
THE

Formal Aspects of Language Modeling

Ryan Cotterell, Anej Svete, Luca Malagutti, Clara Meister,
Tianyu Liu, Vilém Zouhar and Li Du

Thursday 23rd March, 2023

Contents

Chapter 1

Introduction

1.1 Introduction

Welcome to the class notes for the first half of Large Language Models (263-5354-00L). The course comprises an omnibus introduction to language modeling. The first half of the lectures focuses on a formal treatment of the subject. The second half focuses on the practical aspects of implementing a language model and its applications. Many universities are offering similar courses at the moment, e.g., CS324 at Stanford University (https://stanford-cs324.github.io/winter2022/) and CS 600.471 (https://self-supervised.cs.jhu.edu/sp2023/) at Johns Hopkins University. Their syllabi may serve as useful references.

Chapter 2

Probabilistic Foundations

2.1 An Invitation to Language Modeling

The first module of the course focuses on *defining* a language model mathematically. To see why such a definition is nuanced, we are going to give an informal definition of a language model and demonstrate two ways in which that definition breaks and fails to meet our desired criteria.

Definition 2.1.1 (Informal)**.** *Given an alphabet*[1] Σ *and a distinguished end-of-sequence symbol* EOS $\notin \Sigma$*, a language model is a collection of conditional probability distributions* $p(y \mid \boldsymbol{y})$ *for* $y \in \Sigma \cup \{\text{EOS}\}$ *and* $\boldsymbol{y} \in \Sigma^*$*, the Kleene closure of* Σ*.*[2] $p(y \mid \boldsymbol{y})$ *therefore represents the probability of* y *being the next token given the history* $\boldsymbol{y}$*.*

Definition 2.1.1 is the definition of a language model that is implicitly assumed in most papers on language modeling. We say implicitly since most technical papers on language modeling simply write down the following autoregressive factorization

$$p(\boldsymbol{y}) = p(y_1 \cdots y_T) = p(\text{EOS} \mid \boldsymbol{y}) \prod_{t=1}^{T} p(y_t \mid \boldsymbol{y}_{<t}) \tag{2.1}$$

as the probability of a string according to the distribution p.[3] The part that is left implicit in Eq. (2.1) is whether or not p is indeed a probability distribution and, if it is, over what space. The natural assumption in Definition 2.1.1 is that p is a distribution over Σ^*, i.e., the set of all *finite* strings[4] over an alphabet Σ. However, in general, it is not true that all such collections of conditionals will yield a valid probability distribution over Σ^*; some may "leak" probability

[1] An alphabet is a *finite*, non-empty set.
[2] We will define the Kleene closure in more detail later in the course.
[3] Many authors (erroneously) avoid writing EOS for concision.
[4] Some authors assert that strings are by definition finite.

mass to infinite sequences.[5] More subtly, we additionally have to be very careful when dealing with uncountably infinite spaces lest we run into a classic paradox, which we will demonstrate below.

We highlight the two issues above with two very simple examples. The first example is a well-known paradox in probability theory.

Example 2.1.1 (Infinite Coin Toss)**.** *Consider the infinite independent fair coin toss model, where we aim to place a distribution over* $\{\mathsf{H}, \mathsf{T}\}^\infty$*, the (uncountable) set of infinite sequences of* $\{\mathsf{H}, \mathsf{T}\}$ *(*H *represents the event of throwing heads and* T *the event of throwing tails). Intuitively, such a distribution corresponds to a "language model" as defined above in which for all* $\boldsymbol{y}_{<t}$*,* $p(\mathsf{H} \mid \boldsymbol{y}_{<t}) = p(\mathsf{T} \mid \boldsymbol{y}_{<t}) = \frac{1}{2}$ *and* $p(\text{EOS} \mid \boldsymbol{y}_{<t}) = 0$*. However, each individual infinite sequence over* $\{\mathsf{H}, \mathsf{T}\}$ *should also be assigned probability* $(\frac{1}{2})^\infty = 0$*. Without a formal foundation, one arrives at the following paradox:*

$$
\begin{aligned}
1 = p\left(\{\mathsf{H}, \mathsf{T}\}^\infty\right) &= p\left(\bigcup_{\boldsymbol{\omega} \in \{\mathsf{H}, \mathsf{T}\}^\infty} \{\boldsymbol{\omega}\}\right) \\
&= \sum_{\boldsymbol{\omega} \in \{\mathsf{H}, \mathsf{T}\}^\infty} p(\{\boldsymbol{\omega}\}) = \sum_{\boldsymbol{\omega} \in \{\mathsf{H}, \mathsf{T}\}^\infty} 0 \stackrel{?}{=} 0.
\end{aligned}
$$

The second example is more specific to language modeling. As we stated above, an implicit assumption made by most language modeling papers is that a language model constitutes a distribution over Σ^*. However, in our next example, we show that a collection of conditions that satisfy Definition 2.1.1 may not sum to 1 if the sum is restricted to elements of Σ^*. This means that it is not a-priori clear what space our probability distribution is defined over.[6]

Example 2.1.2 (Possibly Finite Coin Toss)**.** *Consider now the possibly finite "coin toss" model with a rather peculiar coin: when tossing the coin for the first time, both* H *and* T *are equally likely. After the first toss, however, the coin gets stuck: If* $y_1 = \mathsf{H}$*, we can only ever toss another* H *again, whereas if* $y_1 = \mathsf{T}$*, the next toss can result in another* T *or "end" the sequence of throws (*EOS*) with equal probability. We, therefore, model a probability distribution over* $\{\mathsf{H}, \mathsf{T}\}^* \cup \{\mathsf{H}, \mathsf{T}\}^\infty$*,*

[5] However, the converse *is* true: All valid distributions over Σ^* may be factorized as the above.

[6] This also holds true for the first example.

the set of finite and infinite sequences of tosses. Formally:[7]

$$
\begin{aligned}
p(\mathsf{H} \mid \boldsymbol{y}_{<1}) = p(\mathsf{T} \mid \boldsymbol{y}_{<1}) &= \frac{1}{2} \\
p(\mathsf{H} \mid \boldsymbol{y}_{<t}) &= \begin{cases} 1 & \text{if } t > 1 \text{ and } y_{t-1} = \mathsf{H} \\ 0 & \text{if } t > 1 \text{ and } y_{t-1} = \mathsf{T} \end{cases} \\
p(\mathsf{T} \mid \boldsymbol{y}_{<t}) &= \begin{cases} \frac{1}{2} & \text{if } t > 1 \text{ and } y_{t-1} = \mathsf{T} \\ 0 & \text{if } t > 1 \text{ and } y_{t-1} = \mathsf{H} \end{cases} \\
p(\text{EOS} \mid \boldsymbol{y}_{<t}) &= \begin{cases} \frac{1}{2} & \text{if } t > 1 \text{ and } y_{t-1} = \mathsf{T} \\ 0 & \text{otherwise.} \end{cases}
\end{aligned}
$$

If you are familiar with (weighted) finite-state automata,[8] *you can imagine the model as depicted in Fig. 2.1.*

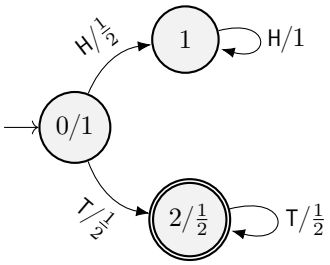


Figure 2.1: Graphical depiction of the possibly finite coin toss model. The final weight $\frac{1}{2}$ of the state 2 corresponds to the probability $p\,(\text{EOS} \mid y_{t-1} = \mathsf{T}) = \frac{1}{2}$.

It is easy to see that this model only places the probability of $\frac{1}{2}$ *on* finite *sequences of tosses. If we were only interested in those (analogously to how we are only interested in finite strings when modeling language), yet still allowed the model to specify the probabilities as in this example, the resulting probability distribution would not model what we require.*

It takes some mathematical heft to define a language model in a manner that avoids such paradoxes. The tool of choice for mathematicians is measure theory, as it allows us to define probability over uncountable sets[9] in a principled way. Thus, we begin our formal treatment of language modeling with a primer of measure theory in §2.2. Then, we will use concepts discussed in the primer to work up to a formal definition of a language model.

[7] Note that $p(\mathsf{H} \mid \boldsymbol{y}_{<1}) = p(\mathsf{H} \mid \varepsilon)$ and $p(\mathsf{T} \mid \boldsymbol{y}_{<1}) = p(\mathsf{T} \mid \varepsilon)$.

[8] They will be formally introduced in §4.1.5

[9] As stated earlier, $\{\mathsf{H}, \mathsf{T}\}^\infty$ is uncountable. It's easy to see there exists a surjection from $\{\mathsf{H}, \mathsf{T}\}^\infty$ to the binary expansion of the real interval $(0, 1]$. Readers who are interested in more details and mathematical implications can refer to §1 in Billingsley (1995).

2.2 A Measure-theoretic Foundation

At their core, (large) language models are an attempt to place a probabilistic distribution over natural language utterances. However, our toy examples in Examples 2.1.1 and 2.1.2 in the previous section reveal that it is quite nuanced to get a satisfying definition of a language model. Thus, our first step forward is to review the basics of rigorous probability theory,[10] the tools we need to come to a satisfying definition. Our course will assume that you have had some exposure to rigorous probability theory before, and just review the basics. However, it is also possible to learn the basics of rigorous probability on the fly during the course if it is new to you. Specifically, we will cover *measure-theoretic* foundations of probability theory. This might come as a bit of a surprise since we are mostly going to be talking about discrete objects—namely, strings. However, as we will see in §2.5 soon, formal treatment of language modeling indeed requires some mathematical rigor from measure theory.

The goal of measure-theoretic probability is to assign probabilities to *subsets* of an **outcome space** Ω. However, in the course of the study of measure theory, it has become clear that for many common Ω, it is impossible to assign probabilities in a way that satisfies a set of reasonable desiderata.[11] Consequently, the standard approach to probability theory resorts to only assigning probability to certain "nice" (but not all) subsets of Ω, which are referred to as **events** or **measurable subsets**, as in the theory of integration or functional analysis. The set of measurable subsets is commonly denoted as $\mathcal{F}$ (Definition 2.2.1) and a probability measure $\mathbb{P} : \mathcal{F} \to [0, 1]$ is the function that assigns a probability to each measurable subset. The triple $(\Omega, \mathcal{F}, \mathbb{P})$ is collectively known as a probability space (Definition 2.2.2). As it turns out, the following simple and reasonable requirements imposed on $\mathcal{F}$ and $\mathbb{P}$ are enough to rigorously discuss probability.

Definition 2.2.1. *Let $\mathcal{P}(\Omega)$ be the power set of Ω. Then $\mathcal{F} \subseteq \mathcal{P}(\Omega)$ is called a* ***σ-algebra*** *(or σ-field) over Ω if the following conditions hold:*

1) $\Omega \in \mathcal{F}$,

2) if $\mathcal{E} \in \mathcal{F}$, then $\mathcal{E}^c \in \mathcal{F}$,

3) if $\mathcal{E}_1, \mathcal{E}_2, \ldots$ is a finite or infinite sequence of sets in $\mathcal{F}$, then $\bigcup_n \mathcal{E}_n \in \mathcal{F}$.

If $\mathcal{F}$ is a σ-algebra over Ω, we call the tuple $(\Omega, \mathcal{F})$ a ***measurable space****.*

Example 2.2.1 (σ-algebras)**.** *Let Ω be any set. Importantly, there is no one way to construct a σ-algebra over Ω:*

1. *The family consisting of only the empty set $\varnothing$ and the set Ω, i.e., $\mathcal{F} \stackrel{\text{def}}{=} \{\varnothing, \Omega\}$, is called the* minimal *or* trivial*.*

[10] By rigorous probability theory we mean a measure-theoretic treatment of probability theory.

[11] Measure theory texts commonly discuss such desiderata and the dilemma that comes with it. See, e.g., Chapter 7 in Tao (2016), Chapter 3 in Royden (1988) or Chapter 3 in Billingsley (1995). We also give an example later.

2. *The full power set* $\mathcal{F} \overset{\text{def}}{=} \mathcal{P}(\Omega)$ *is called the* discrete σ-algebra.

3. *Given* $\mathcal{A} \subseteq \Omega$*, the family* $\mathcal{F} \overset{\text{def}}{=} \{\varnothing, \mathcal{A}, \Omega \backslash \mathcal{A}, \Omega\}$ *is a* σ*-algebra induced by* $\mathcal{A}$.

4. *Suppose we are rolling a six-sided die. There are six events that can happen: we can roll any of the numbers* 1–6*. In this case, we will then define the set of outcomes* Ω *as* $\Omega \overset{\text{def}}{=} \{\textit{The number observed is } n \mid n = 1, \ldots, 6\}$*. There are of course multiple ways to define an event space* $\mathcal{F}$ *and with it a* σ*-algebra over this sample space. By definition,* $\varnothing \in \mathcal{F}$ *and* $\Omega \in \mathcal{F}$*. One way to intuitively construct a* σ*-algebra is to consider that all individual events (observing any number) are possible, meaning that we would like to later assign probabilities to them (see Definition 2.2.2). This means that we should include individual singleton events in the event space:* $\{\textit{The number observed is } n\} \in \mathcal{F}$ *for* $n = 1, \ldots, 6$*. It is easy to see that in this case, to satisfy the axioms in Definition 2.2.1, the resulting event space should be* $\mathcal{F} = \mathcal{P}(\Omega)$.

You might want to confirm these are indeed σ*-algebras by checking them against the axioms in Definition 2.2.1.*

A measurable space guarantees that operations on countably many sets are always valid, and hence permits the following definition.

Definition 2.2.2. *A* ***probability measure*** $\mathbb{P}$ *over a measurable space* $(\Omega, \mathcal{F})$ *is a function* $\mathbb{P} : \mathcal{F} \to [0, 1]$ *such that*

1) $\mathbb{P}(\Omega) = 1$,

2) if $\mathcal{E}_1, \mathcal{E}_2, \ldots$ *is a countable sequence of disjoint sets in* $\mathcal{F}$*, then* $\mathbb{P}(\bigcup_n \mathcal{E}_n) = \sum_n \mathbb{P}(\mathcal{E}_n)$.

In this case we call $(\Omega, \mathcal{F}, \mathbb{P})$ *a* ***probability space****.*

As mentioned, measure-theoretic probability only assigns probabilities to "nice" subsets of Ω. In fact, it is often impossible to assign a probability measure to every single subset of Ω and we must restrict our probability space to a strict subset of $\mathcal{P}(\Omega)$. More precisely, the sets $\mathcal{B} \subseteq \Omega$ for which a probability (or more generally, a *volume*) can not be defined are called *non-measurable sets*. An example of under certain assumptions such sets is the Vitali set[12] See also Appendix A.2 in Durrett (2019).

Later, we will be interested in modeling probability spaces over sets of (infinite) sequences. By virtue of a theorem due to Carathéodory, there is a natural way to construct such a probability space for sequences (and many other spaces) that behaves in accordance with our intuition, as we will clarify later. Here, we shall lay out a few other necessary definitions.

Definition 2.2.3. $\mathcal{A} \subseteq \mathcal{P}(\Omega)$ *is called an* ***algebra*** *(or field) over* Ω *if*

[12] See https://en.wikipedia.org/wiki/Non-measurable_set and https://en.wikipedia.org/wiki/Vitali_set.

1) $\Omega \in \mathcal{A}$,

2) if $\mathcal{E} \in \mathcal{A}$, *then* $\mathcal{E}^c \in \mathcal{A}$,

3) if $\mathcal{E}_1, \mathcal{E}_2 \in \mathcal{A}$, *then* $\mathcal{E}_1 \cup \mathcal{E}_2 \in \mathcal{A}$.

Definition 2.2.4. *Let* $\mathcal{A}$ *be an algebra over some set* Ω. *A* ***probability pre-measure*** *over* $(\Omega, \mathcal{A})$ *is a function* $\mathbb{P}_0 : \mathcal{A} \to [0,1]$ *such that*

1) $\mathbb{P}_0(\Omega) = 1$,

2) if $\mathcal{E}_1, \mathcal{E}_2, \ldots$ *is a (countable) sequence of disjoint sets in* $\mathcal{A}$ whose (countable) union is also in $\mathcal{A}$, *then* $\mathbb{P}_0(\cup_{n=1}^{\infty} \mathcal{E}_n) = \sum_{n=1}^{\infty} \mathbb{P}_0(\mathcal{E}_n)$.

Note that the only difference between a σ-algebra (Definition 2.2.1) and an algebra is that condition 3 is weakened from countable to finite, and the only difference between a probability measure (Definition 2.2.2) and a pre-measure is that the latter is defined with respect to an algebra instead of a σ-algebra.

The idea behind Carathéodory's extension theorem is that there is often a simple construction of an algebra $\mathcal{A}$ over Ω such that there is a natural way to define a probability pre-measure. One can then *extend* this probability pre-measure to a probability measure that is both minimal and unique in a precise sense. For example, the standard Lebesgue measure over the the real line can be constructed this way.

Finally, we define random variables.

Definition 2.2.5 (Random variable)**.** *A mapping* $\mathrm{x} : \Omega \to \mathcal{S}$ *between two measurable spaces* $(\Omega, \mathcal{F})$ *and* $(\mathcal{S}, \mathcal{T})$ *is an* $(\mathcal{S}, \mathcal{T})$*-valued* ***random variable****, or a measurable mapping, if, for all* $\mathcal{B} \in \mathcal{T}$,

$$\mathrm{x}^{-1}(\mathcal{B}) \stackrel{\text{def}}{=} \{\omega \in \Omega : \mathrm{x}(\omega) \in \mathcal{B}\} \in \mathcal{F}. \tag{2.2}$$

Any measurable function (random variable) induces a new probability measure on the *output* σ-algebra based on the one defined on the original σ-algebra. This is called the **pushforward measure** (cf. §2.4 in Tao, 2011), which we will denote by $\mathbb{P}_*$, given by

$$\mathbb{P}_* \left(\mathrm{x} \in \mathcal{E}\right) \stackrel{\text{def}}{=} \mathbb{P}\left(\mathrm{x}^{-1}\left(\mathcal{E}\right)\right), \tag{2.3}$$

that is, the probability of the result of x being in some event $\mathcal{E}$ is determined by the probability of the event of all the elements which x maps into $\mathcal{E}$, i.e., the pre-image of $\mathcal{E}$ given by x.

Example 2.2.2 (Random Variables)**.** *We give some simple examples of random variables.*

1. *Let* Ω *be the set of possible outcomes of throwing a fair coin, i.e.,* $\Omega \stackrel{\text{def}}{=} \{\mathsf{T}, \mathsf{H}\}$. *Define* $\mathcal{F} \stackrel{\text{def}}{=} \mathcal{P}(\Omega)$, $\mathcal{S} \stackrel{\text{def}}{=} \{0, 1\}$, *and* $\mathcal{T} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{S})$. *Then, the random variable*

$$\mathbf{x} : \begin{cases} \mathsf{T} \mapsto 0 \\ \mathsf{H} \mapsto 1 \end{cases}$$

assigns tails (T) *the value* 0 *and heads* (H) *the value* 1.

2. *Consider the probability space of throwing two dice (similar to Example 2.2.1) where* $\Omega = \{(i,j) : i,j = 1,\ldots,6\}$ *where the element* (i,j) *refers to rolling* i *on the first and* j *on the second die and* $\mathcal{F} = \mathcal{P}(\Omega)$*. Define* $\mathcal{S} \stackrel{\text{def}}{=} \mathbb{Z}$ *and* $\mathcal{T} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{S})$*. Then, the random variable*

$$\mathbf{x} : (i,j) \mapsto i + j$$

is an $(\mathcal{S},\mathcal{T})$*-valued random variable which represents the sum of two dice.*

2.3 Language Models: Distributions over Strings

Language models are defined as probability distributions over sequences of words, referred to as utterances. This chapter delves into the formalization of the term "utterance" and introduces fundamental concepts such as the alphabet, string, and language. Utilizing these concepts, a formal definition of a language model is presented, along with a discussion on the intricacies of defining distributions over infinite sets.

2.3.1 Sets of Strings

We begin by defining the very basic notions of alphabets and strings, where we take inspiration from **formal language theory**. First and foremost, formal language theory concerns itself with *sets of structures*. The simplest structure it considers is a **string**. So what is a string? We start with the notion of an alphabet.

Definition 2.3.1 (Alphabet)**.** *An **alphabet** is a finite, non-empty set. In this course, we will denote an alphabet using Greek capital letters, e.g.,* Σ *and* Δ*. We refer to the elements of an alphabet as **symbols** or letters and will denote them with lowercase letters: a, b, c.*

Definition 2.3.2 (String)**.** *A **string**[13] over an alphabet is any* finite *sequence of letters. Strings made up of symbols from* Σ *will denoted by bolded Latin letters, e.g.,* $\boldsymbol{y} = y_1 \cdots y_T$ *where each* $y_n \in \Sigma$*.*

The length of a string, written as $|\boldsymbol{y}|$, is the number of letters it contains. Usually, we will use T to denote $|\boldsymbol{y}|$ more concisely whenever the usage is clear from the context. There is only one string of length zero, which we denote with the distinguished symbol ε and refer to as the *empty string*. By convention, ε is *not* an element of the original alphabet.

