Sci.* 74, pp. 84–96.

McCallum, Andrew, Dayne Freitag, and Fernando Pereira (2000). "Maximum Entropy Markov Models for Information Extraction and Segmentation". In: *ICML*. URL: http://www.cs.cmu.edu/~mccallum/papers/memm-icml2000.pdf.

Merity, Stephen, Caiming Xiong, James Bradbury, and Richard Socher (2017). "Pointer Sentinel Mixture Models". In: *ArXiv* abs/1609.07843. URL: https://arxiv.org/abs/1609.07843.

Merrill, William, Gail Weiss, Yoav Goldberg, Roy Schwartz, Noah A. Smith, and Eran Yahav (2020). "A Formal Hierarchy of RNN Architectures". In: *ACL*. URL: https://aclanthology.org/2020.acl-main.43.

Mikolov, Tomas, M. Karafiát, L. Burget, J. Cernocký, and S. Khudanpur (2010). "Recurrent neural network based language model". In: *INTERSPEECH*.

Mnih, Andriy and Karol Gregor (2014). "Neural variational inference and learning in belief networks". In: *ICML*. URL: http://proceedings.mlr.press/v32/mnih14.html.

Mohri, Mehryar (2009). "Weighted Automata Algorithms". In: *Handbook of Weighted Automata*. Springer, pp. 213–254.

Mohri, Mehryar, Fernando Pereira, and Michael Riley (2008). "Speech recognition with weighted finite-state transducers". In: *Springer Handbook of Speech Processing*. Springer, pp. 559–584.

Nishii, Ryuei (1984). "Asymptotic Properties of Criteria for Selection of Variables in Multiple Regression". In: *Annals of Statistics* 12, pp. 758–765.

Nishii, Ryuei (1988). "Maximum likelihood principle and model selection when the true model is unspecified". In: *Journal of Multivariate Analysis* 27, pp. 392–403.

Oord, Aaron van den, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu (2016). "WaveNet:

A Generative Model for Raw Audio". In: *ArXiv* abs/1609.03499. URL: https://arxiv.org/abs/1609.03499.

Owen, Art B. (2013). *Monte Carlo theory, methods and examples.*

Owen, Art B. and Yi Zhou (2000). "Safe and Effective Importance Sampling". In: *Journal of the American Statistical Association* 95, pp. 135–143.

Paige, Brooks and Frank D. Wood (2016). "Inference Networks for Sequential Monte Carlo in Graphical Models". In: *ICML*. URL: https://arxiv.org/abs/1602.06701.

Paige, Brooks, Frank D. Wood, Arnaud Doucet, and Yee Whye Teh (2014). "Asynchronous Anytime Sequential Monte Carlo". In: *NIPS*. URL: http://proceedings.mlr.press/v48/paige16.pdf.

Peng, Hao, Roy Schwartz, Sam Thomson, and Noah A. Smith (2018). "Rational Recurrences". In: *EMNLP*. URL: https://aclanthology.org/D18-1152.

Peng, Nanyun and Mark Dredze (2015). "Named Entity Recognition for Chinese Social Media with Jointly Trained Embeddings". In: *EMNLP*. URL: http://www.aclweb.org/anthology/D15-1064.

Pennington, Jeffrey, Richard Socher, and Christopher D. Manning (2014). "GloVe: Global Vectors for Word Representation". In: *EMNLP*. URL: http://www.aclweb.org/anthology/D14-1162.

Pereira, Fernando C. N. and Michael D. Riley (1997). "Speech Recognition by Composition of Weighted Finite Automata". In: *Finite-State Language Processing*, p. 431. URL: http://www.aclweb.org/anthology/D15-1064.

Pérez, Jorge, Pablo Barceló, and Javier Marinkovic (2021). "Attention is Turing-Complete". In: *Journal of Machine Learning Research* 22.75, pp. 1–35. URL: http://jmlr.org/papers/v22/20-302.html.

Piperski, Alexander (2016). *The CMU Pronouncing Dictionary converted to IPA*. URL: https://github.com/menelik3/cmudict-ipa.