New strings are formed from other strings and symbols with **concatenation**. Concatenation, denoted with $\boldsymbol{x} \circ \boldsymbol{y}$ or just $\boldsymbol{xy}$, is an associative operation on strings. Formally, the concatenation of two words $\boldsymbol{y}$ and $\boldsymbol{x}$ is the word $\boldsymbol{y} \circ \boldsymbol{x} = \boldsymbol{yx}$, which is obtained by writing the second argument after the first one. The result of concatenating with ε from either side results in the original string, which means that ε is the **unit** of concatenation and the set of all words over an alphabet with the operation of concatenation forms a **monoid**.

We have so far only defined strings as individual sequences of symbols. To give our strings made up of symbols in Σ a set to live in, we now define Kleene closure of an alphabet Σ.

Definition 2.3.3 (Kleene Star)**.** *Let* Σ *be an alphabet. The **Kleene star** Σ^* is defined as*

$$\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n \tag{2.4}$$

[13] A string is also referred to as a **word**, which continues with the linguistic terminology.

where

$$\Sigma^n \stackrel{\text{def}}{=} \underbrace{\Sigma \times \cdots \times \Sigma}_{n \text{ times}} \tag{2.5}$$

Note that we define $\Sigma^0 \stackrel{\text{def}}{=} \{\varepsilon\}$. *We call the* Σ^* *the* ***Kleene closure*** *of the alphabet* Σ. *We also define*

$$\Sigma^+ \stackrel{\text{def}}{=} \bigcup_{n=1}^{\infty} \Sigma^n = \Sigma\Sigma^*. \tag{2.6}$$

Finally, we also define the set of all infinite sequences of symbols from some alphabet Σ as Σ^∞.

Definition 2.3.4. *Let* Σ *be an alphabet. The set of all* ***infinite sequences*** *over* Σ *is defined as:*

$$\Sigma^\infty \stackrel{\text{def}}{=} \underbrace{\Sigma \times \cdots \times \Sigma}_{\infty\text{-times}}, \tag{2.7}$$

Since strings are canonically *finite* in computer science, we will explicitly use the terms infinite sequence or infinite string to refer to elements of Σ^∞.

More informally, we can think of Σ^* as the set which contains ε and all (finite-length) strings which can be constructed by concatenating arbitrary symbols from Σ. Σ^+, on the other hand, does *not* contain ε, but contains all other strings of symbols from Σ. The Kleene closure of an alphabet is a *countably infinite* set (this will come into play later!). In contrast, the set Σ^∞ is *uncountably infinite* for any Σ such that $|\Sigma| \geqslant 2$.

The notion of the Kleene closure leads us very naturally to our next definition.

Definition 2.3.5 (Formal language)**.** *Let* Σ *be an alphabet. A* ***language*** L *is a subset of* Σ^*.

That is, a language is just a specified subset of all possible strings made up of the symbols in the alphabet. This subset can be specified by simply enumerating a finite set of strings, or by a *formal model*. We will see examples of those later. Importantly, these strings are *finite*. If not specified explicitly, we will often assume that $L = \Sigma^*$.

A note on terminology. As we mentioned, these definitions are inspired by formal language theory. We defined strings as our main structures of interest and symbols as their building blocks. When we talk about natural language, the terminology is often slightly different: we may refer to the basic building blocks (symbols) as **tokens** or **words** (which might be composed of one or more *characters* and form some form of "words"; more on this in the chapter on tokenization) and their compositions (strings) as **sequences** or **sentences**. Furthermore, what we refer to here as an alphabet may be called a **vocabulary** (of words or tokens) in the context of natural language. Sentences are therefore concatenations of words from a vocabulary in the same way that strings are concatenations of symbols from an alphabet.

Example 2.3.1 (Kleene Closure)**.** *Let* $\Sigma = \{a, b, c\}$*. Then*

$$\Sigma^* = \{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, aac, \dots\}.$$

Examples of a languages over this alphabet include $L_1 \stackrel{\text{def}}{=} \{a, b, ab, ba\}$*,* $L_2 \stackrel{\text{def}}{=} \{\boldsymbol{y} \in \Sigma^* \mid y_1 = a\}$*, and* $L_3 \stackrel{\text{def}}{=} \{\boldsymbol{y} \in \Sigma^* \mid |\boldsymbol{y}| \text{ is even}\}$*.*

Next, we introduce two notions of subelements of strings.

Definition 2.3.6 (String Subelements)**.** *A* ***subsequence*** *of a string* $\boldsymbol{y}$ *is defined as a sequence that can be formed from* $\boldsymbol{y}$ *by deleting some or no symbols, leaving the order untouched. A* ***substring*** *is a contiguous subsequence. For instance,* ab *and* bc *are substrings and subsequences of* $\boldsymbol{y} = abc$*, while* ac *is a subsequence but not a substring.* ***Prefixes*** *and* ***suffixes*** *are special cases of substrings. A prefix is a substring of* $\boldsymbol{y}$ *that shares the same first letter as* $\boldsymbol{y}$ *and a suffix is a substring of* $\boldsymbol{y}$ *that shares the same last letter as* $\boldsymbol{y}$*. We will also denote a prefix* $y_1 \dots y_{n-1}$ *of the string* $\boldsymbol{y} = y_1 \dots y_T$ *as* $\boldsymbol{y}_{<n}$*.*

2.3.2 Defining a Language Model

We are now ready to introduce the main interest of the entire lecture series: language models.

Definition 2.3.7 (Language model)**.** *Let* Σ *be an alphabet. A* ***language model*** *is a (discrete) distribution* p_{LM} *over* Σ^**.*

Example 2.3.2 (A very simple language model)**.** *Let* $\Sigma \stackrel{\text{def}}{=} \{a\}$*. For* $n \in \mathbb{N}_{\geqslant 0}$*, define*

$$p_{LM}(a^n) \stackrel{\text{def}}{=} 2^{-(n+1)},$$

where $a^0 \stackrel{\text{def}}{=} \varepsilon$ *and* $a^n \stackrel{\text{def}}{=} \underbrace{a \dots a}_{n \text{ times}}$*.*

We claim that p_{LM} *is a language model. To see that, we verify that it is a valid probability distribution over* Σ^**. It is easy to see that* $p_{LM}(a^n) \geqslant 0$ *for any* n*. Additionally, we see that the probabilities of finite sequences indeed sum to* 1*:*

$$\sum_{\boldsymbol{y} \in \Sigma^*} p_{LM}(\boldsymbol{y}) = \sum_{n=0}^{\infty} p_{LM}(a^n) = \sum_{n=0}^{\infty} 2^{-(n+1)} = \frac{1}{2} \sum_{n=0}^{\infty} 2^{-n} = \frac{1}{2} \frac{1}{1 - \frac{1}{2}} = 1.$$

In our formal analysis of language models, we will also often refer to the *language* defined by a language model.

Definition 2.3.8. *Let* p_{LM} *be a language model. The* ***weighted language*** *of* p_{LM} *is defined as*

$$L(p_{LM}) \stackrel{\text{def}}{=} \{(\boldsymbol{y}, p_{LM}(\boldsymbol{y})) \mid \boldsymbol{y} \in \Sigma^*\} \tag{2.8}$$

Example 2.3.3. *The language of the language model from Example 2.3.2 is*

$$L(p_{LM}) \stackrel{\text{def}}{=} \left\{ \left(a^n, 2^{-(n+1)} \right) \mid n \in \mathbb{N}_{\geqslant 0} \right\} \tag{2.9}$$

A language model is itself a very simple concept—it is simply a distribution that weights strings (natural utterances) by their probabilities to occur in a particular language. Note that we have not said anything about how we can represent or model this distribution yet. Besides, for any (natural) language, the ground-truth language model p_{LM} is of course *unknown* and complex. The next chapter, therefore, discusses in depth the computational models which we can use to try to tractably represent distributions over strings and ways of *approximating* (learning) the ground-truth distribution based on finite datasets using such models.

2.4 Global and Local Normalization

The previous chapter introduced a formal definition of a language as a set of strings and the definition of a language model as a distribution over strings. We now delve into a potpourri of technical questions to complete the theoretical minimum for discussing language models. While doing so, we will introduce (and begin to answer) three fundamental questions in the first part of the course. We will introduce them later in the section.

A note on terminology. Unfortunately, we will encounter some ambiguous terminology. In §2.5, we explicitly define a language model as a valid probability distribution over Σ^*, the Kleene closure of some alphabet Σ, which means that $\sum_{\boldsymbol{y} \in \Sigma^*} p_{\text{LM}}(\boldsymbol{y}) = 1$. As we will see later, this means that the model is *tight*, whereas it is *non-tight* if $\sum_{\boldsymbol{y} \in \Sigma^*} p_{\text{LM}}(\boldsymbol{y}) < 1$. Definitionally, then, all language models are tight. However, it is standard in the literature to refer to many non-tight language models as language models well. We pardon in advance the ambiguity that this introduces. Over the course of the notes, we attempt to stick to the convention that the term "language model" without qualification only refers to a tight language model whereas a "non-tight language model" is used to refer to a language model in the more colloquial sense. Linguistically, tight is acting as a non-intersective adjective. Just as in English, where a fake gun is not a gun, so too in our course notes a non-tight language model is not a language model. This distinction does in fact matter. On one hand, we can prove that many language models whose parameters are estimated from data (e.g., a finite-state language model estimated by means of maximum-likelihood estimation) are, in fact, tight. On the other hand, we can show that this is *not* true in general, i.e., *not* all language models estimated from data will be tight. For instance, a recurrent neural network language model estimated through gradient descent may not be tight (Chen et al., 2018).

When specifying p_{LM}, we have two fundamental options. Depending on whether we model $p_{\text{LM}}(\boldsymbol{y})$ for each string $\boldsymbol{y}$ *directly* or we model *individual* conditional probabilities $p_{\text{LM}}(y_n \mid \boldsymbol{y}_{<t})$ we distinguish *globally* and *locally* normalized models. The names naturally come from the way the distributions in the two families are normalized: whereas globally normalized models are normalized by summing over the entire (infinite) space of strings, locally normalized models define a sequence of *conditional distributions* and make use of the chain rule of probability to define the joint probability of a whole string.

The bos symbol. Conventionally, we will include a special symbol over which globally or locally normalized models operate: the **beginning of sequence** (BOS) symbol, which, as the name suggests, denotes the beginning of a string or a sequence. For a string $\boldsymbol{y} = y_1 \cdots y_T$, we will suggestively denote $y_0 \stackrel{\text{def}}{=} \text{BOS}$.

2.4.1 Globally Normalized Language Models

We start with globally normalized models. Such models are also called **energy-based** language models in the literature (Bakhtin et al., 2021). To define a globally normalized language model, we start with the definition of an energy function.

Definition 2.4.1. *An **energy function** is a function* $\widehat{p} : \Sigma^* \to \mathbb{R}$.

Inspired by concepts from statistical mechanics, an energy function can be used to define a very general class of probability distributions by normalizing its exponentiated negative values.

Now, we can define a globally normalized language model in terms of an energy function over Σ^*.

Definition 2.4.2. *Let* $\widehat{p}_{GN}(\boldsymbol{y}) : \Sigma^* \to \mathbb{R}$ *be an energy function. A **globally normalized model** (GNM) is defined as*

$$p_{LM}(\boldsymbol{y}) \stackrel{\text{def}}{=} \frac{\exp\left[-\widehat{p}_{GN}(\boldsymbol{y})\right]}{\sum_{\boldsymbol{y}' \in \Sigma^*} \exp\left[-\widehat{p}_{GN}(\boldsymbol{y}')\right]} \stackrel{\text{def}}{=} \frac{1}{Z_G} \exp\left[-\widehat{p}_{GN}(\boldsymbol{y})\right], \tag{2.10}$$

where $Z_G \stackrel{\text{def}}{=} \sum_{\boldsymbol{y}' \in \Sigma^*} \exp\left[-\widehat{p}_{GN}(\boldsymbol{y}')\right]$.[14] *We call* Z_G *the **normalization constant**.*

Globally normalized models are attractive because one only needs to define an (unnormalized) energy function $\widehat{p}_{\text{GN}}$, which scores entire sequences at once. This is often easier than specifying a probability distribution. Furthermore, they define a probability distribution over strings $\boldsymbol{y} \in \Sigma^*$ *directly*. As we will see in §2.4.2, this stands in contrast to locally normalized language models which require care with the space over which they operate. However, the downside is that it may be difficult to compute the normalizer Z_G.

Normalizability

In defining the normalizer $Z_G \stackrel{\text{def}}{=} \sum_{\boldsymbol{y}' \in \Sigma^*} \exp\left[-\widehat{p}_{\text{GN}}(\boldsymbol{y}')\right]$, we notationally cover up a certain subtlety. The set Σ^* is countably infinite, so Z_G may diverge to ∞. In this case, Eq. (2.10) is not well-defined. This motivates the following definition.

Definition 2.4.3 (Normalizable energy function)**.** *We say that an energy function is **normalizable** if the quantity* Z_G *in Eq. (2.10) is finite, i.e., if* $Z_G < \infty$.

With this definition, we can state a relatively trivial result that characterizes when an energy function can be turned into a globally normalized language model.

Theorem 2.4.1. *Any normalizable energy function* p_{GN} *induces a language model, i.e., a distribution over* Σ^*.

[14] We will later return to this sort of normalization when we define the softmax function in §3.1.

Proof. Given an energy function $\widehat{p}_{\text{GN}}$, we have $\exp\left[-\widehat{p}_{\text{GN}}\left(\boldsymbol{y}\right)\right] \geqslant 0$ and

$$\sum_{\boldsymbol{y}\in\Sigma^*} p_{\text{GN}}\left(\boldsymbol{y}\right) = \sum_{\boldsymbol{y}\in\Sigma^*} \frac{\exp\left[-\widehat{p}_{\text{GN}}\left(\boldsymbol{y}\right)\right]}{\sum_{\boldsymbol{y}'\in\Sigma^*} \exp\left[-\widehat{p}_{\text{GN}}\left(\boldsymbol{y}'\right)\right]} \tag{2.11}$$

$$= \frac{1}{\sum_{\boldsymbol{y}'\in\Sigma^*} \exp\left[-\widehat{p}_{\text{GN}}\left(\boldsymbol{y}'\right)\right]} \sum_{\boldsymbol{y}\in\Sigma^*} \exp\left[-\widehat{p}_{\text{GN}}\left(\boldsymbol{y}\right)\right] \tag{2.12}$$

$$= 1, \tag{2.13}$$

which means that p_{GN} is a valid probability distribution over Σ^*. ■

While the fact that normalizable energy functions always form a language model is a big advantage, we will see later that *ensuring* that they are normalizable can be difficult and restrictive. This brings us to the first fundamental question of the section:

Question 2.4.1. *When is an energy function normalizable? More precisely, for which energy functions $\widehat{p}_{GN}$ is $Z_G < \infty$?*

We will not discuss any specific results here, as there are no general necessary or sufficient conditions—the answer to this of course depends on the precise definition of $\widehat{p}_{\text{GN}}$. Later in the course notes, we will present two formalisms where we can exactly characterize when an energy function is normalizable. First, when it is weighted finite-state automaton (cf. §4.1), and, second, when it is defined through weighted context-free grammars (§4.2) and discuss the specific sufficient and necessary conditions there. However, under certain assumptions, determining whether an energy function is normalizable in the general case is undecidable.

Moreover, even if it is known that an energy function is normalizable, we still need an efficient algorithm to compute it. But, efficiently computing Z_G can be challenging: the fact that Σ^* is *infinite* means that we cannot always compute Z_G in a *tractable* way. In fact, there are no general-purpose algorithms for this. Moreover, sampling from the model is similarly intractable, as entire sequences have to be drawn at a time from the large space Σ^*.

2.4.2 Locally Normalized Language Models

The inherent difficulty in computing the normalizer, an infinite summation over Σ^*, motivates the definition of locally normalized language models, which we will denote with p_{LN}. Rather than defining a probability distribution over Σ^* directly, they decompose the problem into the problem of modeling a series of conditional distributions over the next possible symbol in the string given the context so far, i.e., $p_{\text{LN}}\left(y \mid \boldsymbol{y}\right)$, which could be naïvely combined into the full probability of the string by multiplying the conditional probabilities.[15] Intuitively, this reduces the problem of having to normalize the distribution over an infinite set Σ^* to the problem of modeling the distribution of the *next possible symbol* y_n given

[15] We will soon see why this would not work and why we have to be a bit more careful.

the symbols seen so far $\boldsymbol{y}_{<n}$. This means that normalization would only ever require summation over $|\Sigma|$ symbols at a time, solving the tractability issues encountered by globally normalized models.

However, we immediately encounter another problem: In order to be a language model, $p_{\text{LN}}(y \mid \boldsymbol{y})$ must constitute a probability distribution over Σ^*. However, as we will discuss in the next section, this may not be the case because locally normalized models can place positive probability mass on *infinitely long* sequences (cf. Example 2.5.1 in §2.5.1). Additionally, we also have to introduce a new symbol that tells us to "stop" generating a string, which we call the **end of sequence** symbol, EOS. Throughout the notes, we will assume $\text{EOS} \notin \Sigma$ and we define

$$\overline{\Sigma} \stackrel{\text{def}}{=} \Sigma \cup \{\text{EOS}\}. \tag{2.14}$$

Moreover, we will explicitly denote elements of $\overline{\Sigma}^*$ as $\overline{\boldsymbol{y}}$ and symbols in $\overline{\Sigma}$ as $\overline{y}$. Given a sequence of symbols and the EOS symbol, we take the string to be the sequence of symbols encountered *before* the *first* EOS symbol. Informally, you can think of the BOS symbol as marking the beginning of the string, and the EOS symbol as denoting the end of the string or even as a language model terminating its generation, as we will see later.

Due to the issues with defining valid probability distributions over Σ^*, we will use the term sequence model to refer to any model that may place positive probability on infinitely long sequences. Thus, sequence models are strictly more general than language models, which, by definition, only place positive probability mass on strings, i.e., final sequences.

Definition 2.4.4. *Let Σ be an alphabet. A **sequence model** (SM) over Σ is defined as a set of conditional probability distributions*

$$p_{SM}(y \mid \boldsymbol{y}) \tag{2.15}$$

*for $y \in \Sigma$ and $\boldsymbol{y} \in \Sigma^*$. We will refer to the string $\boldsymbol{y}$ in $p_{SM}(y \mid \boldsymbol{y})$ as the **history** or the **context**.*

Note that we will mostly consider SMs over the set $\overline{\Sigma}$. To reiterate, we have just formally defined locally normalized *sequence* models rather than locally normalized *language* models. That has to do with the fact that, in contrast to a globally normalized model with a normalizable energy function, a SM might not correspond to a *language* model, as alluded to at the beginning of this section and as we discuss in more detail shortly.

We will now work up to a locally normalized *language* model.

Definition 2.4.5 (Locally normalized language model)**.** *Let Σ be an alphabet. Next, let p_{SM} be a sequence model over $\overline{\Sigma}$. A **locally normalized language model** (LNM) over Σ is defined as*

$$p_{LN}(\boldsymbol{y}) \stackrel{\text{def}}{=} p_{SM}(\text{EOS} \mid \boldsymbol{y}) \prod_{t=1}^{T} p_{SM}(y_t \mid \boldsymbol{y}_{<t}) \tag{2.16}$$

for $\boldsymbol{y} \in \Sigma^*$ *with* $|\boldsymbol{y}| = T$*. We say a locally normalized language model is* ***tight*** *if*

$$\sum_{\boldsymbol{y} \in \Sigma^*} p_{LN}(\boldsymbol{y}) = 1. \tag{2.17}$$

Tightness is a nuanced concept that will be discussed in great detail in §2.5.

We now contrast globally and locally normalized models pictorially in the following example.

Example 2.4.1. *Fig. 2.2a shows a simple instance of what a locally normalized language model would look like. We can compute the probabilities of various strings by starting at the root node* BOS *and choosing one of the paths to a leaf node, which will always be* EOS*. The values on the edges represent the conditional probabilities of observing the new word given at the target of the edge given the context seen on the path so far, i.e.,* $p_{LN}(\overline{y}_t \mid \overline{\boldsymbol{y}}_{<t})$ *at the level* t *of the tree. For example, the probability of the string* BOS *"The best"* EOS *under this language model is* $0.04 \cdot 0.13 \cdot 0.22 = 0.001144$*. On the other hand, a globally normalized model would simply score all possible sentences using the score function* $\widehat{p}_{GN}(\boldsymbol{y})$*, as is hinted at in Fig. 2.2b.*

Locally Normalizing a Language Model

The second fundamental question of this section concerns the relationship between language models and local normalization.

Question 2.4.2. *When can a language model be locally normalized?*

The answer to that is simple: *every* language model can be locally normalized! While the intuition behind this is very simple, the precise formulation is not. Before we discuss the details, we have to introduce the concept of prefix probabilities, which denote the sum of the probabilities of all strings beginning with a certain prefix.

Definition 2.4.6. *Let* p_{LM} *be a language model. We define* p_{LM}*'s* ***prefix probability*** π *as*

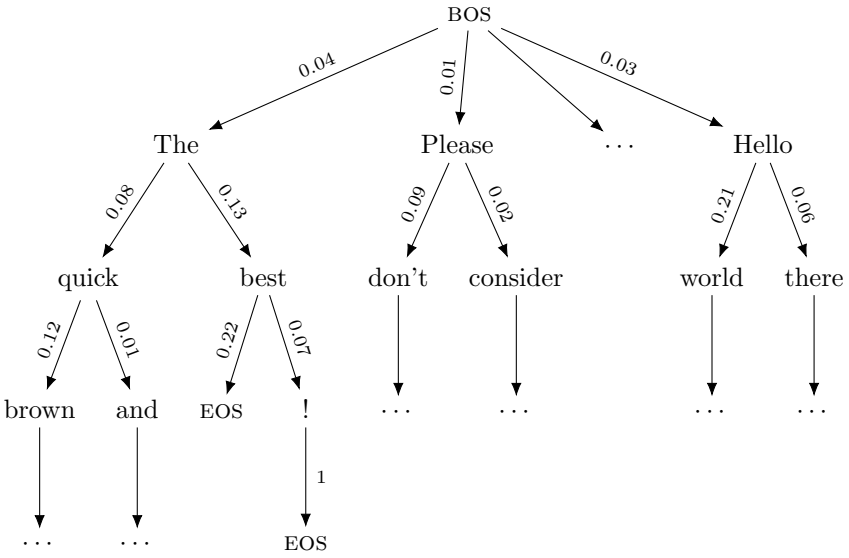
$$\pi(\boldsymbol{y}) \stackrel{\text{def}}{=} \sum_{\boldsymbol{y}' \in \Sigma^*} p_{LM}(\boldsymbol{y}\boldsymbol{y}'), \tag{2.18}$$

that is, the probability that $\boldsymbol{y}$ *is a prefix of any string* $\boldsymbol{y}\boldsymbol{y}'$ *in the language, or, equivalently, the cumulative probability of all strings beginning with* $\boldsymbol{y}$*.*

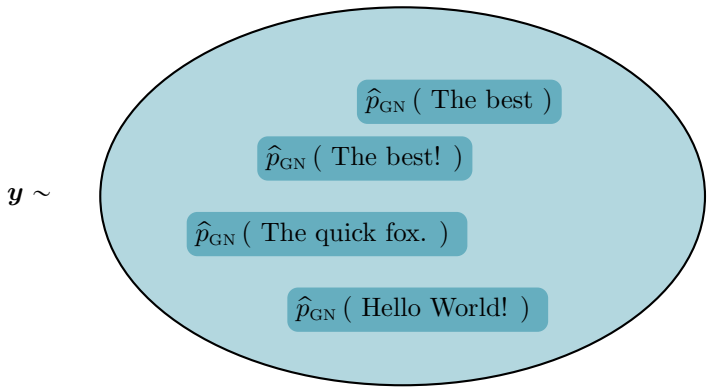
Note that, naturally, $\pi(\varepsilon) = 1$.

Theorem 2.4.2. *Let* p_{LM} *be a language model. Then, there exists a locally normalized language model* p_{LN} *such that, for all* $\boldsymbol{y} \in \Sigma^*$ *with* $|\boldsymbol{y}| = T$*,*

$$p_{LM}(\boldsymbol{y}) = p_{LN}(\boldsymbol{y}) = p_{SM}(\text{EOS} \mid \boldsymbol{y}) \prod_{t=1}^{T} p_{SM}(y_t \mid \boldsymbol{y}_{<t}). \tag{2.19}$$



(a) An example of a locally normalized language model. The values of the edges represent the conditional probability of observing the new word given the observed words (higher up on the path from the root node BOS. Note that the probabilities stemming from any inner node should sum to 1—however, to avoid clutter, only a subset of the possible arcs is drawn.



(b) An example of a globally normalized model which can for example generate sentences based on the probabilities determined by normalizing the assigned scores $\widehat{p}_{\text{GN}}$.

Figure 2.2: "Examples" of a locally and a globally normalized language model.

Proof. We define the individual conditional probability distributions over the next symbol of the SM p_{SM} using the chain rule of probability. If $\pi(\boldsymbol{y}) > 0$, then define

$$p_{\text{SM}}\left(y \mid \boldsymbol{y}\right) \stackrel{\text{def}}{=} \frac{\pi\left(\boldsymbol{y}y\right)}{\pi\left(\boldsymbol{y}\right)} \tag{2.20}$$

for $y \in \Sigma$ and $\boldsymbol{y} \in \Sigma^*$ such that $p\left(\boldsymbol{y}\right) > 0$. We still have to define the probabilities of *ending* the sequence using p_{SM} by defining the EOS probabilities. We define, for any $\boldsymbol{y} \in \Sigma^*$ such that $\pi(\boldsymbol{y}) > 0$,

$$p_{\text{SM}}\left(\text{EOS} \mid \boldsymbol{y}\right) \stackrel{\text{def}}{=} \frac{p_{\text{LM}}(\boldsymbol{y})}{\pi(\boldsymbol{y})} \tag{2.21}$$

that is, the probability that the globally normalized model will generate *exactly* the string $\boldsymbol{y}$ and not any continuation of it $\boldsymbol{y}\boldsymbol{y}'$, given that $\boldsymbol{y}$ has already been generated. Each of the conditional distributions of this model (Eqs. (2.20) and (2.21)) is clearly defined over $\overline{\Sigma}$. This, therefore, defines a valid SM. To see that p_{LN} constitutes the same distribution as p_{LM}, consider two cases.

Case 1: Assume $\pi(\boldsymbol{y}) > 0$. Then, we have

$$\begin{aligned}
p_{\text{LN}}\left(\boldsymbol{y}\right) &= \left[\prod_{t=1}^{T} p_{\text{SM}}\left(y_t \mid \boldsymbol{y}_{<t}\right)\right] p_{\text{SM}}\left(\text{EOS} \mid \boldsymbol{y}\right) && (2.22)\\
&= \frac{\cancel{\pi\left(y_1\right)}}{\pi\left(\varepsilon\right)} \frac{\cancel{\pi\left(y_1 y_2\right)}}{\cancel{\pi\left(y_1\right)}} \cdots \frac{\cancel{\pi\left(\boldsymbol{y}_{<T}\right)}}{\cancel{\pi\left(\boldsymbol{y}_{<T-1}\right)}} \frac{\pi\left(\boldsymbol{y}\right)}{\cancel{\pi\left(\boldsymbol{y}_{<T}\right)}} p_{\text{SM}}\left(\text{EOS} \mid \boldsymbol{y}\right) \\
&= \frac{\cancel{\pi\left(\boldsymbol{y}\right)}}{\pi\left(\varepsilon\right)} \frac{p_{\text{LM}}\left(\boldsymbol{y}\right)}{\cancel{\pi\left(\boldsymbol{y}\right)}} && (2.23)\\
&= p_{\text{LM}}\left(\boldsymbol{y}\right) && (2.24)
\end{aligned}$$

where $\pi\left(\varepsilon\right) = 1$.

Case 2: Assume $\pi(\boldsymbol{y}) = 0$. Let $\boldsymbol{y} = y_1 \cdots y_T$. Then, there must exist a $1 \leqslant t' \leqslant T$ such that $\pi(\boldsymbol{y}_{<t'}) = 0$. Note that

$$p_{\text{LN}}(\boldsymbol{y}) = \prod_{t=1}^{t'} p_{\text{SM}}(y_t \mid \boldsymbol{y}_{<t}) = 0 \tag{2.25}$$

whereas the conditional probabilities after t' can be arbitrarily defined since they do not affect the string having 0 probability. ■

When Is a Locally Normalized Language Model a Language Model?

LNMs which specify distributions over strings $p_{\text{LN}}\left(y_1 \ldots y_T\right)$ in terms of their conditional probabilities $p_{\text{SM}}(y_t \mid \boldsymbol{y}_{<t})$ for $t = 1, \ldots, T$ and $p_{\text{SM}}\left(\text{EOS} \mid \boldsymbol{y}\right)$ have become the standard in NLP literature. However, LNMs come with their own set of problems. An advantage of normalizable globally normalized models is that

they, by definition, always define a *valid* probability space over $\overline{\Sigma}$. Although this might be counterintuitive at first, the same cannot be said for LNMs—in this sense, locally normalized "language models" might not even be language models! One might expect that in a LNM p_{LN}, it would hold that $\sum_{\boldsymbol{y} \in \Sigma^*} p_{\text{LN}}(\boldsymbol{y}) = 1$. However, this might not be the case! This is the issue with the terminology we brought up earlier and it brings us to the last fundamental question of this section.

Question 2.4.3. *When does an LNM encode a language model?*

As the conditions are a bit more nuanced, it requires a longer treatment. We explore this issue in much more detail in the next section.

2.5 Tight Language Models

We saw in the last section that any language model p_{LM} can be converted into a locally normalized sequence model (cf. §2.4.2). The converse, however, is *not* true. As alluded to in the previous section and as we detail in this section, there exist sets of conditional distributions $p_{\text{LN}}(\overline{y} \mid \overline{\boldsymbol{y}})$ over $\overline{\Sigma}^*$ such that $p_{\text{LN}}(\overline{\boldsymbol{y}})$ as defined in Eq. (2.15) does not represent a valid probability measure over Σ^* (after taking into account the semantics of EOS), i.e., over the set of *finite* strings. Indeed, we will later show that some popular classes of locally normalized sequence models used in practice have parameter settings in which the generative process terminates with probability < 1. This means that p_{LN} "leaks" some of its probability mass to *infinite* sequences. This section investigates this behavior in a lot of detail. It is based on the recent work from Du et al. (2022).

2.5.1 Tightness

Models whose generative process may fail to terminate are called **non-tight** (Chi, 1999).[16]

Definition 2.5.1. *A locally normalized language model p_{LN} derived from a sequence model p_{SM} is called **tight** if it defines a valid probability distribution over Σ^*:*

$$\sum_{\boldsymbol{y} \in \Sigma^*} p_{LN}(\boldsymbol{y}) = \sum_{\boldsymbol{y} \in \Sigma^*} \left[p_{SM}(\text{EOS} \mid \boldsymbol{y}) \prod_{t=1}^{T} p_{SM}(y_t \mid \boldsymbol{y}_{<t}) \right] = 1. \tag{2.26}$$

Note that the individual conditional distributions $p_{\text{SM}}(y \mid \boldsymbol{y})$ in a non-tight LNM still are valid conditional distributions (i.e., they sum to one). However, the distribution over all possible strings that they induce may not sum to 1. To be able to investigate this phenomenon more closely, let us first examine what the conditional probabilities of an LNM actually define and how they can result in non-tightness. We now ask ourselves: given a sequence model p_{SM}, what is p_{LN}? Is p_{LN} a language model, i.e., a distribution over Σ^* (after taking into account the semantics of EOS)? Certainly, the answer is yes if the LNM's conditional probabilities match the conditional probabilities of some known language model p_{LM} as defined in §2.4.2,[17] in which case p_{LN} is specifically the language model p_{LM} itself. In this case clearly $p_{\text{LN}}(\Sigma^*) \stackrel{\text{def}}{=} \sum_{\boldsymbol{y} \in \Sigma^*} p_{\text{LN}}(\boldsymbol{y}) = \sum_{\boldsymbol{y} \in \Sigma^*} p_{\text{LM}}(\boldsymbol{y}) = 1$. If instead $p_{\text{LN}}(\Sigma^*) < 1$, the LNM's conditional probabilities do *not* match the conditional probabilities of any language model p_{LM}.

To see how this can happen, we now exhibit such an LNM in the following example.

[16] Tight models are also called **consistent** (Booth and Thompson, 1973; Chen et al., 2018) and **proper** (Chi, 1999) in the literature.

[17] That is, $p_{\text{LM}}(y_t \mid \boldsymbol{y}_{<t}) = p_{\text{LN}}(y_t \mid \boldsymbol{y}_{<t})$ whenever the former conditional probability is well-defined under the language model p_{LM}, i.e., whenever $y_t \in \overline{\Sigma}$ and $\boldsymbol{y}_{<t} \in \Sigma^*$ with $p_{\text{LM}}(\boldsymbol{y}_{<t}) > 0$.

Example 2.5.1. *Consider the bigram model defined in Fig. 2.3a over the alphabet* $\Sigma = \{a, b\}$.[18] *Although the conditional probability distributions* $p_{LN}(\cdot \mid \boldsymbol{y}_{<n})$ *each sum to 1 over* $\overline{\Sigma}$*, they fail to combine into a model* p_{LN} *that sums to 1 over* Σ^* *(i.e., a language model): under this model, any finite string that contains the symbol b will have probability 0, since* $p_{LN}(\text{EOS} \mid b) = p_{LN}(a \mid b) = 0$*. This implies* $p_{LN}(\Sigma^*) = \sum_{n=0}^{\infty} p_{LN}(a^n) = \sum_{n=0}^{\infty} (0.7)^n \cdot 0.1 = \frac{0.1}{1-0.7} = \frac{1}{3} < 1$.

Example 2.5.2. *On the other hand, in the bigram model in Fig. 2.3b, obtained from Example 2.5.1 by changing the arcs from the b state,* $p_{LN}(\Sigma^*) = 1$*. We can see that by calculating:*

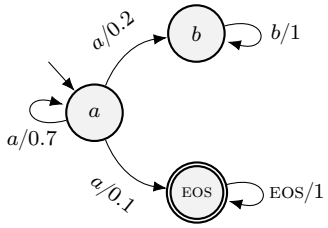
$$
\begin{aligned}
\mathbb{P}(\Sigma^*) &= \sum_{n=1}^{\infty} \sum_{m=0}^{\infty} \mathbb{P}(a^n b^m) \\
&= \sum_{n=1}^{\infty} \left(\mathbb{P}(a^n) + \sum_{m=1}^{\infty} \mathbb{P}(a^n b^m) \right) \\
&= \sum_{n=1}^{\infty} \left(0.1 \cdot (0.7)^{n-1} + \sum_{m=1}^{\infty} (0.7)^{n-1} \cdot 0.2 \cdot (0.9)^{m-1} \cdot 0.1 \right) \\
&= \sum_{n=1}^{\infty} \left(0.1 \cdot (0.7)^{n-1} + (0.7)^{n-1} \cdot 0.2 \cdot \frac{1}{1-0.9} \cdot 0.1 \right) \\
&= \sum_{n=1}^{\infty} \left(0.1 \cdot (0.7)^{n-1} + 0.2 \cdot (0.7)^{n-1} \right) \\
&= \sum_{n=1}^{\infty} 0.3 \cdot (0.7)^{n-1} = \frac{0.3}{1-0.7} = 1.
\end{aligned}
$$

Example 2.5.1 confirms that the local normalization does not necessarily yield p_{LN} that is a valid distribution over Σ^*. But if p_{LN} is not a language model, *what* is it? It is intuitive to suspect that, in a model with $p_{\text{LN}}(\Sigma^*) < 1$, the remainder of the probability mass "leaks" to infinite sequences, i.e., the generative process may continue forever with probability > 0. This means that, to be able to characterize p_{LN}, we will have to be able to somehow take into account infinite sequences. We will make this intuition formal below.

Delving a bit deeper, the non-tightness of Example 2.5.1 is related to the fact that the conditional probability of EOS is 0 at some states, in contrast to Example 2.5.2. However, requiring $p_{\text{LN}}(y_n = \text{EOS} \mid \boldsymbol{y}_{<n}) > 0$ for all prefixes $\boldsymbol{y}_{<n}$ is neither *necessary* nor *sufficient* to ensure tightness. It is not necessary because one can, for example, construct an LNM in which $p_{\text{LN}}(y_n = \text{EOS} \mid \boldsymbol{y}_{<n}) = 0.1$ when n is even but $= 0$ otherwise. Such a model generates only odd-length strings but is tight. We will postpone non-sufficienty for later, where we will

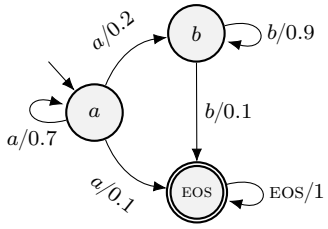
[18] The graphical representation of the LNM depicts a so-called weighted finite-state automaton, a framework of language models we will introduce shortly. For now, it is not crucial that you understand the graphical representation and you can simply focus on the conditional probabilities specified in the figure.

$p_{\text{LN}}(a \mid \text{BOS})$	1
$p_{\text{LN}}(a \mid a)$	0.7
$p_{\text{LN}}(b \mid a)$	0.2
$p_{\text{LN}}(\text{EOS} \mid a)$	0.1
$p_{\text{LN}}(b \mid b)$	1
$p_{\text{LN}}(\text{EOS} \mid \text{EOS})$	1



(a) Non-tight 2-gram model.

$p_{\text{LN}}(a \mid \text{BOS})$	1
$p_{\text{LN}}(a \mid a)$	0.7
$p_{\text{LN}}(b \mid a)$	0.2
$p_{\text{LN}}(\text{EOS} \mid a)$	0.1
$p_{\text{LN}}(b \mid b)$	0.9
$p_{\text{LN}}(\text{EOS} \mid b)$	0.1
$p_{\text{LN}}(\text{EOS} \mid \text{EOS})$	1



(b) Tight 2-gram model.

Figure 2.3: Tight and non-tight bigram models, expressed as Mealy machines. Symbols with conditional probability of 0 are omitted.

present specific LNMs under which the conditional probability of EOS is always > 0, yet are non-tight.

2.5.2 Defining the probability measure of an LNM

We now rigorously characterize the kind of distribution induced by an LNM, i.e., we investigate what p_{LN} is. As mentioned earlier, an LNM can lose probability mass to the set of infinite sequences, Σ^∞. However, Σ^∞, unlike Σ^*, is *uncountable*, and it is due to this fact that we need to work explicitly with the *measure-theoretic* formulation of probability which we introduced in §2.2. We already saw the peril of not treating distributions over uncountable sets carefully is necessary in Example 2.1.1—the set of all infinite sequences of coin tosses is indeed uncountable.

Including infinite strings and eos. As we saw in Example 2.1.1, sampling successive symbols from a non-tight LNM has probability > 0 of continuing forever, i.e., generating infinite strings. Motivated by that, we hope to regard the LNM as defining a valid probability space over $\Omega = \Sigma^* \cup \Sigma^\infty$, i.e., both finite as well as infinite strings, and then "relate" it to our definition of true language models. Notice, however, that we also have to account for the difference in the alphabets: while we would like to characterize language models in terms of strings over the alphabet Σ, LNMs work over symbols in $\overline{\Sigma}$.

With this in mind, we now embark on our journey of discovering what p_{LN} represents. Given an LNM, we will first need to turn its p_{LN} into a measurable space by defining an appropriate σ-algebra. This type of distribution is more general than a language model as it works over both finite as well as infinite sequences. To distinguish the two, we will expand our vocabulary and explicitly *differentiate* between true language models and non-tight LNMs. We will refer to a distribution over $\Sigma^* \cup \Sigma^\infty$ as a sequence model. As noted in our definition of a sequence model (cf. Definition 2.4.4), an LNM defines a probabilty measure over $\Sigma^* \cup \Sigma^\infty$. Thus, an equivalent distribution, which will be useful for this section, would be the following.

Definition 2.5.2 (Sequence model)**.** *A* ***sequence model*** *is a probability space over the set* $\Sigma^* \cup \Sigma^\infty$*.*

Intuitively, and we will make this precise later, the set $\Sigma^\infty \subset \Sigma^* \cup \Sigma^\infty$ in Definition 2.5.2 represents the *event* where the sequence model is **non-terminating**, i.e., it attempts to generate an infinitely long sequence. We can then understand language models in a new sense.

Definition 2.5.3 (Re-definition of a Language model)**.** *A* ***language model*** *is a probability space over* Σ^**. Equivalently, a language model is a sequence model such that* $\mathbb{P}(\Sigma^\infty) = 0$*.*

Now buckle up! Our goal through the rest of this section is to rigorously construct a probability space of a sequence model as in Definition 2.2.2 and Definition 2.5.2 which encodes the probabilities assigned by an LNM. Then, we will use this characterization to formally investigate tightness. An outline of what this is going to look like is shown in Fig. 2.4.