Post, Matt, Shuoyang Ding, Marianna Martindale, and Winston Wu (Aug. 2019). "An Exploration of Placeholding in Neural Machine Translation". In: *Proceedings of Machine Translation Summit XVII: Research Track*. Dublin, Ireland: European Association for Machine Translation, pp. 182–192. URL: https://aclanthology.org/W19-6618.

Post, Matt and David Vilar (2018). "Fast Lexically Constrained Decoding with Dynamic Beam Allocation for Neural Machine Translation". In: *NAACL*. URL: https://aclanthology.org/N18-1119.

Rabiner, Lawrence R. (1989). "A Tutorial On Hidden Markov Models And Selected Applications In Speech Recognition". In: *Proceedings of IEEE* 77.2, pp. 257–285. URL: http://www.ece.ucsb.edu/Faculty/Rabiner/ece259/Reprints/tutorial%20on%20hmm%20and%20applications.pdf.

Radford, Alec, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever (2019). "Language Models are Unsupervised Multitask Learners". In: URL: https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf.

Ramshaw, Lance A. and Mitchell P. Marcus (1999). "Text chunking using transformation-based learning". In: *Natural Language Processing Using Very Large Corpora*. Springer, pp. 157–176. URL: https://www.aclweb.org/anthology/W95-0107.

Rastogi, Pushpendre, Ryan Cotterell, and Jason Eisner (June 2016). "Weighting Finite-State Transductions With Neural Context". In: *NAACL*. URL: https://www.aclweb.org/anthology/N16-1076.

Raymond, Christian and Giuseppe Riccardi (2007). "Generative and discriminative algorithms for spoken language understanding". In: *Interspeech*.

Ristic, Branko, Sanjeev Arulampalam, and Neil James Gordon (2004). *Beyond the Kalman Filter: Particle Filters for Tracking Applications*. Artech House.

Roark, Brian, Richard Sproat, and Izhak Shafran (2011). "Lexicographic Semirings for Exact Automata Encoding of Sequence Models". In: *ACL*. URL: https://aclanthology.org/P11-2001.

Roark, Brian, Lawrence Wolf-Sonkin, Christo Kirov, Sabrina J. Mielke, Cibu Johny, Isin Demirsahin, and Keith Hall (2020). "Processing South Asian Languages Written in the Latin Script: the Dakshina Dataset". English. In: *LREC*. URL: https://aclanthology.org/2020.lrec-1.294.

Robbins, Herbert and Sutton Monro (1951). "A Stochastic Approximation Method". In: *The Annals of Mathematical Statistics*, pp. 400–407. URL: https://www.jstor.org/stable/2236626.

Roche, Emmanuel and Yves Schabes, eds. (1997). *Finite-State Language Processing*. MIT Press. URL: https://mitpress.mit.edu/books/finite-state-language-processing.

Rosenfeld, Ronald, Stanley Chen, and Xiaojin Zhu (Jan. 2001). "Whole-Sentence Exponential Language Models: A Vehicle for Linguistic-Statistical Integration". In: *Computer Speech & Language* 15.1, pp. 55–73. URL: https://doi.org/10.1006/csla.2000.0159.

Roth, Dan (Apr. 1996). "On the Hardness of Approximate Reasoning". In: *Artificial Intelligence* 82.1–2, 273–302. URL: https://doi.org/10.1016/0004-3702(94)00092-1.

Saharia, Chitwan, William Chan, Saurabh Saxena, and Mohammad Norouzi (2020). "Non-Autoregressive Machine Translation with Latent Alignments". In: *EMNLP*.

Salomaa, Arto and Matti Soittola (2012). *Automata-theoretic aspects of formal power series*. Springer Science & Business Media.

Sandbank, Ben (2008). "Refining generative language models using discriminative learning". In: *EMNLP*. URL: https://www.aclweb.org/anthology/D08-1006.pdf.

Schwartz, Roy, Sam Thomson, and Noah A. Smith (2018). "Bridging CNNs, RNNs, and Weighted Finite-State Machines". In: *ACL*. URL: https://aclanthology.org/P18-1028.

Serdyuk, Dmitriy, Nan Rosemary Ke, Alessandro Sordoni, Adam Trischler, Chris Pal, and Yoshua Bengio (2018). "Twin Networks: Matching the Future for Sequence Generation". In: *ICLR*. URL: https://openreview.net/forum?id=BydLzGb0Z.