Defining an Algebra over $\overline{\Sigma}^\infty$ (Step 1)

Since an LNM produces conditional distributions over the augmented alphabet $\overline{\Sigma}$ (first box in Fig. 2.4) and results in possibly infinite strings, we will first construct a probability space over $\overline{\Sigma}^\infty$, which will naturally induce a sequence model. We will do that by first constructing an *algebra* (cf. Definition 2.2.3) over $\Omega = \overline{\Sigma}^\infty$ for some alphabet Σ (second box in Fig. 2.4). Then, assuming we are given an LNM p_{LN} over $\overline{\Sigma}$, we will associate the constructed algebra with a pre-measure (cf. Definition 2.2.4) that is "consistent" with p_{LN} (third box in Fig. 2.4).

We will make use of the following definition to construct the algebra:

Definition 2.5.4. *Given any set* $\mathcal{H} \subseteq \overline{\Sigma}^k$*, i.e., a set of sequences of symbols from* $\overline{\Sigma}$ *of length* k*, define its* ***cylinder set*** *(of rank* k*) to be*

$$\overline{C}(\mathcal{H}) \stackrel{\text{def}}{=} \left\{ \boldsymbol{y}\boldsymbol{\omega} \;:\; \boldsymbol{y} \in \mathcal{H}, \boldsymbol{\omega} \in \overline{\Sigma}^\infty \right\} \tag{2.27}$$

In essence, a cylinder set of rank k is the set of infinite strings that share their k-prefix with some string $\overline{\boldsymbol{y}} \in \mathcal{H} \subseteq \overline{\Sigma}^k$. In particular, for a length-k string

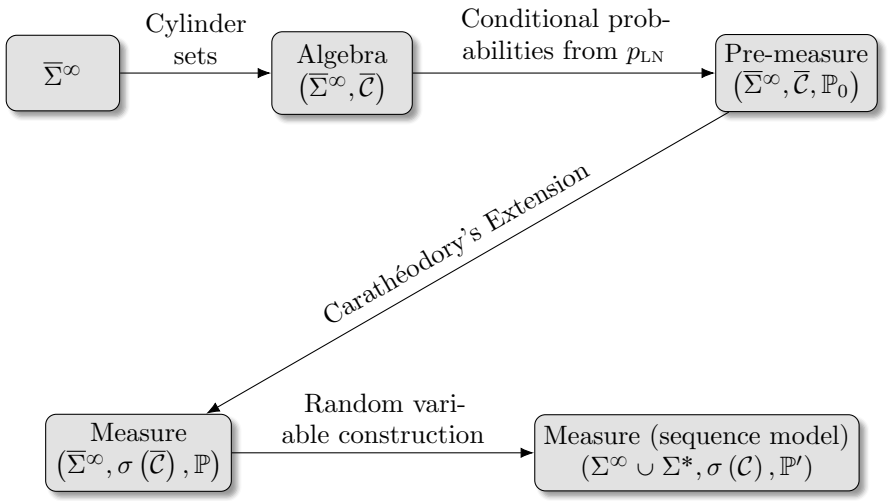


Figure 2.4: The outline of our measure-theoretic treatment of LNMs in this section to arrive at a precise characterization of p_{LN}. The final box corresponds to the sequence model (probability measure over $\Sigma^* \cup \Sigma^\infty$) constructed for p_{LN}.

$\overline{\boldsymbol{y}} = \overline{y}_1 \cdots \overline{y}_k$, the cylinder set $\overline{C}(\overline{\boldsymbol{y}}) \stackrel{\text{def}}{=} \overline{C}(\{\overline{\boldsymbol{y}}\})$ is the set of all infinite strings prefixed by $\overline{\boldsymbol{y}}$.[19]

We denote the collection of all rank-k cylinder sets by

$$\overline{\mathcal{C}}_k \stackrel{\text{def}}{=} \left\{\overline{\mathcal{C}}(\mathcal{H}) \;:\; \mathcal{H} \in \mathcal{P}\left(\overline{\Sigma}^k\right)\right\} \tag{2.28}$$

and define

$$\overline{\mathcal{C}} \stackrel{\text{def}}{=} \bigcup_{k=1}^{\infty} \overline{\mathcal{C}}_k \tag{2.29}$$

to be the collection of all cylinder sets over Ω.[20]

The following lemma asserts $\overline{\mathcal{C}} \subseteq \mathcal{P}(\Omega)$ is what we want in the second block of Fig. 2.4.

Lemma 2.5.1. *$\overline{\mathcal{C}} \subseteq \mathcal{P}(\Omega)$ is an algebra over $\Omega = \overline{\Sigma}^\infty$.*

Proof. First, $\overline{\Sigma}^\infty = \overline{C}(\overline{\Sigma}^k)$ for any k, and in particular is a cylinder set of any rank. Secondly, given a cylinder set $\overline{C}(\mathcal{H})$ of rank k, i.e., $\mathcal{H} \subseteq \overline{\Sigma}^k$, $\left(\overline{C}(\mathcal{H})\right)^{\mathsf{c}} = \overline{C}\left(\overline{\Sigma}^k \backslash \mathcal{H}\right)$. Hence, $\overline{\mathcal{C}}$ is closed under complements. Finally, notice that the intersection of two cylinder sets of ranks $k_1 \leqslant k_2$ is another cylinder set of rank k_2. Hence, $\overline{\mathcal{C}}$ is an algebra over Ω. ■

With this, the first step of Fig. 2.4 is done!

[19] This type of cylinder set, i.e., one that is generated by a singleton, is also called a **thin cylinder**.

[20] We invite the reader to verify that $\overline{\mathcal{C}}_1 \subset \overline{\mathcal{C}}_2 \subset \overline{\mathcal{C}}_3 \subset \cdots$.

Defining a Pre-measure over $\overline{\mathcal{C}}$ (Step 2)

We are now ready to define the pre-measure $\mathbb{P}_0$ for the cylinder algebra $\overline{\mathcal{C}}$. Given an LNM p_{LN} and any set $\overline{C}(\mathcal{H}) \in \overline{\mathcal{C}}$, let

$$\mathbb{P}_0(\overline{C}(\mathcal{H})) \stackrel{\text{def}}{=} \sum_{\overline{\boldsymbol{y}} \in \mathcal{H}} p_{\text{LN}}(\overline{\boldsymbol{y}}) \tag{2.30}$$

where we have defined

$$p_{\text{LN}}(\overline{\boldsymbol{y}}) \stackrel{\text{def}}{=} \prod_{t=1}^{T} p_{\text{LN}}(\overline{y}_t \mid \overline{\boldsymbol{y}}_{<t}). \tag{2.31}$$

Note that there is a caveat here since the same cylinder set may admit different $\mathcal{H}$.[21] Before showing that $\mathbb{P}_0$ defines a valid pre-measure, we address this and show that $\mathbb{P}_0$ is indeed well defined.

Proposition 2.5.1. *$\mathbb{P}_0$ as defined in Eq. (2.30) is a well-defined function.*

Proof. Suppose a cylinder set can be described by two different prefix sets: $H_1 \subseteq \overline{\Sigma}^{k_1}$ and $H_2 \subseteq \overline{\Sigma}^{k_2}$. In other words, $\overline{C}(H_1) = \overline{C}(H_2)$. Without loss of generality, assume that $k_1 \leqslant k_2$. Then,

$$\begin{aligned}
\overline{C}(H_2) &= \overline{C}(H_1) && \text{(2.32a)}\\
&= \bigcup_{\boldsymbol{y} \in H_1} \overline{C}(\boldsymbol{y}) && \text{(2.32b)}\\
&= \bigcup_{\boldsymbol{y} \in H_1} \bigcup_{\overline{\boldsymbol{y}} \in \overline{\Sigma}^{k_2 - k_1}} \overline{C}(\boldsymbol{y}\boldsymbol{y}). && \text{(2.32c)}
\end{aligned}$$

All the unions above are disjoint, and hence $H_2 = \bigcup_{\boldsymbol{y} \in \overline{\Sigma}^{k_2 - k_1}} \{\boldsymbol{y}\boldsymbol{y} : \boldsymbol{y} \in H_1\}$. Then, by the locally-normalizing property of p_{LN}, we have that

$$\mathbb{P}_0(\overline{C}(H_1)) = \mathbb{P}_0(\overline{C}(H_2)). \tag{2.33}$$

■

With this, we are able to state and prove the lemma which shows that $\mathbb{P}_0$ is a pre-measure, which is what we need in the third block of Fig. 2.4.

Lemma 2.5.2. *$\mathbb{P}_0$ is a pre-measure over $\overline{\mathcal{C}}$.*

For the proof of Lemma 2.5.2, we will mostly follow the proof of Theorem 2.3 in Billingsley (1995), with the exception of invoking the Tychonoff theorem directly. This proof depends on the following lemma, which is Example 2.10 in Billingsley (1995). We repeat the statement and proof here for the reader's convenience.

[21] For example, in the infinite coin toss model, $C(\mathtt{H}) = C(\{\mathtt{HH}, \mathtt{HT}\})$.

Lemma 2.5.3. *Let $\mathbb{P}_0$ be a finitely additive probability pre-measure over $\overline{\mathcal{C}}$ such that, given a decreasing sequence of sets $A_1 \supset A_2 \supset \cdots$ in $\overline{\mathcal{C}}$ where $\bigcap_{n=1}^{\infty} A_n = \varnothing$, $\lim_{n\to\infty} \mathbb{P}_0(A_n) = 0$. Then, $\mathbb{P}_0$ is also countably additive over $\overline{\mathcal{C}}$.*

Proof. Let $\{A_n\}$ be a sequence of disjoint sets in $\overline{\mathcal{C}}$ such that $A = \bigcup_n A_n \in \overline{\mathcal{C}}$. Then, defining $B_n = \bigcup_{m>n} A_m$, we see that $B_1 \supset B_2 \supset \cdots$ and $\bigcap_n B_n = \varnothing$. Notice that

$$A = A_1 \cup B_1 = A_1 \cup A_2 \cup B_2 = \cdots = A_1 \cup \cdots \cup A_n \cup B_n \tag{2.34}$$

for any n and hence by finite additivity of $\mathbb{P}_0$

$$\mathbb{P}_0(A) = \mathbb{P}_0(A_1) + \cdots + \mathbb{P}_0(A_n) + \mathbb{P}_0(B_n) \tag{2.35}$$

or equivalently

$$\mathbb{P}_0(A_1) + \cdots + \mathbb{P}_0(A_n) = \mathbb{P}_0(A) - \mathbb{P}_0(B_n). \tag{2.36}$$

Since, $B_n \downarrow \varnothing$ implies that $\mathbb{P}_0(B_n) \downarrow 0$ by assumption, taking the limits on both sides of Eq. (2.36) yields

$$\sum_n \mathbb{P}_0(A_n) = \lim_{n\to\infty} \sum_{i\leqslant n} \mathbb{P}_0(A_i) = \mathbb{P}_0(A) - \lim_{n\to\infty} \mathbb{P}_0(B_n) = \mathbb{P}_0(A) \tag{2.37}$$

which shows countable additivity. ■

We also recall the Tychonoff theorem.[22]

Theorem 2.5.1 (Tychonoff)**.** *Let $\{\mathcal{X}_\alpha\}_{\alpha\in J}$ be an indexed family of compact topologies. Then, their product topology $\prod_{\alpha\in J} \mathcal{X}_\alpha$ is also compact.*

We can now give the proof for Lemma 2.5.2.

Proof of Lemma 2.5.2. We first show that $\mathbb{P}_0$ is finitely additive over $\overline{\mathcal{C}}$. Let $\mathcal{C}(\mathcal{H}_1)$ and $\mathcal{C}(\mathcal{H}_2)$ be two disjoint cylinder sets. By Proposition 2.5.1, we can assume they are of the same rank without loss of generality. Then,

$$\mathcal{C}(\mathcal{H}_1) \cup \mathcal{C}(\mathcal{H}_2) = \bigcup_{\boldsymbol{y}\in\mathcal{H}_1} \{\boldsymbol{y}\boldsymbol{\omega} : \boldsymbol{\omega} \in \overline{\Sigma}^\infty\} \cup \bigcup_{\boldsymbol{y}\in\mathcal{H}_2} \{\boldsymbol{y}\boldsymbol{\omega} : \boldsymbol{\omega} \in \overline{\Sigma}^\infty\} \tag{2.38a}$$

$$= \bigcup_{\boldsymbol{y}\in\mathcal{H}_1\cup\mathcal{H}_2} \{\boldsymbol{y}\boldsymbol{\omega} : \boldsymbol{\omega} \in \overline{\Sigma}^\infty\} \qquad (\mathcal{H}_1 \text{ and } \mathcal{H}_2 \text{ equal rank and disjoint}) \tag{2.38b}$$

$$= \mathcal{C}(\mathcal{H}_1 \cup \mathcal{H}_2) \tag{2.38c}$$

which leads to

$$\mathbb{P}_0(\mathcal{C}(\mathcal{H}_1) \cup \mathcal{C}(\mathcal{H}_2)) = \mathbb{P}_0(\mathcal{C}(\mathcal{H}_1 \cup \mathcal{H}_2)) \tag{2.39a}$$

$$= \sum_{\boldsymbol{y}\in\mathcal{H}_1\cup\mathcal{H}_2} p_{\text{LN}}(\boldsymbol{y}) \tag{2.39b}$$

$$= \mathbb{P}_0(\mathcal{C}(\mathcal{H}_1)) + \mathbb{P}_0(\mathcal{C}(\mathcal{H}_2)). \tag{2.39c}$$

[22] See §37 in Munkres (2000) for a detailed and well-written treatise.

Hence, $\mathbb{P}_0$ is finitely additive.

Now, equip $\overline{\Sigma}$ with the discrete topology. Since $\overline{\Sigma}$ is finite, it is compact under the discrete topology and so is $\overline{\Sigma}^\infty$ by Theorem 2.5.1. Then, by properties of the product topology over discrete finite spaces, all cylinder sets in $\overline{\Sigma}^\infty$ are compact. To apply Lemma 2.5.3, let $\mathcal{C}_1 \supset \mathcal{C}_2 \supset \cdots$ be a decreasing sequence of cylinder sets with empty intersection. Suppose to the contrary that $\mathbb{P}_0\left(\bigcap_n \mathcal{C}_n\right) > 0$. This would imply that all $\mathcal{C}_n$ are nonempty (any of these being empty would result in a measure 0). However, by Cantor's intersection theorem[23], $\bigcap_n \mathcal{C}_n$ is nonempty, contradicting the assumption. Hence, $\mathbb{P}_0\left(\bigcap_n \mathcal{C}_n\right) = 0$, and by Lemma 2.5.3, $\mathbb{P}_0$ is countably additive. ■

With this, we have successfully completed the first two steps of Fig. 2.4! However, we have only defined a *pre*-measure over the set of *infinite* EOS-containing sequences $\overline{\Sigma}^\infty$. This does not yet satisfy all the properties we would like from a probability space. Because of that, we next *extend* the constructed probability pre-measure $\mathbb{P}_0$ into a valid probability measure $\mathbb{P}$ to arrive to a valid probability space.

Extending the Pre-measure $\mathbb{P}_0$ into a Measure $\mathbb{P}$ (Step 3)

To extend $\mathbb{P}_0$ into a measure, we will use Carathéodory's theorem:

Theorem 2.5.2 (Carathéodory's Extension Theorem)**.** *Given an algebra $\mathcal{A}$ over some set Ω and a probability pre-measure $\mathbb{P}_0 : \mathcal{A} \to [0, 1]$, there exists a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ such that $\mathcal{A} \subset \mathcal{F}$ and $\mathbb{P}|_{\mathcal{A}} = \mathbb{P}_0$. Furthermore, the σ-algebra $\mathcal{F}$ depends only on $\mathcal{A}$ and is minimal and unique, which we will also denote by $\sigma(\mathcal{A})$, and the probability measure $\mathbb{P}$ is unique.*

Proof Sketch. First, construct an outer measure by approximation with countable coverings. Then, show that the collection of sets that is measurable with respect to this outer measure is a σ-algebra $\mathcal{F}$ that contains $\mathcal{A}$. Finally, restricting the outer measure to this σ-algebra, one is then left with a probability space. To show minimality, one can show that $\mathcal{F}$ is contained in any σ-algebra that contains $\mathcal{A}$. Uniqueness is given by applying Dynkin's π-λ theorem (Theorem 3.2 in Billingsley, 1995).

Great care must be taken in each step involved in the outline above. To address these is well beyond the scope of this treatment and we refer the reader to the many excellent texts with a proof of this theorem, such as Chapter 12 in Royden (1988) and Chapter 11 in Billingsley (1995). ■

Applying Carathéodory's extension theorem to our cylinder algebra $\overline{\mathcal{C}}$ and pre-measure $\mathbb{P}_0$, we see that there exists a probability space $(\overline{\Sigma}^\infty, \sigma(\overline{\mathcal{C}}), \mathbb{P})$ over $\overline{\Sigma}^\infty$ that agrees with the LNM p_{LN}'s probabilities.

Phew! This now gets us to the fourth box in Fig. 2.4 and we only have one step remaining.

[23] Cantor's intersection theorem states that a decreasing sequence of nonempty compact sets have a nonempty intersection. A version of this result in introductory real analysis is the Nested Interval Theorem.

Defining a Sequence Model (Step 4)

We now have to make sure that the *outcome space* of the defined probability space fits the definition of a sequence model. That is, we have to find a way to convert (map) the infinite EOS-containing sequences from $\overline{\Sigma}^\infty$ into EOS-free finite or possibly infinite strings processed by a sequence model as required by Definition 2.5.2. We will achieve this through the use of a *random variable*.

Recall from Definition 2.2.5 that a random variable is a mapping between *two* sigma-algebras. Since we want our final measure space to work with the outcome space $\Sigma^\infty \cup \Sigma^*$, we, therefore, want to construct a σ-algebra over $\Sigma^* \cup \Sigma^\infty$ and then map elements from $\overline{\Sigma}^\infty$ to $\Sigma^* \cup \Sigma^\infty$ to have the appropriate objects. We will do so in a similar fashion as we constructed $(\overline{\Sigma}^\infty, \overline{\mathcal{C}})$. Given $\mathcal{H} \subset \Sigma^k$, define a rank-$k$ cylinder set in $\Sigma^* \cup \Sigma^\infty$ to be

$$\mathcal{C}(\mathcal{H}) \stackrel{\text{def}}{=} \{\boldsymbol{y}\boldsymbol{\omega} \ : \ \boldsymbol{y} \in \mathcal{H}, \boldsymbol{\omega} \in \Sigma^* \cup \Sigma^\infty\}. \tag{2.40}$$

Notice the major change from Eq. (2.27): the suffixes $\boldsymbol{\omega}$ of the elements in $C(\mathcal{H})$ now come from $\Sigma^* \cup \Sigma^\infty$ rather than $\overline{\Sigma}^\infty$. This means *(i)* that they do not contain EOS and *(ii)* that they (and thus, elements of $C(\mathcal{H})$) can also be finite. Let $\mathcal{C}_k$ be the set of all rank-k cylinder sets. Define $\mathcal{C} \stackrel{\text{def}}{=} \cup_{k=1}^\infty \mathcal{C}_k$. Then, $\sigma(\mathcal{C})$ is a σ-algebra by the same reasoning as in Lemma 2.5.1 and Theorem 2.5.2. We can now define the following random variable

$$\mathrm{x}(\boldsymbol{\omega}) = \begin{cases} \boldsymbol{\omega}_{<k} & \textbf{if } k \text{ is the first EOS in } \boldsymbol{\omega}, \\ \boldsymbol{\omega} & \textbf{otherwise } (\textbf{if } \text{EOS} \notin \boldsymbol{\omega}) \end{cases} \tag{2.41}$$

given any $\boldsymbol{\omega} \in \overline{\Sigma}^\infty$. The proposition below shows that x is well-defined.

Proposition 2.5.2. *The function* $\mathrm{x} : (\overline{\Sigma}^\infty, \sigma(\overline{\mathcal{C}})) \to (\Sigma^* \cup \Sigma^\infty, \sigma(\mathcal{C}))$ *defined in Eq.* (2.41) *is a measurable mapping.*

Proof. To show that x is measurable, it suffices to show the measurability of preimage of a generating set of the σ-algebra. Note that the set of thin cylinder sets is a generating set. Let $\mathcal{C}(\boldsymbol{y})$ be a thin cylinder set,

$$\begin{aligned} \mathrm{x}^{-1}(\mathcal{C}(\boldsymbol{y})) =& \mathrm{x}^{-1}(\{\boldsymbol{y}\boldsymbol{\omega} : \boldsymbol{\omega} \in \Sigma^* \cup \Sigma^\infty\}) && \text{(2.42a)} \\ =& \mathrm{x}^{-1}(\{\boldsymbol{y}\boldsymbol{\omega} : \boldsymbol{\omega} \in \Sigma^*\}) \cup \mathrm{x}^{-1}(\{\boldsymbol{y}\boldsymbol{\omega} : \boldsymbol{\omega} \in \Sigma^\infty\}) && \text{(2.42b)} \\ =& \left(\bigcup_{\boldsymbol{\omega} \in \Sigma^*} \overline{C}(\boldsymbol{y}\boldsymbol{\omega}\text{EOS}) \right) \cup \left(\overline{C}(\boldsymbol{y}) \cap \bigcap_{k=1}^\infty {\mathcal{A}_k}^{\mathsf{c}} \right) && \text{(2.42c)} \end{aligned}$$

Note that the sets $\mathcal{A}_k$ above are defined in Eq. (2.55) which are cylinder sets representing the event of terminating at step k. Then, from the derivation above, we can see that $\mathrm{x}^{-1}(\mathcal{C}(\boldsymbol{y}))$ is formed by countable operations over measurable sets (cylinder sets) in $\overline{\Sigma}^\infty$, and is hence measurable. So x is a measurable function. ■

x intuitively "cuts out" the first stretch of $\boldsymbol{\omega}$ before the first EOS symbol (where an LNM would stop generating) or leaves the sequence intact if there is

no termination symbol EOS. One can check that $\mathbb{P}_*$, defined using $\mathbb{P}$, is indeed a probability measure on $(\Sigma^* \cup \Sigma^\infty, \sigma(\mathcal{C}))$ and hence $(\Sigma^* \cup \Sigma^\infty, \sigma(\mathcal{C}), \mathbb{P}_*)$ is a probability space. We have therefore arrived at the final box of Fig. 2.4 and shown that, given any LNM, we can construct an associated sequence model as defined in Definition 2.5.2! In other words, given an LNM p_{LN}, we have constructed a sequence model p_{SM} (a probability space over $\Sigma^\infty \cup \Sigma^*$ where the probabilities assigned to (infinite) strings by p_{SM} agree with p_{LN}.

2.5.3 Interpreting the Constructed Probability Space

Under the formulation of a probability space together with a random variable, useful probability quantities arise naturally and intuitively.

Consider, for example, the probability of a single finite string $\boldsymbol{y} \in \Sigma^*$, $\mathbb{P}_*(\boldsymbol{y})$. By definition of x, this equals

$$\begin{aligned} \mathbb{P}_*(\boldsymbol{y}) &= \mathbb{P}_*(\mathrm{x} = \boldsymbol{y}) && (2.43)\\ &= \mathbb{P}\left(\mathrm{x}^{-1}(\boldsymbol{y})\right) && (2.44)\\ &= \mathbb{P}\left(\text{All the sequences } \boldsymbol{\omega} \in \overline{\Sigma}^\infty \text{ which map to } \boldsymbol{y}.\right) && (2.45) \end{aligned}$$

All the sequences $\boldsymbol{\omega} \in \overline{\Sigma}^\infty$ which map to $\boldsymbol{y}$ are sequences of the form $\boldsymbol{\omega} = \boldsymbol{y}\text{EOS}\boldsymbol{\omega}'$ for $\boldsymbol{\omega}' \in \overline{\Sigma}^\infty$—this is exactly the cylinder $\overline{C}(\boldsymbol{y}\text{EOS})$! By the definition of the probability space $\left(\overline{\Sigma}, \sigma\left(\overline{C}\right), \mathbb{P}\right)$, this is

$$\mathbb{P}\left(\overline{C}(\boldsymbol{y}\text{EOS})\right) = \sum_{\boldsymbol{y}' \in \{(\boldsymbol{y}\text{EOS})\}} p_{\text{LN}}\left(\boldsymbol{y}'\right) = p_{\text{LN}}(\boldsymbol{y}\text{EOS}) \tag{2.46}$$

and as before $p_{\text{LN}}(\boldsymbol{y}\text{EOS}) = \prod_{t=1}^{T} p_{\text{LN}}(y_t \mid \boldsymbol{y}_{<t})\, p_{\text{LN}}(\text{EOS} \mid \boldsymbol{y})$.

Altogether, this means that, given a finite string $\boldsymbol{y} \in \Sigma^*$, we intuitively have

$$\mathbb{P}_*(\mathrm{x} = \boldsymbol{y}) = p_{\text{LN}}(\text{EOS} \mid \boldsymbol{y}) p_{\text{LN}}(\boldsymbol{y}). \tag{2.47}$$

Additionally, as we will show in the next section, the probability of the set of infinite strings $\mathbb{P}_*(\mathrm{x} \in \Sigma^\infty)$ is the probability of generating an infinite string.

An important technical detail left out in this discussion so far is that both the singleton set $\{\boldsymbol{y}\}$ and Σ^∞ need to be measurable in $(\Sigma^* \cup \Sigma^\infty, \sigma(\mathcal{C}))$ for the above to make sense. This is addressed by Proposition 2.5.3 and Proposition 2.5.4.

Proposition 2.5.3. *In measure space $(\Sigma^* \cup \Sigma^\infty, \sigma(\mathcal{C}))$, $\{\boldsymbol{y}\}$ is measurable for all $\boldsymbol{y} \in \Sigma^*$.*

Proof. By definition in Eq. (2.40), for any $\boldsymbol{y} \in \Sigma^*$,

$$\begin{aligned} \mathcal{C}(\boldsymbol{y}) &= \{\boldsymbol{y}\boldsymbol{\omega} : \boldsymbol{\omega} \in \Sigma^* \cup \Sigma^\infty\} && (2.48a)\\ &= \{\boldsymbol{y}\boldsymbol{\omega} : \boldsymbol{\omega} \in \Sigma^*\} \cup \{\boldsymbol{y}\boldsymbol{\omega} : \boldsymbol{\omega} \in \Sigma^\infty\} && (2.48b) \end{aligned}$$

where

$$\{\boldsymbol{y}\boldsymbol{\omega} : \boldsymbol{\omega} \in \Sigma^*\} = \{\boldsymbol{y}\} \cup \bigcup_{a \in \Sigma} \{\boldsymbol{y}a\boldsymbol{\omega} : \boldsymbol{\omega} \in \Sigma^*\} \tag{2.49a}$$

and

$$\{\boldsymbol{y}\boldsymbol{\omega} : \boldsymbol{\omega} \in \Sigma^\infty\} = \bigcup_{a\in\Sigma} \{\boldsymbol{y}a\boldsymbol{\omega} : \boldsymbol{\omega} \in \Sigma^\infty\}. \tag{2.50}$$

So,

$$\begin{aligned} \mathcal{C}(\boldsymbol{y}) &= \{\boldsymbol{y}\} \cup \bigcup_{a\in\Sigma} \Big(\{\boldsymbol{y}a\boldsymbol{\omega} : \boldsymbol{\omega} \in \Sigma^*\} \cup \{\boldsymbol{y}a\boldsymbol{\omega} : \boldsymbol{\omega} \in \Sigma^\infty\} \Big) && \text{(2.51a)} \\ &= \{\boldsymbol{y}\} \cup \bigcup_{a\in\Sigma} \mathcal{C}(\boldsymbol{y}a) && \text{(2.51b)} \end{aligned}$$

which implies that $\{\boldsymbol{y}\} = \mathcal{C}(\boldsymbol{y}) \backslash \bigcup_{a\in\Sigma} \mathcal{C}(\boldsymbol{y}a)$ and hence measurable. ■

Proposition 2.5.4. *In the measure space* $(\Sigma^* \cup \Sigma^\infty, \sigma(\mathcal{C}))$*,* Σ^∞ *is measurable.*

Proof. First, the outcome space $\Sigma^* \cup \Sigma^\infty$ is measurable by definition of σ-algebra. Notice that

$$\Sigma^\infty = (\Sigma^* \cup \Sigma^\infty) \backslash \bigcup_{\boldsymbol{y}\in\Sigma^*} \{\boldsymbol{y}\}. \tag{2.52}$$

Since each $\{\boldsymbol{y}\}$ in the above is measurable by Proposition 2.5.3 and Σ^* is a countable set, Σ^∞ is then measurable. ■

Since both $\{\boldsymbol{y}\}$ and Σ^∞ are measurable in $(\Sigma^* \cup \Sigma^\infty, \sigma(\mathcal{C}))$ by Propositions 2.5.3 and 2.5.4, we have the following.

Proposition 2.5.5. *A sequence model* $(\Sigma^* \cup \Sigma^\infty, \sigma(\mathcal{C}), \mathbb{P})$ *is tight if and only if* $\sum_{\boldsymbol{y}\in\Sigma^*} \mathbb{P}(\{\boldsymbol{y}\}) = 1$.

Proof. By definition, a sequence model is tight if and only if $\mathbb{P}(\Sigma^\infty) = 0$. By Propositions 2.5.3 and 2.5.4, we can write

$$\begin{aligned} \mathbb{P}(\Sigma^* \cup \Sigma^\infty) &= \mathbb{P}(\Sigma^\infty) + \mathbb{P}(\Sigma^*) && \text{(countable additivity)} && \text{(2.53a)} \\ &= \mathbb{P}(\Sigma^\infty) + \sum_{\boldsymbol{y}\in\Sigma^*} \mathbb{P}(\{\boldsymbol{y}\}). && \text{(countable additivity)} && \text{(2.53b)} \end{aligned}$$

Hence, a sequence model is tight if and only if $\sum_{\boldsymbol{y}\in\Sigma^*} \mathbb{P}(\{\boldsymbol{y}\}) = 1$. ■

Deriving eos

As an aside, the preceding section allows us to motivate the EOS token in LNM as a construct that emerges naturally. Specifically, for any $\boldsymbol{y} \in \Sigma^*$, rearranging Eq. (2.47):

$$\begin{aligned} p_{\text{LN}}(\text{EOS} \mid \boldsymbol{y}) &= \frac{\mathbb{P}_*(\mathrm{x} = \boldsymbol{y})}{p_{\text{LN}}(\boldsymbol{y})} && \text{(2.54a)} \\ &= \frac{\mathbb{P}_*(\mathrm{x} = \boldsymbol{y})}{\mathbb{P}_*(\mathrm{x} \in \mathcal{C}(\boldsymbol{y}))} && \text{(2.54b)} \\ &= \mathbb{P}_*(\mathrm{x} = \boldsymbol{y} \mid \mathrm{x} \in \mathcal{C}(\boldsymbol{y})) && \text{(2.54c)} \end{aligned}$$

where we have used $p_{\text{LN}}(\boldsymbol{y}) = \mathbb{P}(\overline{C}(\boldsymbol{y})) = \mathbb{P}(\mathrm{x}^{-1}(\mathcal{C}(\boldsymbol{y}))) = \mathbb{P}_*(\mathrm{x} \in \mathcal{C}(\boldsymbol{y}))$. This means that the EOS probability in an LNM emerges as the conditional probability that, given that we must generate a string with a prefix $\boldsymbol{y} \in \Sigma^*$, the string is *exactly* $\boldsymbol{y}$, i.e., that generation ends there.

2.5.4 Characterizing Tightness

Now that we have derived a measure-theoretic formalization of the probability space induced by locally-normalized models, we can use it to provide an exact characterization of tightness in LNMs. First, we consider the event

$$\mathcal{A}_k \stackrel{\text{def}}{=} \{\boldsymbol{\omega} \in \overline{\Sigma}^\infty : \omega_k = \text{EOS}\} \tag{2.55}$$

in the probability space $(\overline{\Sigma}^\infty, \sigma(\overline{\mathcal{C}}), \mathbb{P})$. Intuitively, $\mathcal{A}_k$ is the event that an EOS symbol appears at position k in the string. Note that under this definition the $\mathcal{A}_k$ are not disjoint. For example, the string $\boldsymbol{\omega} = ab\,\text{EOS}\,c\,\text{EOS}\,dddd\cdots$ lives in the intersection of $\mathcal{A}_3$ and $\mathcal{A}_5$ since EOS appears at both position 3 and position 5. Using Eq. (2.55), we can express the event consisting of all finite strings as

$$\bigcup_{k=1}^{\infty} \mathcal{A}_k. \tag{2.56}$$

It follows that we can express the event of an infinite string as

$$\left(\bigcup_{k=1}^{\infty} \mathcal{A}_k\right)^{\mathsf{c}} = \bigcap_{k=1}^{\infty} {\mathcal{A}_k}^{\mathsf{c}}. \tag{2.57}$$

Thus, using the random variable x, we can express the probability of generating an infinite string as

$$\mathbb{P}_*(\mathrm{x} \in \Sigma^\infty) = \mathbb{P}(\mathrm{x}^{-1}(\Sigma^\infty)) \tag{2.58a}$$

$$= \mathbb{P}\left(\bigcap_{k=1}^{\infty} {\mathcal{A}_k}^{\mathsf{c}}\right). \tag{2.58b}$$

Hence, we can now restate and formalize the notion of tightness.

Definition 2.5.5. *A sequence model is said to be* ***tight*** *if* $\mathbb{P}_*(\mathrm{x} \in \Sigma^\infty) = 0$*, in which case it is also a language model. Otherwise, we say that it is* ***non-tight****.*

Note that the definition of $\mathcal{A}_k$ only uses a string's k-prefix, and hence is a cylinder set of rank k. Recalling that the cylinder sets are measurable and so are the sets countably generated by them, we see that both the event consisting of all finite strings and the event consisting of all infinite strings are measurable. Thus, $\mathbb{P}\left(\cup_{k=1}^{\infty}\mathcal{A}_k\right)$ and $\mathbb{P}\left(\cap_{k=1}^{\infty}{\mathcal{A}_k}^{\mathsf{c}}\right)$ are well defined.

A Lower Bound Result

We have characterized tightness in terms of the probability of a specific event $\mathbb{P}\left(\cap_{k=1}^{\infty}\mathcal{A}_k{}^{\mathsf{c}}\right)$, a quantity we now seek to determine.

Lemma 2.5.4. *If* $\sum_{n=2}^{\infty}\mathbb{P}\left(\mathcal{A}_n \mid \cap_{m=1}^{n-1}\mathcal{A}_m{}^{\mathsf{c}}\right) = \infty$, *then* $\mathbb{P}\left(\cap_{m=1}^{\infty}\mathcal{A}_m{}^{\mathsf{c}}\right) = 0$.

Proof. First, recall an elementary inequality that for $x > 0$,

$$x - 1 \geqslant \log x \quad \Leftrightarrow \quad 1 - x \leqslant \log\frac{1}{x}. \tag{2.59}$$

Note that $\mathbb{P}(\cap_{m=1}^{n}\mathcal{A}_m^{\mathsf{c}}) > 0$ for any n, for otherwise the conditional probabilities would be undefined. Let $p_n \stackrel{\text{def}}{=} \mathbb{P}(\cap_{m=1}^{n}\mathcal{A}_m^{\mathsf{c}})$. Then we have that $p_n > 0$ for all n, and

$$\begin{aligned}
\infty &= \sum_{n=2}^{\infty}\mathbb{P}(\mathcal{A}_n \mid \cap_{m=1}^{n-1}\mathcal{A}_m^{\mathsf{c}}) && \text{(2.60a)}\\
&= \sum_{n=2}^{\infty} 1 - \mathbb{P}(\mathcal{A}_n^{\mathsf{c}} \mid \cap_{m=1}^{n-1}\mathcal{A}_m^{\mathsf{c}}) && \text{(2.60b)}\\
&= \lim_{N\to\infty}\sum_{n=2}^{N} 1 - \mathbb{P}(\mathcal{A}_n^{\mathsf{c}} \mid \cap_{m=1}^{n-1}\mathcal{A}_m^{\mathsf{c}}) && \text{(2.60c)}\\
&\leqslant \lim_{N\to\infty}\sum_{n=2}^{N} \log 1/\mathbb{P}(\mathcal{A}_n^{\mathsf{c}} \mid \cap_{m=1}^{n-1}\mathcal{A}_m^{\mathsf{c}}) \qquad \text{(by Eq. (2.59))} && \text{(2.60d)}\\
&= \lim_{N\to\infty}\sum_{n=2}^{N} \log\frac{\mathbb{P}(\cap_{m=1}^{n-1}\mathcal{A}_m^{\mathsf{c}})}{\mathbb{P}(\cap_{m=1}^{n}\mathcal{A}_m^{\mathsf{c}})} && \text{(2.60e)}\\
&= \lim_{N\to\infty}\sum_{n=2}^{N} \log\frac{p_{n-1}}{p_n} && \text{(2.60f)}\\
&= \lim_{N\to\infty}\sum_{n=2}^{N} (\log p_{n-1} - \log p_n) && \text{(2.60g)}\\
&= \lim_{N\to\infty}(\log p_1 - \log p_N) && \text{(2.60h)}\\
&= \log p_1 - \lim_{N\to\infty}\log p_N && \text{(2.60i)}
\end{aligned}$$

which implies that

$$\begin{aligned}
&\lim_{N\to\infty}\log p_N = -\infty && \text{(2.61a)}\\
\Leftrightarrow\quad &\lim_{N\to\infty} p_N = 0 && \text{(2.61b)}\\
\Leftrightarrow\quad &\lim_{N\to\infty}\mathbb{P}(\cap_{m=1}^{N}\mathcal{A}_m^{\mathsf{c}}) = 0 && \text{(2.61c)}\\
\Leftrightarrow\quad &\mathbb{P}(\cap_{m=1}^{\infty}\mathcal{A}_m^{\mathsf{c}}) = 0. \qquad \text{(by continuity of measure)} && \text{(2.61d)}
\end{aligned}$$

■

Using Lemma 2.5.4, we can derive the following useful condition of tightness of a language model. Specifically, it applies when the probability of EOS is lower bounded by a function that depends only on the *length* and not the *content* of the prefix.

Proposition 2.5.6. *If* $p_{LN}(\text{EOS} \mid \boldsymbol{y}) \geqslant f(t)$ *for all* $\boldsymbol{y} \in \Sigma^t$ *and for all* t *and* $\sum_{t=1}^{\infty} f(t) = \infty$*, then* $\mathbb{P}(\cap_{k=1}^{\infty} \mathcal{A}_k{}^{\mathsf{c}}) = 0$*. In other words,* p_{LN} *is tight.*

Proof. Suppose $p_{\text{LN}}(\text{EOS} \mid \boldsymbol{y}) \geqslant f(t)$ for all $\boldsymbol{y} \in \Sigma^t$. To apply Lemma 2.5.4, we observe that

$$\mathcal{A}_n \cap (\mathcal{A}_1^{\mathsf{c}} \cap \cdots \cap \mathcal{A}_{n-1}^{\mathsf{c}}) = \{\boldsymbol{\omega} \in \overline{\Sigma}^{\infty} : \omega_n = \text{EOS}\} \cap \left(\bigcap_{i=1}^{n-1} \{\boldsymbol{\omega} \in \overline{\Sigma}^{\infty} : \omega_i \neq \text{EOS}\} \right) \tag{2.62a}$$

$$= \{\boldsymbol{\omega} \in \overline{\Sigma}^{\infty} : \boldsymbol{\omega} = \text{EOS}, \forall\ i < n, \boldsymbol{\omega} \neq \text{EOS}\} \tag{2.62b}$$

$$= \{\boldsymbol{\omega} \in \overline{\Sigma}^{\infty} : \boldsymbol{\omega}\text{'s first EOSis at position } n\} \tag{2.62c}$$

and similarly

$$\mathcal{A}_1^{\mathsf{c}} \cap \cdots \cap \mathcal{A}_{n-1}^{\mathsf{c}} = \{\boldsymbol{\omega} \in \overline{\Sigma}^{\infty} : \text{There is no EOSin } \boldsymbol{\omega}\text{'s first } n-1 \text{ positions}\} \tag{2.63}$$

Setting $\mathcal{G} \stackrel{\text{def}}{=} \{\boldsymbol{\omega}\text{EOS} : \boldsymbol{\omega} \in \Sigma^{n-1}\} \subset \overline{\Sigma}^n$, we get

$$\mathbb{P}(\mathcal{A}_n \mid \mathcal{A}_1^{\mathsf{c}} \cap \cdots \cap \mathcal{A}_{n-1}^{\mathsf{c}}) = \frac{\mathbb{P}(\mathcal{A}_n \cap (\mathcal{A}_1^{\mathsf{c}} \cap \cdots \cap \mathcal{A}_{n-1}^{\mathsf{c}}))}{\mathbb{P}(\mathcal{A}_1^{\mathsf{c}} \cap \cdots \cap \mathcal{A}_{n-1}^{\mathsf{c}})} \tag{2.64a}$$

$$= \frac{\mathbb{P}(\overline{C}(\mathcal{G}))}{\mathbb{P}(\overline{C}(\Sigma^{n-1}))} \qquad \text{(definition of } \mathcal{G}\text{)} \tag{2.64b}$$

$$= \frac{\sum_{\boldsymbol{\omega} \in \Sigma^{n-1}} p(\text{EOS} \mid \boldsymbol{\omega}) p(\boldsymbol{\omega})}{\sum_{\boldsymbol{\omega} \in \Sigma^{n-1}} p(\boldsymbol{\omega})} \qquad \text{(by Eq. (2.30))} \tag{2.64c}$$

$$\geqslant \frac{\sum_{\boldsymbol{\omega} \in \Sigma^{n-1}} f(n-1) p(\boldsymbol{\omega})}{\sum_{\boldsymbol{\omega} \in \Sigma^{n-1}} p(\boldsymbol{\omega})} \qquad \text{(definition of } f(t)\text{)} \tag{2.64d}$$

$$= f(n-1) \frac{\sum_{\boldsymbol{\omega} \in \Sigma^{n-1}} p(\boldsymbol{\omega})}{\sum_{\boldsymbol{\omega} \in \Sigma^{n-1}} p(\boldsymbol{\omega})} \tag{2.64e}$$

$$= f(n-1). \tag{2.64f}$$

Since $\sum_{t=0}^{\infty} f(t) = \infty$, Lemma 2.5.4 shows that the event of a string never terminating, i.e., $\cap_{k=1}^{\infty} \mathcal{A}_k^{\mathsf{c}}$ has probability measure $\mathbb{P}(\cap_{k=1}^{\infty} \mathcal{A}_k^{\mathsf{c}}) = 0$. In other words, if the EOS probability of a language model is lower bounded by a divergent sequence at every step, then the event that this language model terminates has probability 1. ■

The Borel–Cantelli Lemmata

It turns out that Proposition 2.5.6 admits a converse statement in which we can prove a similar property of p_{LN} by assuming that the model is tight. To show this result, we will use a fundamental inequality from probability theory—the Borel–Cantelli lemmata. The Borel–Cantelli lemmata are useful for our purposes because they relate the probability measure of sets of the form $\bigcap_{n=0}^{\infty} \mathcal{A}_n$ or $\bigcup_{n=0}^{\infty} \mathcal{A}_n$ to a series $\sum_{n=0}^{\infty} p_n$. We will only state the lemmata here without supplying their proofs;[24] however, we point out that Lemma 2.5.4 can be viewed as a parallel statement to the Borel–Cantelli lemmata and one can prove the lemmata using a very similar proof (cf. proof of Theorem 2.3.7 in Durrett, 2019).