Shen, Sheng, Daniel Fried, Jacob Andreas, and Dan Klein (2019). "Pragmatically Informative Text Generation". In: *NAACL*. URL: https://aclanthology.org/N19-1410.

Shi, Haoyue, Hao Zhou, Jiaze Chen, and Lei Li (2018a). "On Tree-Based Neural Sentence Modeling". In: *EMNLP*. URL: https://aclanthology.org/D18-1492.

Shi, Tian, Yaser Keneshloo, Naren Ramakrishnan, and Chandan K. Reddy (2018b). "Neural Abstractive Text Summarization with Sequence-to-Sequence Models". In: *ArXiv* abs/1812.02303.

Shwartz, Vered, Peter West, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi (Nov. 2020). "Unsupervised Commonsense Question Answering with Self-Talk". In: *EMNLP*. URL: https://www.aclweb.org/anthology/2020.emnlp-main.373.

Siegelmann, Hava T. and Eduardo D. Sontag (1992). "On the computational power of neural nets". In: *COLT*. URL: https://dl.acm.org/doi/10.1145/130385.130432.

Siegelmann, H.T. and E.D. Sontag (1995). "On the Computational Power of Neural Nets". In: *Journal of Computer and System Sciences* 50.1, pp. 132–150. URL: https://www.sciencedirect.com/science/article/pii/S0022000085710136.

Sipser, Michael (2013). *Introduction to the Theory of Computation*. Third. Boston, MA: Course Technology.

Smith, Noah A. and Jason Eisner (2005). "Contrastive Estimation: Training Log-Linear Models on Unlabeled Data". In: *ACL*. URL: https://aclanthology.org/P05-1044.

Sproat, Richard, Mahsa Yarmohammadi, Izhak Shafran, and Brian Roark (Dec. 2014). "Applications of Lexicographic Semirings to Problems in Speech and Language Processing". In:

Computational Linguistics 40.4, pp. 733–761. URL: https://aclanthology.org/J14-4002.

Stroock, Daniel W. (2014). *An Introduction to Markov Processes*. 2nd ed. Vol. 230. Graduate Texts in Mathematics. Heidelberg: Springer.

Stuhlmüller, Andreas, Jacob Taylor, and Noah Goodman (2013). "Learning Stochastic Inverses". In: *NIPS*. URL: http://papers.nips.cc/paper/4966-learning-stochastic-inverses.pdf.

Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le (2014). "Sequence to Sequence Learning with Neural Networks". In: *NIPS*.

Sutton, Charles and Andrew McCallum (2012). "An Introduction to Conditional Random Fields". In: *Foundations and Trends in Machine Learning* 4, pp. 267–373.

Sutton, Richard (2019). "The bitter lesson". In: *Incomplete Ideas (blog)* 13, p. 12.

Thrun, Sebastian (2000). "Monte Carlo POMDPs". In: *NIPS*. URL: https://papers.nips.cc/paper/1772-monte-carlo-pomdps.pdf.

Turing, A. M. (Jan. 1937). "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2–42.1, pp. 230–265. eprint: https://academic.oup.com/plms/article-pdf/s2-42/1/230/4317544/s2-42-1-230.pdf. URL: https://doi.org/10.1112/plms/s2-42.1.230.

Valiant, L. G. (Nov. 1984). "A Theory of the Learnable". In: *Commun. ACM* 27.11, 1134–1142. URL: https://doi.org/10.1145/1968.1972.

Veldhuizen, Todd L. (2003). *C++ Templates are Turing Complete*. Tech. rep.

Venezky, Richard L (2011). *The structure of English orthography*. Vol. 82. Walter de Gruyter. URL: https://www.degruyter.com/document/doi/10.1515/9783110804478/html.

Vinyals, Oriol, Lukasz Kaiser, Terry K Koo, Slav Petrov, Ilya Sutskever, and Geoffrey E. Hinton (2015). "Grammar as a Foreign Language". In: *NIPS*.

Viterbi, A. (1967). "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm". In: *IEEE Trans. Inf. Theory* 13, pp. 260–269.

Wang, Po-Wei, P. Donti, B. Wilder, and J. Z. Kolter (2019). "SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver". In: *ICML*. URL: https://arxiv.org/abs/1905.12149.