Concretely, given a sequence of events $\{\mathcal{A}_n\}_{n=1}^{\infty}$ in some probability space, the Borel–Cantelli lemmata are statements about the event

$$\{\mathcal{A}_n \text{ i.o.}\} \stackrel{\text{def}}{=} \bigcap_{m=1}^{\infty} \bigcup_{n=m}^{\infty} \mathcal{A}_n \tag{2.65}$$

where i.o. stands for "infinitely often." Intuitively, $\{\mathcal{A}_n \text{ i.o.}\}$ is the set of outcomes that appear in infinitely many sets in the collection $\{\mathcal{A}_n\}_{n=1}^{\infty}$—they are the events that always remain in the union of an infinite family of sets no matter how many of the leading ones we remove (hence the name). We will not use Borel–Cantelli directly, but they offer a probabilistic proof of a key result (Corollary 2.5.1) which will in turn lead to the desired statement about tightness. We formally state the first and second Borel–Cantelli lemmata below.

Lemma 2.5.5 (Borel–Cantelli I)**.** *If* $\sum_{n=1}^{\infty} \mathbb{P}(\mathcal{A}_n) < \infty$*, then* $\mathbb{P}(\mathcal{A}_n \text{ i.o.}) = 0$.

Lemma 2.5.6 (Borel–Cantelli II)**.** *Assume* $\{\mathcal{A}_n\}$ *is a sequence of independent events, then* $\sum_{n=1}^{\infty} \mathbb{P}(\mathcal{A}_n) = \infty \Rightarrow \mathbb{P}(\mathcal{A}_n \text{ i.o.}) = 1$.

Using the Borel–Cantelli lemmata, we can prove the following useful fact.

Corollary 2.5.1. *Given a sequence* $\{p_n\}$ *where* $p_n \in [0, 1)$*. Then,*

$$\prod_{n=1}^{\infty} (1 - p_n) = 0 \iff \sum_{n=1}^{\infty} p_n = \infty. \tag{2.66}$$

To show Corollary 2.5.1, we first show the following simple consequence of Borel–Cantelli.

Corollary 2.5.2. *If* $\mathbb{P}(\mathcal{A}_n \text{ i.o.}) = 1$*, then* $\sum_{n=1}^{\infty} \mathbb{P}(\mathcal{A}_n) = \infty$.

Proof. Suppose to the contrary that $\sum_{n=1}^{\infty} \mathbb{P}(\mathcal{A}_n) < \infty$, then, by Borel–Cantelli I (Lemma 2.5.5), $\mathbb{P}(\mathcal{A}_n \text{ i.o.}) = 0$, which contradicts the assumption. Hence, $\sum_{n=1}^{\infty} \mathbb{P}(\mathcal{A}_n) = \infty$. ■

[24] See §2.3 in Durrett (2019) or §4 in Billingsley (1995) instead.

Proof. We can use a product measure to construct a sequence of independent events $\{\mathcal{A}_n\}_{n=1}^{\infty}$ such that $\mathbb{P}(\mathcal{A}_n) = p_n$. (The product measure ensures independence.) Then, by definition in Eq. (2.65),

$$\{\mathcal{A}_n \text{ i.o.}\}^{\complement} = \bigcup_{m=1}^{\infty} \bigcap_{n \geqslant m} \mathcal{A}_n^{\complement} \tag{2.67}$$

So,

$$1 - \mathbb{P}(\mathcal{A}_n \text{ i.o.}) = \mathbb{P}\left(\bigcup_{m} \bigcap_{n \geqslant m} \mathcal{A}_n^{\complement}\right) \tag{2.68a}$$

$$= \lim_{m \to \infty} \mathbb{P}\left(\bigcap_{n \geqslant m} \mathcal{A}_n^{\complement}\right) \tag{2.68b}$$

$$= \lim_{m \to \infty} \prod_{n \geqslant m} \mathbb{P}(\mathcal{A}_n^{\complement}) \qquad (\mathcal{A}_n \text{ are independent by construction}) \tag{2.68c}$$

$$= \lim_{m \to \infty} \prod_{n \geqslant m} (1 - p_n) \tag{2.68d}$$

($\Rightarrow$): Assume $\prod_{n=1}^{\infty}(1 - p_n) = 0$. Then, for any m,

$$0 = \prod_{n \geqslant 1}(1 - p_n) = \underbrace{\left(\prod_{1 \leqslant n < m} (1 - p_n)\right)}_{>0} \left(\prod_{n \geqslant m} (1 - p_n)\right) \tag{2.69}$$

So it must the case that, for any m, $\prod_{n \geqslant m}(1 - p_n) = 0$. Therefore,

$$1 - \mathbb{P}(\mathcal{A}_n \text{ i.o.}) = \lim_{m \to \infty} \prod_{n \geqslant m} (1 - p_n) = 0 \tag{2.70}$$

which implies $\mathbb{P}(\mathcal{A}_n \text{ i.o.}) = 1$. Corollary 2.5.2 implies that $\sum_{n=1}^{\infty} p_n = \infty$.

($\Leftarrow$): Assume $\sum_{n=1}^{\infty} p_n = \infty$. Then by Borel–Cantelli II (Lemma 2.5.6), $\mathbb{P}(\mathcal{A}_n \text{ i.o.}) = 1$ which implies

$$0 = 1 - \mathbb{P}(\mathcal{A}_n \text{ i.o.}) = \lim_{m \to \infty} \prod_{n \geqslant m} (1 - p_n) \tag{2.71}$$

Observe that $\left\{ \prod_{n \geqslant m}(1 - p_n) \right\}_m$ is a non-decreasing sequence in m; to see this, note that as m grows larger we multiply strictly fewer values $(1 - p_n) \in (0, 1]$. However, since we know the sequence is non-negative and tends to 0, it follows that for *any* m, we have

$$\prod_{n \geqslant m} (1 - p_n) = 0. \tag{2.72}$$

It follows that, for any m, we have

$$\prod_{n=1}^{\infty}(1-p_n) = \prod_{n<m}(1-p_n)\underbrace{\prod_{n\geqslant m}(1-p_n)}_{=0} = \prod_{n<m}(1-p_n)\cdot 0 = 0. \tag{2.73}$$

■

We now turn to proving a more general version of Proposition 2.5.6, which would imply its converse. First, we define the following quantity

$$\tilde{p}_{\text{EOS}}(t) \stackrel{\text{def}}{=} \mathbb{P}(\mathcal{A}_t \mid \mathcal{A}_1^{\mathsf{c}} \cap \cdots \cap \mathcal{A}_{t-1}^{\mathsf{c}}) \tag{2.74}$$

which can be viewed as the EOSprobability at step t, given that EOSwas not generated at any earlier step. One can also show that, when $\tilde{p}_{\text{EOS}}(t)$ is defined, it has the same value as

$$\tilde{p}_{\text{EOS}}(t) = \frac{\sum_{\boldsymbol{\omega}\in\Sigma^{t-1}} p_{\text{LN}}(\boldsymbol{\omega}) p_{\text{LN}}(\text{EOS} \mid \boldsymbol{\omega})}{\sum_{\boldsymbol{\omega}\in\Sigma^{t-1}} p_{\text{LN}}(\boldsymbol{\omega})}, \tag{2.75}$$

which one can see as the weighted average probability of terminating at a string of length t.

We can now completely characterize the tightness of an LNM with the following theorem.

Theorem 2.5.3. *An LNM is tight if and only if $\tilde{p}_{\text{EOS}}(t) = 1$ for some t or $\sum_{t=1}^{\infty}\tilde{p}_{\text{EOS}}(t) = \infty$.*

Proof. Recall the definition of $\tilde{p}_{\text{EOS}}$, as previously defined in Eq. (2.74), is

$$\tilde{p}_{\text{EOS}}(t) \stackrel{\text{def}}{=} \mathbb{P}(\mathcal{A}_t \mid \mathcal{A}_1^{\mathsf{c}} \cap \cdots \cap \mathcal{A}_{t-1}^{\mathsf{c}}). \tag{2.76}$$

Case 1. Suppose that $\tilde{p}_{\text{EOS}}(t) < 1$ for all t. Consider the termination probability again:

$$\mathbb{P}\left(\bigcap_{t=1}^{\infty}\mathcal{A}_t{}^{\mathsf{c}}\right) = \lim_{T\to\infty}\mathbb{P}\left(\bigcap_{t=1}^{T}\mathcal{A}_t{}^{\mathsf{c}}\right) \tag{2.77a}$$

$$= \lim_{T\to\infty}\prod_{t=1}^{T}\mathbb{P}(\mathcal{A}_t{}^{\mathsf{c}} \mid \mathcal{A}_1^{\mathsf{c}} \cap \cdots \cap \mathcal{A}_{t-1}^{\mathsf{c}}) \tag{2.77b}$$

$$= \lim_{T\to\infty}\prod_{t=1}^{T}(1-\tilde{p}_{\text{EOS}}(t)) \tag{2.77c}$$

$$= \prod_{t=1}^{\infty}(1-\tilde{p}_{\text{EOS}}(t)). \tag{2.77d}$$

In the above, we have assumed that $\mathbb{P}(\mathcal{A}_1^{\mathsf{c}} \cap \cdots \cap \mathcal{A}_t^{\mathsf{c}}) > 0$ for all t, which is true by assumption that $\tilde{p}_{\text{EOS}}(t) < 1$.. Hence, by Corollary 2.5.1, Eq. (2.77d) is 0 if and only if $\sum_t \tilde{p}_{\text{EOS}}(t) = \infty$.

Case 2. If $\widetilde{p}_{\text{EOS}}(t) = 1$ is true for some $t = t_0$, then $\mathbb{P}(\mathcal{A}_1^{\complement} \cap \cdots \cap \mathcal{A}_{t_0}^{\complement}) = 0$ and hence $\mathbb{P}\left(\bigcap_{t=1}^{\infty} \mathcal{A}_t{}^{\complement}\right) = 0$ and such a language model is guaranteed to terminate at t_0. ■

The first condition intuitively says that there exists a step t at which the LNM will stop with probability 1. If the first case of the condition does not hold, the second case can be checked since its summands will be well-defined (the conditional probabilities in Eq. (2.75) will not divide by 0). We remark that Theorem 2.5.3 is a generalization of Proposition 2.5.6 since if $\tilde{p}_{\text{EOS}}(t)$ is lower-bounded by $f(t)$ whose series diverges, its own series would also diverge. However, since $\tilde{p}_{\text{EOS}}(t)$ involves the computation of a partition function in its denominator, it is most likely intractable to calculate (Lin et al., 2021; Lin and McCarthy, 2022). Hence, Proposition 2.5.6 will be the main tool for determining tightness when we explore concrete language modeling frameworks later.

We have now very thoroughly defined the notion of language model tightness and provided sufficient and necessary conditions for an LNM or a sequence model to be tight. In the next sections, we start our exploration of concrete computational models of language, from the very simple and historically important finite-state language models, their neural variants, to the modern Transformer architectures. For each of them, we will also individually discuss their tightness results and conditions.

Chapter 3

Modeling Foundations

The previous chapter introduced the fundamental measure-theoretic characteristics of language modeling. We will revisit those over and over as they will serve as the foundations on which subsequent concepts are built.

In this chapter, we turn our attention to *modeling* foundations, that is, the decisions we face when we want to *build* a distribution over strings and *learn* the appropriate parameters for that distribution. We first discuss *how* to parameterize a distribution over strings (§3.1), what it means to *learn* good parameters, and how this can be done with modern optimization techniques and objectives (§3.2).

Continuing our framing of the notes in terms of questions, we will try to address the following:

Question 3.0.1. *How can a sequence model be parameterized?*

We introduce a more formal definition of a "parametrized model" later. For now, you can simply think of it as a function $p_{\boldsymbol{\theta}} : \overline{\Sigma}^* \to \mathbb{R}$ described by some *free parameters* $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ from a parameter space $\boldsymbol{\Theta}$. This means that the values that $p_{\boldsymbol{\theta}}$ maps its inputs to might depend on the choice of the parameters $\boldsymbol{\theta}$—the presence of parameters in a model, therefore, allows us to *fit* them, which in our context specifically, means choosing them to maximize some objective with respect to data. This raises the following question:

Question 3.0.2. *Given a parameterized model and a dataset, how can model parameters be chosen to reflect the dataset as well as possible?*

We begin with Question 3.0.1.

3.1 Representation-based Language Models

Most modern language models are defined as locally normalized models. However, in order to define locally normalized language model, we first define a sequence model $p_{\text{SM}}(\overline{y} \mid \overline{\boldsymbol{y}})$. Then, we prove that the specific parameterization used in $p_{\text{SM}}(\overline{y} \mid \overline{\boldsymbol{y}})$ encodes a tight locally normalized language model. However, as we demonstrated in Example 2.5.1, not all sequence models encode tight locally normalized language models in the sense of definition Definition 2.5.1. So far, however, we have only talked about this process abstractly. For example, we have proven that every language model can be locally normalized and we have also given necessary and sufficient conditions for when a sequence model encodes a tight locally normalized language model. In this section, we start making the abstraction more concrete by considering a very general framework for parameterizing a locally normalized language model through sequence models $p_{\text{SM}}(\overline{y} \mid \overline{\boldsymbol{y}})$. We will call this the **representation-based language modeling** framework.

In the representation-based language modeling framework, each conditional distribution in a sequence model $p_{\text{SM}}(\overline{y} \mid \overline{\boldsymbol{y}})$ directly models the probability of the next symbol $\overline{\boldsymbol{y}} \in \overline{\Sigma}$ given the context $\overline{\boldsymbol{y}}$—in other words, it tells us how likely $\overline{y}$ is appears in the context of $\overline{\boldsymbol{y}}$.[1] For example, given the string $\overline{\boldsymbol{y}} =$ "Papa eats caviar with a", we would like $p_{\text{LN}}(\overline{y} \mid \overline{\boldsymbol{y}})$ to capture that "spoon" is more likely than "fork" At the same time, since eating caviar with a fork is technically possible, we would also like $p_{\text{LN}}(\overline{y} \mid \overline{\boldsymbol{y}})$ to capture that "fork" is likelier than, for example, "pencil".

However, it is not *a-priori* clear how we should model $p_{\text{SM}}(\overline{y} \mid \overline{\boldsymbol{y}})$ concretely. We want to define a function that can map contexts $\overline{\boldsymbol{y}}$ to a distribution over possible continuations $\overline{y}$ with the caveat that this distribution can be easily adjusted, i.e., we can optimize its parameters with some objective in mind (cf. §3.2). We will do this by adopting a very general idea of defining $p_{\text{SM}}(\overline{y} \mid \overline{\boldsymbol{y}})$ in terms of similarity between representations that represent the symbol $\overline{y}$ and the context $\overline{\boldsymbol{y}}$. The more compatible the symbol $\overline{y}$ is with the context $\overline{\boldsymbol{y}}$, the more probable it should be. Intuitively, going from the example above, this means that "spoon" should be more similar to "Papa eats caviar with a" than "fork" should be, and that should still be more similar than "pencil". On the other hand, notice that this also means that "spoon" and "fork" should be closer together than any of them to "pencil".

One possibility for doing this is by *embedding* individual symbols $\overline{y}$ and all possible contexts $\overline{\boldsymbol{y}}$ as vectors in a Hilbert space, i.e., a complete vector space endowed with an inner product. Once we embed the symbols and contexts in such a space, we can talk about how similar they are. We will first describe how this can be done abstractly **??** and then discuss how exactly vector representations can be used when defining *discrete* probability distributions over the symbols

[1] Unless explicitly stated otherwise, we use the phrase "in the context of" to imply given *prior* context—i.e., when discussing probability distributions, this refers to the distribution $p_{\text{SM}}(\overline{y}_t \mid \overline{\boldsymbol{y}}_{<t})$ with $\overline{\boldsymbol{y}} = \overline{y}_t$. We will also see examples of models which specify the conditional probabilities in terms of symbols that do not necessarily appear before the current one.

in §3.1.3 by taking into account the notion of similarities between vectors. We discuss methods for *learning* representations later in this chapter (§3.2) and in Chapter 5.

3.1.1 Vector Space Representations

It is not immediately obvious how to measure the similarity or compatibility between two symbols, two contexts or a symbol and a context. However, such a notion is required as part of our intuitive desiderata for $p_{\text{SM}}(\overline{y} \mid \overline{\boldsymbol{y}})$. We begin by stating an important guiding principle, which we describe in detail next and use heavily throughout the rest of the notes.

Principle 3.1.1 (Representation Learning)**.** *The* ***good representation principle*** *states that the success of a machine learning model depends—in great part—on the representation that is chosen (or learned) for the objects that are being modeled. In the case of language modeling, the two most salient choice points are the representations chosen for the symbols, elements of $\overline{\Sigma}$, and the representations chosen for the contexts, elements of $\overline{\Sigma}^*$.*

Learning vector representations from data where individual entities are represented in some **representation space** (i.e., a Hilbert space) has a rich history in NLP and machine learning in general (Bengio et al., 2013).

To discuss the representations of symbols and strings more formally, we first introduce the notion of a **Hilbert space**, which leads us to a useful geometric manner to discuss the similarity and compatibility of symbols and contexts. We first start with some more basic definitions. A vector space over a field $\mathbb{F}$ is a set $\mathbb{V}$ together with two binary operations that satisfy certain axioms. The elements of $\mathbb{F}$ are often referred to as **scalars** and the elements of $\mathbb{V}$ as **vectors**. The two operations in the definition of a vector space are the *addition of vectors* and *scalar multiplication of vectors*.

Definition 3.1.1 (Vector space)**.** *A* ***vector space*** *over a field $\mathbb{F}$ is a set $\mathbb{V}$ together with two binary operations that satisfy the following axioms:*

1. ***Associativity*** *of vector addition: for all $\mathbf{v}, \mathbf{u}, \mathbf{q} \in \mathbb{V}$*
$$(\mathbf{v} + \mathbf{u}) + \mathbf{q} = \mathbf{v} + (\mathbf{u} + \mathbf{q}) \tag{3.1}$$
2. ***Commutativity*** *of vector addition: for all $\mathbf{v}, \mathbf{u} \in \mathbb{V}$*
$$\mathbf{v} + \mathbf{u} = \mathbf{u} + \mathbf{v} \tag{3.2}$$
3. ***Identity*** *element of vector addition: there exists $\mathbf{0} \in \mathbb{V}$ such that for all $\mathbf{v} \in \mathbb{V}$*
$$\mathbf{v} + \mathbf{0} = \mathbf{v} \tag{3.3}$$
4. ***Inverse*** *elements of vector addition: for every $\mathbf{v} \in \mathbb{V}$ there exists a $-\mathbf{v} \in \mathbb{V}$ such that*
$$\mathbf{v} + (-\mathbf{v}) = \mathbf{0} \tag{3.4}$$

5. ***Compatibility*** *of scalar multiplication with field multiplication: for all* $\mathbf{v} \in \mathbb{V}$ *and* $x, y \in \mathbb{F}$

$$x\,(y\mathbf{v}) = (xy)\,\mathbf{v} \tag{3.5}$$

6. ***Identity*** *element of scalar multiplication: for all* $\mathbf{v} \in \mathbb{V}$

$$1\mathbf{v} = \mathbf{v} \tag{3.6}$$

where 1 *is the multiplicative identity in* $\mathbb{F}$*.*

7. ***Distributivity*** *of scalar multiplication with respect to vector addition: for all* $x \in \mathbb{F}$ *and all* $\mathbf{u}, \mathbf{v} \in \mathbb{V}$

$$x\,(\mathbf{v} + \mathbf{u}) = x\mathbf{v} + x\mathbf{u} \tag{3.7}$$

8. ***Distributivity*** *of scalar multiplication with respect to field addition: for all* $x, y \in \mathbb{F}$ *and all* $\mathbf{v} \in \mathbb{V}$

$$(x + y)\,\mathbf{v} = x\mathbf{v} + y\mathbf{v} \tag{3.8}$$

In almost all practical cases, $\mathbb{F}$ will be $\mathbb{R}$ and $\mathbb{V}$ will be $\mathbb{R}^D$ for some $D \in \mathbb{N}$.

An important characteristic of a vector space is its **dimensionality**, which, informally, corresponds to the number of independent directions—**basis vectors**—in the space. Any $\mathbf{v} \in \mathbb{V}$ can be expressed as a *linear combination* of the D basis vectors. The coefficients of this linear combination can then be combined into a D-dimensional **coordinate vector** in $\mathbb{F}^D$. Vector spaces, therefore, allow us to talk about their elements in terms of their expressions with respect to the basis vectors. Inner product spaces additionally define an **inner product**, mapping pairs of elements of the vector space to scalars. More formally, it is a vector space together with a map $\langle \cdot, \cdot \rangle$ (the inner product) defined as follows.

Definition 3.1.2 (Inner product space)**.** *An* ***inner product space*** *is a vector space* $\mathbb{V}$ *over a field* $\mathbb{F}$ *coupled with a map*

$$\langle \cdot, \cdot \rangle : \mathbb{V} \times \mathbb{V} \to \mathbb{F} \tag{3.9}$$

such that the following axioms hold

1. ***Conjugate symmetry****: for all* $\mathbf{v}, \mathbf{u} \in \mathbb{V}$

$$\langle \mathbf{v}, \mathbf{u} \rangle = \overline{\langle \mathbf{u}, \mathbf{v} \rangle} \tag{3.10}$$

where $\overline{x}$ *denotes the* ***conjugate*** *of the element* $x \in \mathbb{F}$*.*

2. ***Linearity*** *in the first argument: for all* $\mathbf{v}, \mathbf{u}, \mathbf{z} \in \mathbb{V}$ *and* $x, y \in \mathbb{F}$

$$\langle x\mathbf{v} + y\mathbf{u}, \mathbf{z} \rangle = x\langle \mathbf{v}, \mathbf{z} \rangle + y\langle \mathbf{u}, \mathbf{z} \rangle \tag{3.11}$$

3. ***Positive-definiteness****: for all* $\mathbf{v} \neq 0$

$$\langle \mathbf{v}, \mathbf{v} \rangle > 0 \tag{3.12}$$

Inner products are often defined such that they capture some notion of similarity of the vectors in $\mathbb{V}$. We will use this when formally defining $p_{\text{SM}}(\overline{y} \mid \overline{\boldsymbol{y}})$ in §3.1.2.

Every inner product on a real or complex vector space induces a vector norm defined as follows.

Definition 3.1.3 (Norm)**.** *Given a vector space $\mathbb{V}$ over $\mathbb{R}$ or $\mathbb{C}$ and an inner product $\langle \cdot, \cdot \rangle$ over it, the **norm** induced by the inner product is defined as the function $\|\cdot\| : \mathbb{V} \to \mathbb{R}_{\geqslant 0}$ where*

$$\|\mathbf{v}\| \stackrel{\text{def}}{=} \sqrt{\langle \mathbf{v}, \mathbf{v} \rangle}. \tag{3.13}$$

A Hilbert space is then an inner product space in which all sequences of elements satisfy a useful property with respect to the norm defined by the inner product: every convergent series with respect to the norm converges to a vector in $\mathbb{V}$.

Definition 3.1.4 (Hilbert space)**.** *A **Hilbert space** is an inner product space that is **complete** with respect to the norm defined by the inner product. An inner product space is complete with respect to the norm if every Cauchy sequence (an absolutely convergent sequence, i.e., a sequence whose elements become arbitrarily close to each other) converges to an element in $\mathbb{V}$. More precisely, an inner product space is complete if, for every series*

$$\sum_{n=1}^{\infty} \mathbf{v}_n \tag{3.14}$$

such that

$$\sum_{n=1}^{\infty} \|\mathbf{v}_n\| < \infty, \tag{3.15}$$

it holds that

$$\sum_{n=1}^{\infty} \mathbf{v}_n \in \mathbb{V}. \tag{3.16}$$

Note that even if an inner product space $\mathbb{V}$ is not necessarily a Hilbert space, $\mathbb{V}$ can always be *completed* to a Hilbert space.

Theorem 3.1.1 (Completion theorem for inner product spaces)**.** *Any inner product space can be* completed *into a Hilbert space.*

We omit the proof for this theorem. More precisely, the inner product space can be completed into a Hilbert space by completing it with respect to the norm induced by the inner product on the space. For this reason, inner product spaces are also called pre-Hilbert spaces.

To motivate our slightly more elaborate treatment of representation spaces, we consider an example of a model which falls under our definition of a representation-based language model but would be ill-defined if it worked under any space with fewer axioms than a Hilbert space.

Space	Utility
Vector space	A space in which representations of symbols and string live. It also allows the expression of the vector representations in terms of the basis vectors.
Inner product space	Defines an inner product, which defines a norm and can measure similarity.
Hilbert space	There are no "holes" in the representation space with respect to the defined norm, since all convergent sequences converge into $\mathbb{V}$.

Table 3.1: The utility of different spaces introduced in this section.

Example 3.1.1 (A series of representations)**.** *Recurrent neural networks are a type of neural network that sequentially process their input and compute the output (context representation) at time step t based on the output at time step $t-1$: $\mathbf{h}_t = \mathbf{f}(\mathbf{h}_{t-1}, y_t)$. A formal definition of a recurrent neural network, which we subsequently in §5.1.2, is not required at the moment. However, note that a recurrent neural network with one-dimensional representations h could, for example, take the specific form*

$$h_t = \frac{1}{2}h_{t-1} + \frac{1}{h_{t-1}} \tag{3.17}$$

with $h_0 = 2$.

Suppose we chose the inner product space $\mathbb{Q}$ over the field $\mathbb{Q}$ for our representation space. All elements of the sequence h_t are indeed rational numbers. However, the limit of the sequence, which can be shown to be $\sqrt{2}$, is not *in the inner product space! This shows that $\mathbb{Q}$ is not a Hilbert space and that we must, in full generality, work with Hilbert spaces whenever we are dealing with possibly infinite sequences of data. The reason this is especially relevant for language modeling is the need to consider arbitrarily long strings (contexts), whose representations we would like to construct in a way similar to Eq. (3.17). Such representations can, therefore, approach a limiting representation* outside *the space whenever the representation space does not satisfy the axioms of a Hilbert space.*

A summary of the utilities of the three algebraic spaces introduced in this subsection is summarized in Tab. 3.1.

Representation Functions

We can now introduce the notion of a general representation function.

Definition 3.1.5 (Representation function)**.** *Let $\mathcal{S}$ be a set and $\mathbb{V}$ a Hilbert space over some field $\mathbb{F}$. A* ***representation function*** $\mathbf{f}$ *for the elements of $\mathcal{S}$ is a function of the form*

$$\mathbf{f}\colon \mathcal{S} \mapsto \mathbb{V}. \tag{3.18}$$

The dimensionality of the Hilbert space of the representations, D, is determined by the modeler. In NLP, D usually ranges between 10 to 10000.

Importantly, in the case that $\mathcal{S}$ is finite, we can represent a representation function as a *matrix* $\mathbf{E} \in \mathbb{R}^{|\mathcal{S}| \times D}$ (assuming $\mathbb{V} = \mathbb{R}^D$ where the n^{th} row corresponds to the representation of the n^{th} element of $\mathcal{S}$. This method for representing $\mathbf{f}$ is both more concise and will be useful for integrating the symbol representation function into a model, where matrix multiplications are often the most efficient way to implement such functions on modern hardware.

This is the case for the representations of the individual symbols $\overline{y}$ from $\overline{\Sigma}$, where the representation function, which we will denote as $\mathbf{e}(\cdot)$, is implemented as a lookup into the embedding matrix $\mathbf{E} \in \mathbb{R}^{|\overline{\Sigma}| \times D}$, i.e., $\mathbf{e}(\overline{y}) = \mathbf{E}_{\overline{y}}$.[2] In this case, we will also refer to $\mathbf{e}(\cdot)$ as the embedding function.

Definition 3.1.6 (Symbol embedding function)**.** *Let Σ be an alphabet. An* ***embedding function*** $\mathbf{e}(\cdot) \colon \overline{\Sigma} \to \mathbb{R}^D$ *is a representation function of individual symbols $\overline{y} \in \overline{\Sigma}$.*

The representations $\mathbf{e}(\overline{y})$ are commonly referred to as **embeddings**, but, for consistency, we will almost exclusively use the term representations in this text. Let us first consider possibly the simplest way to represent discrete symbols with real-valued vectors: **one-hot encodings**.

Example 3.1.2 (One-hot encodings)**.** *Let $n : \overline{\Sigma} \to \{1, \ldots, |\overline{\Sigma}|\}$ be a bijection (i.e., an ordering of the alphabet, assigning an index to each symbol in Σ). A one-hot encoding $\mathbf{o}$. is a representation function which assigns the symbol $\overline{y} \in \overline{\Sigma}$ the $n(\overline{y})^{th}$ basis vector:*

$$\mathbf{o}_y \stackrel{\text{def}}{=} \mathbf{d}_{n(y)}, \tag{3.19}$$

where here $\mathbf{d}_n$ is the n^{th} canonical basis vector, i.e., a vector of zeros with a 1 at position n.

While one-hot encodings are an easy way to create vector representations of symbols, they have a number of drawbacks. First, these representations are relatively large—we have $D = |\overline{\Sigma}|$—and *sparse*, since only one of the dimensions is non-zero. Second, such representations are not ideal for capturing the variation in the *similarity* between different words. For example, the cosine similarity—a metric we will motivate in the next section for measuring the similarity between symbol representations—between symbols' one-hot encodings is zero for all non-identical symbols. Ideally, we would like symbol representations to encode semantic information, in which case, a metric such as cosine similarity could be used to quantify semantic similarity. This motivates the use of more complex representation functions, which we subsequently discuss.

While most systems use this standard way of defining individual symbol representations using the embedding matrix, the way that the *context* is encoded (and what even is considered as context) is really the major difference between the different architectures which we will consider later. Naturally, since the set

[2] Here, we use the notation $\mathbf{E}_{\overline{y}}$ to refer to the lookup of the row in $\mathbf{E}$ corresponding to $\overline{y}$.

of all contexts is infinite, we cannot simply represent the representation function with a matrix. Rather, we define the representation of a context $\overline{\boldsymbol{y}}$ through an encoding function.

Definition 3.1.7 (Context encoding function)**.** *Let Σ be an alphabet. A* ***context encoding function*** $\mathsf{enc}(\cdot) : \overline{\Sigma}^* \to \mathbb{R}^D$ *is a representation function of strings $\overline{\boldsymbol{y}} \in \overline{\Sigma}^*$.*[3]

We will refer to $\mathsf{enc}(\overline{\boldsymbol{y}})$ as the **encoding** of $\overline{\boldsymbol{y}} \in \Sigma^*$. In the general framework, we can simply consider the encoding function enc to be a black box—however, a major part of Chapter 5 will concern defining specific functions enc and analyzing their properties.

With this, we now know how we can represent the discrete symbols and histories as real-valued vectors. We next consider how to use such representations for defining probability distributions over the next symbol.

3.1.2 Compatibility of Symbol and Context

Inner products naturally give rise to the geometric notion of angle, by giving us the means to measure the similarity between two representations. Concretely, the smaller the angle between the two representations is, the more similar the two representations are. In a Hilbert space, we *define* the cosine of the angle θ between the two representations

$$\cos(\theta) \stackrel{\text{def}}{=} \frac{\langle \mathbf{u}, \mathbf{v} \rangle}{\|\mathbf{u}\|\|\mathbf{v}\|}. \tag{3.20}$$

The Cauchy–Schwartz inequality immediately gives us that $\cos(\theta) \in [-1, 1]$ since $-\|\mathbf{u}\|\|\mathbf{v}\| \leqslant \langle \mathbf{u}, \mathbf{v} \rangle \leqslant \|\mathbf{u}\|\|\mathbf{v}\|$. Traditionally, however, we take the *unnormalized* cosine similarity as our measure of similarity, which simply corresponds to the inner product of the Hilbert space.

Given a context representation $\mathsf{enc}(\overline{\boldsymbol{y}})$, we can compute its inner products with all symbol representations $\mathbf{e}(\overline{y})$:

$$\langle \mathbf{e}(\overline{y}), \mathsf{enc}(\overline{\boldsymbol{y}}) \rangle. \tag{3.21}$$

which can be achieved simply with a matrix-vector product:

$$\mathbf{E}\,\mathsf{enc}(\overline{\boldsymbol{y}}). \tag{3.22}$$

$\mathbf{E}\,\mathsf{enc}(\overline{\boldsymbol{y}}) \in \mathbb{R}^{|\overline{\Sigma}|}$, therefore, has the nice property that each of the individual entries corresponds to the similarities of a particular symbol to the context $\overline{\boldsymbol{y}}$. For reasons that will become clear soon, the entries of the vector $\mathbf{E}\,\mathsf{enc}(\overline{\boldsymbol{y}})$ are

[3] Note that, to be completely consistent, the encoding function should be defined over the set $\left(\overline{\Sigma} \cup \{\text{BOS}\}\right)^*$ to allow for the case when $y_0 = \text{BOS}$. However, unlike EOS, we do not necessarily require BOS in any formal setting, which is why we leave it out. We apologize for this inconsistency.

often called **scores** or **logits**. This brings us almost to the final formulation of the probability distribution $p_{\text{SM}}(\overline{y} \mid \overline{\boldsymbol{y}})$.

If $\mathbf{E}\,\mathsf{enc}(\overline{\boldsymbol{y}})$ encodes similarity or compatibility, then a natural way to model the probability distribution $p_{\text{SM}}(\overline{y} \mid \overline{\boldsymbol{y}})$ would be as proportional to the inner product between $\mathbf{e}(\overline{y})$ and $\mathsf{enc}(\overline{\boldsymbol{y}})$. However, the inner product $\langle \mathbf{e}(\overline{y}), \mathsf{enc}(\overline{\boldsymbol{y}}) \rangle$ may be negative; further, the sum over the similarity between a context and all tokens is not necessarily 1. To resolve this, we have to introduce the last piece of the puzzle: transforming $\mathbf{E}\,\mathsf{enc}(\overline{\boldsymbol{y}})$ into a valid discrete probability distribution by using a projection function.

3.1.3 Projecting onto the Simplex

In the previous subsections we discussed how to encode symbols and contexts in a Hilbert space and how an inner product gives us a natural notation of similarity between a potentially infinite number of items. We can now finally discuss how to create the conditional distribution $p_{\text{SM}}(\overline{y} \mid \overline{\boldsymbol{y}})$, i.e., how we can *map* the real-valued $\mathbf{E}\,\mathsf{enc}(\overline{\boldsymbol{y}})$ that encodes symbol–context similarities to a valid probability distribution—a vector on the **probability simplex**.

Projection Functions: Mapping Vectors onto the Probability Simplex

$p_{\text{SM}}(\overline{y} \mid \overline{\boldsymbol{y}})$ is a **categorical distribution** with $|\overline{\Sigma}|$ categories, i.e., a vector of probabilities whose components correspond to the probabilities of individual categories. Perhaps the simplest way to represent a categorical distribution is as a vector on a probability simplex.

Definition 3.1.8 (Probability Simplex)**.** *A* ***probability simplex*** $\boldsymbol{\Delta}^{D-1}$ *is the set of non-negative vectors* $\mathbb{R}^D$ *whose components sum to* 1*:*

$$\boldsymbol{\Delta}^{D-1} \stackrel{\text{def}}{=} \left\{ \mathbf{x} \in \mathbb{R}^D \mid x_d \geqslant 0, d = 1, \ldots, D \text{ and } \sum_{d=1}^{D} x_d = 1 \right\} \tag{3.23}$$

So far, we have framed p_{SM} as a function assigning the conditional distribution over $\overline{y}$ to each string $\overline{\boldsymbol{y}}$. The definition of a simplex means that we can more formally express p_{SM} as a **projection** from the Hilbert space of the context representations to $\boldsymbol{\Delta}^{|\overline{\Sigma}|-1}$, i.e., $p_{\text{SM}} \colon \mathbb{V} \to \boldsymbol{\Delta}^{|\overline{\Sigma}|-1}$. Yet all we have discussed so far is creating a vector $\mathbf{E}\,\mathsf{enc}(\overline{\boldsymbol{y}})$ that encodes symbol–context similarities—$\mathbf{E}\,\mathsf{enc}(\overline{\boldsymbol{y}})$ is not necessarily on the probability simplex $\boldsymbol{\Delta}^{|\overline{\Sigma}|-1}$. To address this issue, we turn to projection functions:

Definition 3.1.9 (Projection Function)**.** *A* ***projection function*** $\mathbf{f}_{\boldsymbol{\Delta}^{D-1}}$ *is a mapping from a real-valued Hilbert space* $\mathbb{R}^D$ *to the probability simplex* $\boldsymbol{\Delta}^{D-1}$

$$\mathbf{f}_{\boldsymbol{\Delta}^{D-1}} \colon \mathbb{R}^D \to \boldsymbol{\Delta}^{D-1}. \tag{3.24}$$

which allows us to define a probability distribution according to $\mathbf{E}\,\mathsf{enc}(\overline{\boldsymbol{y}})$:

$$p_{\text{SM}}(\overline{y} \mid \overline{\boldsymbol{y}}) = \mathbf{f}_{\boldsymbol{\Delta}^{|\overline{\Sigma}|-1}}(\mathbf{E}\,\mathsf{enc}(\overline{\boldsymbol{y}}))_{\overline{y}} \tag{3.25}$$

Clearly, we still want the projection of $\mathbf{E}\,\mathsf{enc}\,(\overline{\boldsymbol{y}})$ onto $\boldsymbol{\Delta}^{|\overline{\Sigma}|-1}$ to maintain several attributes of the original vector—otherwise, we will lose the notion of compatibility that $\mathbf{E}\,\mathsf{enc}\,(\overline{\boldsymbol{y}})$ inherently encodes. However, $\mathbf{f}_{\boldsymbol{\Delta}^{|\overline{\Sigma}|-1}}$ must satisfy several additional criteria in order to map onto a valid point in $\boldsymbol{\Delta}^{|\overline{\Sigma}|-1}$. For example, the inner product of two vectors (and consequently $\mathbf{E}\,\mathsf{enc}\,(\overline{\boldsymbol{y}})$) is not necessarily positive—yet all points in $\boldsymbol{\Delta}^{|\overline{\Sigma}|-1}$ are positive (see Definition 3.1.8). These characteristics motivate the use of a projection function that is both monotonic and positive everywhere. Thus, one clear choice is to base our chosen projection function on the exponential function, i.e.,

$$\mathbf{f}_{\boldsymbol{\Delta}^{|\overline{\Sigma}|-1}}(\mathbf{E}\,\mathsf{enc}\,(\overline{\boldsymbol{y}})) \propto \exp\left(\mathbf{E}\,\mathsf{enc}\,(\overline{\boldsymbol{y}})\right). \tag{3.26}$$

To make a function of the form in Eq. (3.26) a valid projection function, we now simply have to ensure that the output of $\mathbf{f}_{\boldsymbol{\Delta}^{D-1}}$ sums to 1, which can easily be accomplished by *re-normalizing* the vector of exponentiated values by their sum. This brings us to the main star of this subsection: the softmax.