Wansbrough, Keith (1998). *Instance Declarations are Universal*. URL: https://www.lochan.org/keith/publications/undec.html.

Wei, Greg C. G. and Martin A. Tanner (1990). "A Monte Carlo Implementation of the EM Algorithm and the Poor Man's Data Augmentation Algorithms". In: *Journal of the American Statistical Association* 85.411, pp. 699–704. URL: http://www.jstor.org/stable/2290005.

Wei, Jason, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou (2022). *Chain of Thought Prompting Elicits Reasoning in Large Language Models*. URL: https://arxiv.org/abs/2201.11903.

Weide, Robert (2005). *The Carnegie Mellon Pronouncing Dictionary (CMUDict 0.6)*. URL: http://www.speech.cs.cmu.edu/cgi-bin/cmudict.

Welleck, Sean, Ilia Kulikov, Jaedeok Kim, Richard Yuanzhe Pang, and Kyunghyun Cho (Nov. 2020). "Consistency of a Recurrent Language Model With Respect to Incomplete Decoding". In: *EMNLP*. URL: https://www.aclweb.org/anthology/2020.emnlp-main.448.

Williams, Ronald J. (1992). "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning* 8.23. URL: https://dl.acm.org/citation.cfm?id=139614.

Wiseman, Sam and Alexander M. Rush (2016). "Sequence-to-Sequence Learning as Beam-Search Optimization". In: *EMNLP*. URL: https://aclweb.org/anthology/D16-1137.

Wu, Di, Liang Ding, Fan Lu, and Jian Xie (2020). "SlotRefine: A Fast Non-Autoregressive Model for Joint Intent Detection and Slot Filling". In: *EMNLP*. URL: https://aclanthology.org/2020.emnlp-main.152.

Wu, Ke, Cyril Allauzen, Keith B. Hall, Michael Riley, and Brian Roark (2014). "Encoding linear models as weighted finite-state transducers". In: *INTERSPEECH*.

Wu, Shijie, Pamela Shapiro, and Ryan Cotterell (2018). "Hard Non-Monotonic Attention for Character-Level Transduction". In: *EMNLP*. URL: https://www.aclweb.org/anthology/D18-1473/.

Yamada, Hiroyasu and Yuji Matsumoto (2003). "Statistical dependency analysis with support vector machines". In: *Proceedings of IWPT*. Vol. 3, pp. 195–206. URL: http://www.jaist.jp/~h-yamada/pdf/iwpt2003.pdf.

Yao, Kaisheng, Geoffrey Zweig, Mei-Yuh Hwang, Yangyang Shi, and Dong Yu (2013). "Recurrent neural networks for language understanding". In: *INTERSPEECH*.

Yedidia, Adam B. and S. Aaronson (2016). "A Relatively Small Turing Machine Whose Behavior Is Independent of Set Theory". In: *ArXiv* abs/1605.04343.

Yelp. *Yelp Open Dataset*. https://www.yelp.com/dataset.

Yogatama, Dani, Cyprien de Masson d'Autume, and Lingpeng Kong (2021). "Adaptive Semiparametric Language Models". In: *ArXiv* abs/2102.02557. URL: https://arxiv.org/abs/2102.02557.

Zellers, Rowan, Ari Holtzman, Hannah Rashkin, Yonatan Bisk, Ali Farhadi, Franziska Roesner, and Yejin Choi (2019). "Defending Against Neural Fake News". In: *NeurIPS*. URL: http://papers.nips.cc/paper/9106-defending-against-neural-fake-news.pdf.

Zibulevsky, Michael and Barak A. Pearlmutter (2001). "Blind source separation by sparse decomposition in a signal dictionary". In: *Neural Computation* 13.4, pp. 863–882. URL: https://ieeexplore.ieee.org/document/6790205.

Vita

Chu-Cheng Lin graduated with a B. Sc. in Computer Science from National Taiwan University, Taiwan. He also obtained an M. S.;from Carnegie Mellon University, Pennsylvania. His research focuses on sequence modeling. During his Ph. D., he interned at Microsoft Research, Facebook AI, and Amazon Alexa. In 2022, he joins Google as a software engineer.