While we simply motivated its introduction by chasing our goal of ending up on the probability simplex, the origin of the softmax function goes back to the Boltzmann distribution from statistical mechanics introduced in the mid-1800s by Boltzmann (1868). It was then studied intensely and popularized by Gibbs (1902). It was originally introduced as a way to convert the energy function of the Boltzmann distribution into a probability distribution.[4] Yet now, for reasons we will see in this subsection, the softmax is the predominant choice of projection function in machine learning applications.

Formally, the softmax is often defined in terms of a temperature parameter τ as follows.

Definition 3.1.10 (Softmax)**.** *Let $\tau \in \mathbb{R}_+$ be the **temperature**. The **softmax** at temperature τ is the projection function defined as:*

$$\operatorname{softmax}(\mathbf{x})_d \stackrel{\text{def}}{=} \frac{\exp\left[\frac{1}{\tau}x_d\right]}{\sum_{j=1}^{D}\exp\left[\frac{1}{\tau}x_j\right]}, \text{ for } d = 1, \ldots, D \tag{3.27}$$

where the temperature parameter τ gives us a mechanism for controlling the entropy of the softmax function by *scaling* the individual scores in the input vector before their exponentiation. In the context of the Boltzmann distribution, it was used to control the "randomness" of the system: When the temperature is high, the softmax function outputs a more uniform probability distribution whose probabilities are relatively evenly spread out among the different categories. When the temperature is low, the softmax function outputs a peaked probability distribution, where the probability mass is concentrated on the most likely category. In the limit, as we take τ to the edge of the possible values it can assume, the following properties hold:

[4] This is precisely the connection we mentioned in Definition 2.4.1.

Theorem 3.1.2.

$$\lim_{\tau \to \infty} \text{softmax}\left(\mathbf{x}\right) = \frac{1}{D}\mathbf{1} \tag{3.28}$$

$$\lim_{\tau \to 0^+} \text{softmax}\left(\mathbf{x}\right) = \mathbf{e}_{\text{argmax}(\mathbf{x})}, \tag{3.29}$$

where $\mathbf{e}_d$ *denotes the* d^{th} *basis vector in* $\mathbb{R}^D$, $\mathbf{1} \in \mathbb{R}^D$ *the vector of all ones, and*

$$\text{argmax}\left(\mathbf{x}\right) \stackrel{\text{def}}{=} \min\left\{d \mid x_d = \max_{d=1,\ldots,D}\left(x_d\right)\right\}, \tag{3.30}$$

i.e., the index of the maximum element of the vector $\mathbf{x}$ *(with the ties broken by choosing the lowest such index). In words, this means that the output of the* softmax *approaches the uniform distribution as* $\tau \to \infty$ *and towards a single mode as* $\tau \to 0^+$*.*[5]

Proof. Let us first consider the case of $\tau \to 0^+$. Without loss of generality, let us consider a 2-dimensional vector $\mathbf{x} = \left[x_1, x_2\right]^\top$

$$\lim_{\tau \to 0^+} \text{softmax}(\mathbf{x})_1 = \lim_{\tau \to 0^+} \frac{\exp\left(\frac{x_1}{\tau}\right)}{\exp\left(\frac{x_1}{\tau}\right) + \exp\left(\frac{x_2}{\tau}\right)} \tag{3.31}$$

$$= \lim_{\tau \to 0^+} \frac{\exp\left(\frac{x_1}{\tau}\right)\exp\left(-\frac{x_1}{\tau}\right)}{\left(\exp\left(\frac{x_1}{\tau}\right) + \exp\left(\frac{x_2}{\tau}\right)\right)\exp\left(-\frac{x_1}{\tau}\right)} \tag{3.32}$$

$$= \lim_{\tau \to 0^+} \frac{1}{1 + \exp\left(\frac{x_2 - x_1}{\tau}\right)} \tag{3.33}$$

which leads us to the following definition for element-wise values:

$$\lim_{\tau \to 0^+} \exp\left(\frac{x_2 - x_1}{\tau}\right) = \begin{cases} 0, & \text{if } x_1 > x_2 \\ 1, & \text{if } x_1 = x_2 \\ \infty, & \text{o.w.} \end{cases} \tag{3.34}$$

Then the limit of softmax as $\tau \to 0^+$ is given as

$$\lim_{\tau \to 0^+} \text{softmax}(\mathbf{x}) = \begin{cases} \left[1, 0\right]^\top, & \text{if } x_1 > x_2 \\ \left[\frac{1}{2}, \frac{1}{2}\right]^\top, & \text{if } x_1 = x_2 \\ \left[0, 1\right]^\top, & \text{o.w.} \end{cases} \tag{3.35}$$

which is equivalent to the argmax operator over $\mathbf{x}$. The proof extends to arbitrary D-dimensional vectors.

The case of $\tau \to \infty$ follows similar logic, albeit $\lim_{\tau \to \infty} \exp\left(\frac{x_2 - x_1}{\tau}\right) = 1$ in all cases. Hence, we get $\lim_{\tau \to \infty} \text{softmax}(\mathbf{x}) = \frac{1}{D}\mathbf{1}$. ■

The second property, specifically, shows that the softmax function resembles the argmax function as the temperature approaches 0—in that sense, a more sensible name for the function would have been "soft*arg*max". We will most

[5] $\tau \to 0^+$ denotes the limit from above.

often simply take τ to be 1. However, different values of the parameter are especially useful when *sampling* or generating text from the model, as we discuss subsequently.

The output of the softmax is equivalent to the solution to a particular optimization problem, giving it a variational interpretation.

Theorem 3.1.3. *Given a set of real-valued scores* $\mathbf{x}$*, the following equality holds*

$$\text{softmax}(\mathbf{x}) = \underset{\mathbf{p}\in\boldsymbol{\Delta}^{D-1}}{\text{argmax}} \left(\mathbf{p}^\top \mathbf{x} - \tau \sum_{d=1}^{D} \mathrm{p}_d \log \mathrm{p}_d \right) \tag{3.36}$$

$$= \underset{\mathbf{p}\in\boldsymbol{\Delta}^{D-1}}{\text{argmax}} \left(\mathbf{p}^\top \mathbf{x} + \tau \mathrm{H}(\mathbf{p}) \right) \tag{3.37}$$

This tells us that softmax can be given a variational characterization, i.e., it can be viewed as the solution to an optimization problem.

Proof. Eq. (3.36) can equivalently be written as

$$\text{softmax}(\mathbf{x}) = \text{argmax} \left(\mathbf{p}^\top \mathbf{x} - \tau \sum_{d=1}^{D} \mathrm{p}_d \log \mathrm{p}_d \right) \tag{3.38}$$

$$\text{s.t.} \sum_d p_d = 1 \tag{3.39}$$

from which we can clearly see that the Lagrangian of this optimization problem is $\Lambda = \mathbf{p}^\top \mathbf{x} - \tau \sum_{d=1}^{D} \mathrm{p}_d \log \mathrm{p}_d + \lambda \sum_d p_d$. Taking the derivative of Λ with respect to p_d, we see that the optimum of is reached when

$$\frac{\partial \Lambda}{\partial p_d} = v_d - \tau(\log p_d + 1) + \lambda = 0 \tag{3.40}$$

Solving for p_d gives us $p_d = Z \exp(\frac{x_i}{\tau})$, where Z is the normalizing constant that ensures $\sum_d p_d = 1$. This solution is equivalent to performing the softmax operation over $\mathbf{x}$, as desired. ■

Theorem 3.1.3 reveals an interpretation of the softmax as the projection $\mathbf{p} \in \boldsymbol{\Delta}^{D-1}$ that has the maximal similarity with $\mathbf{x}$ while being regularized to produce a solution with high entropy. Further, from both Definition 3.1.10 and Eq. (3.36), we can see that softmax leads to non-sparse solutions as an entry $\text{softmax}(\mathbf{x})_i$ can only be 0 if $x_d = -\infty$.

In summary, the softmax has a number of desirable properties for use in machine learning settings.

Theorem 3.1.4. *The* softmax *function with temperature parameter* τ *exhibits the following properties.*

1. *In the limit as* $\tau \to 0^+$ *and* $\tau \to \infty$*, the* softmax *recovers the* argmax *operator and projection to the center of the probability simplex (at which lies the uniform distribution), respectively.*

2. $\text{softmax}(\mathbf{x} + c\mathbf{1}) = \text{softmax}(\mathbf{x})$ *for* $c \in \mathbb{R}$*, i.e., the* softmax *is invariant to adding the same constant to all coordinates in* $\mathbf{x}$*.*

3. *The derivative of the* softmax *is continuous and differentiable everywhere; the value of its derivative can be explicitly computed.*

4. *For all temperatures* $\tau \in \mathbb{R}_+$*, if* $x_i \leqslant x_j$*, then* $\text{softmax}(\mathbf{x})_i \leqslant \text{softmax}(\mathbf{x})_j$*. In words, the* softmax *maintains the rank of* $\mathbf{x}$*.*

Proof. Property 1. is simply a restatement of Theorem 3.1.2. The proof for property 2. can be shown using simple algebraic manipulation:

$$\text{softmax}(\mathbf{x} + c\mathbf{1})_d = \frac{\exp\left[\frac{1}{\tau} x_d + c\right]}{\sum_{j=1}^{D} \exp\left[\frac{1}{\tau} x_j + c\right]} = \frac{\exp\left[\frac{1}{\tau} x_d\right] \cdot \exp c}{\sum_{j=1}^{D} \exp\left[\frac{1}{\tau} x_j\right] \cdot \exp c} = \text{softmax}(\mathbf{x})_d \tag{3.41}$$

The derivative of the softmax at position i with respect to the variable at position j is given by

$$\frac{\partial \text{softmax}(\mathbf{x})_i}{x_j} = \frac{\delta_i(j) \cdot \exp(x_i) \sum_k \exp(x_k) - \exp(x_i) \cdot \exp(x_j)}{(\sum_k \exp(x_k))^2} \tag{3.42}$$

where $\delta_i(j)$ is the Dirac Delta function, defined as $\delta_i(j) = \begin{cases} 1 \textbf{ if } i = j \\ 0 \textbf{ else} \end{cases}$. Clearly, Eq. (3.42) is continuous. Further, it takes on values for all $\mathbf{x} \in \mathbb{R}^d$. Lastly, property 4. follows from the monotonicity of the exp function.

■

There are many other valid projection functions that one could choose from. For example, Martins and Astudillo (2016) introduce the **sparsemax**, which *can* output sparse distributions:

$$\text{sparsemax}(\mathbf{x}) \stackrel{\text{def}}{=} \underset{\mathbf{p} \in \boldsymbol{\Delta}^{D-1}}{\text{argmin}} \, ||\mathbf{p} - \mathbf{x}||_2^2 \tag{3.43}$$

In words, sparsemax directly maps $\mathbf{x}$ onto the probability simplex, which often leads to solutions on the boundary, i.e., where at least one entry of $\mathbf{p}$ is 0. Martins and Astudillo (2016) provide a method for computing the closed form solution of this optimization problem in Alg. 1 of their work. Blondel et al. (2019) later introduced a framework that encompasses many different projection functions, which they term regularized prediction functions. Essentially, this framework considers the subset of projection functions that can be written as:

$$\mathbf{f}_{\boldsymbol{\Delta}^{|\overline{\Sigma}|-1}}(\mathbf{x}) \stackrel{\text{def}}{=} \underset{\mathbf{p} \in \boldsymbol{\Delta}^{D-1}}{\text{argmax}} \left(\mathbf{p}^\top \mathbf{x} - \Omega(\mathbf{p})\right) \tag{3.44}$$

where $\Omega \colon \mathbb{R}^D \to \mathbb{R}$ is regularization term. For certain choices of Ω, there are straightforward closed-form solutions to Eq. (3.44). For example, as we can see from Eq. (3.36), Eq. (3.44) is equivalent to the softmax when $\Omega(\mathbf{p}) = -\text{H}(\mathbf{p})$, meaning we can compute its closed form using Eq. (3.27). Further, we recover

the sparsemax when $\Omega(\mathbf{p}) = -||\mathbf{p}||_2^2$, which likewise has a closed-form solution. The notion of regularizing $\mathbf{p}$ may be unintuitive at first, but we can view it as trying to balance out the "suitability" term $\mathbf{p}^\top \mathbf{x}$ with a "confidence" term $\Omega(\mathbf{p})$, which should be smaller when $\mathbf{p}$ is "uncertain." We point the interested reader to the comprehensive work of Blondel et al. (2019) for further elaboration.

So why aren't these other projection functions more widely employed in machine learning frameworks? First, not all choices of Ω lead to closed-form solutions; further, not all meet the desirable criterion listed in Theorem 3.1.4. For example, the sparsemax is not everywhere differentiable, meaning that one could not simply use out-of-the-box automatic differentiation frameworks when training a model using the sparsemax as its projection function. Rather one would have to specify its gradient explicitly.

Theorem 3.1.5. *The derivative of the the* sparsemax *with respect to its input* $\mathbf{x}$ *is as follows:*

$$\frac{\partial \text{sparsemax}(\mathbf{x})_i}{\partial x_j} = \begin{cases} \delta_{ij} - \frac{1}{S(\mathbf{x})} & \textbf{\textit{if}}\ i, j \in S(\mathbf{x}) \\ 0 & \textbf{\textit{else}} \end{cases} \tag{3.45}$$

Proof. ■

To conclude, projection functions, together with symbol representations and the representation function enc, give us the tools to define a probability distribution over next symbols that encodes complex linguistic interactions. We now bring all the components together into the locally normalized modeling framework in the next section.

3.1.4 Representation-based Locally Normalized Models

With these tools at hand, we now define representation-based locally normalized language models.

Definition 3.1.11 (Representation-Based Locally Normalized Model)**.** *Let* enc *be an encoding function. A* ***representation-based locally normalized model*** *is a model of the following form:*

$$p_{SM}(\overline{y}_t \mid \overline{\boldsymbol{y}}_{<t}) \stackrel{\text{def}}{=} \mathbf{f}_{\boldsymbol{\Delta}^{|\overline{\Sigma}|-1}}\left(\mathbf{E}\,\mathsf{enc}(\overline{\boldsymbol{y}}_{<t})\right)_{\overline{y}_t} \tag{3.46}$$

where unless otherwise stated, we assume $\mathbf{f}_{\boldsymbol{\Delta}^{|\overline{\Sigma}|-1}} = \text{softmax}$. *It defines the probability of an entire string* $\boldsymbol{y} \in \Sigma^*$ *as*

$$p_{LN}(\boldsymbol{y}) \stackrel{\text{def}}{=} p_{SM}(\text{EOS} \mid \boldsymbol{y}) \prod_{t=1}^{T} p_{SM}(y_t \mid \boldsymbol{y}_{<t}) \tag{3.47}$$

where $y_0 \stackrel{\text{def}}{=} \text{BOS}$.

Alternatively, we could also include an additive **bias** term $\mathbf{b}$ as part of the projection function $\mathbf{f}_{\boldsymbol{\Delta}^{|\overline{\Sigma}|-1}}$ in the definition of the conditional distribution $p_{\text{SM}}(\overline{y}_t \mid \overline{\boldsymbol{y}}_{<t})$, i.e., $p_{\text{SM}}(\overline{y}_t \mid \overline{\boldsymbol{y}}_{<t}) = \mathbf{f}_{\boldsymbol{\Delta}^{|\overline{\Sigma}|-1}}(\mathbf{E}\,\mathsf{enc}(\overline{\boldsymbol{y}}_{<t}) + \mathbf{b})_{\overline{y}_t}$. However, note that the bias term can be absorbed into the encoding function enc, meaning that we can assume the form Eq. (3.46) without loss of generality. In representation-based language models, $\mathbf{e}(\overline{y})$ and $\mathsf{enc}(\boldsymbol{y})$ carry all the necessary information to determine how probable individual symbols y are given the context $\boldsymbol{y}$. Therefore, the design choices of $\mathbf{e}(\overline{y})$ and $\mathsf{enc}(\boldsymbol{y})$ are crucial when building language models this way. Indeed, a large portion of the discussion in the remainder of the notes will center around how to build good representations of the context and individual symbols.

3.1.5 Tightness of Softmax Representation-based Models

Having introduced representation-based language models, we can now state a very general result about the tightness of such models. It connects the notion of tightness to the intuition about the "compatibility" of symbols to the context—namely, the compatibility of the EOS symbol to the context (compared to the compatibility of all other symbols). The compatibility is here captured by the distance of the representation of the EOS symbol to the representation of the other symbols—if this distance grows slowly enough with respect to t (modulo the norm of the context representation), the model is tight.

Theorem 3.1.6 (Proposition 5.9 in Du et al., 2022)**.** *Let p_{SM} be a representation-based sequence model over the alphabet Σ, as defined in Definition 3.1.11. Let*

$$s \stackrel{\text{def}}{=} \sup_{y \in \Sigma} \|\mathbf{e}(y) - \mathbf{e}(\text{EOS})\|_2, \tag{3.48}$$

i.e, the largest distance to the representation of the EOS *symbol, and*

$$z_{max} \stackrel{\text{def}}{=} \max_{\boldsymbol{y} \in \Sigma^t} \|\mathsf{enc}(\boldsymbol{y})\|_2, \tag{3.49}$$

i.e., the maximum attainable context representation norm for contexts of length t. Then the locally normalized model p_{LN} induced by p_{SM} is tight if

$$s z_{max} \leqslant \log t. \tag{3.50}$$

Proof. Let $\mathrm{x}_t(\boldsymbol{\omega})$ be the random variable that is equal to the t^{th} token in an

outcome $\boldsymbol{\omega} \in \Omega$. Then for an arbitrary $t \in \mathbb{N}$ and any $\boldsymbol{y} \in \Sigma^t$, we have:

$$\mathbb{P}(\mathrm{x}_t = \textsc{eos} \mid \mathbf{x}_{<t} = \boldsymbol{y}) = \frac{\exp\left[\mathbf{e}(\textsc{eos})^\top \mathsf{enc}(\boldsymbol{y})\right]}{\sum_{y \in \overline{\Sigma}} \exp\left[\mathbf{e}(y)^\top \mathsf{enc}(\boldsymbol{y})\right]} \tag{3.51a}$$

$$= \frac{1}{\frac{\sum_{y \in \overline{\Sigma}} \exp[\mathbf{e}(y)^\top \mathsf{enc}(\boldsymbol{y})]}{\exp[\mathbf{e}(\textsc{eos})^\top \mathsf{enc}(\boldsymbol{y})]}} \tag{3.51b}$$

$$= \frac{1}{1 + \sum_{y \in \Sigma} \exp\left[(\mathbf{e}(y) - \mathbf{e}(\textsc{eos}))^\top \mathsf{enc}(\boldsymbol{y})\right]} \tag{3.51c}$$

$$\geqslant \frac{1}{1 + \sum_{y \in \Sigma} \exp\left[\|\mathbf{e}(y) - \mathbf{e}(\textsc{eos})\|_2 \|\mathsf{enc}(\boldsymbol{y})\|_2\right]} \quad \text{(Cauchy–Schwarz)} \tag{3.51d}$$

$$\geqslant \frac{1}{1 + \sum_{y \in \Sigma} \exp\left[k \|\mathsf{enc}(\boldsymbol{y})\|_2\right]} \tag{3.51e}$$

$$= \frac{1}{1 + |\Sigma| \exp\left[\, \|\mathsf{enc}(\boldsymbol{y})\|_2\right]} \tag{3.51f}$$

Now define $z_{\max} \stackrel{\text{def}}{=} \sup_{\boldsymbol{y} \in \Sigma^t} \|\mathsf{enc}(\boldsymbol{y})\|_2$. We then have that $\forall t \in \mathbb{N}$ and $\forall \boldsymbol{y} \in \Sigma^t$:

$$\mathbb{P}(\mathrm{x}_t = \textsc{eos} \mid \mathbf{x}_{<t} = \boldsymbol{y}) \geqslant \frac{1}{1 + |\Sigma| \exp(k z_{\max})} \tag{3.52}$$

Now, by Proposition 2.5.6, we have that if $\sum_{t=1}^{\infty} \frac{1}{1+|\Sigma| \exp(k\, z_{\max})}$ diverges, then the language model is tight. We will show that if we have that $\exists N \in \mathbb{N}$ such that $\forall t \geqslant N$, $k z_{\max} \leqslant \log t$, then the sequence model must be tight.

First, note that $\lim_{t \to \infty} \frac{1}{t} \frac{1+|\Sigma| t}{1} = \lim_{t \to \infty} \frac{1}{t} + |\Sigma| = |\Sigma| \in (0, \infty)$. Hence, by the limit comparison test, since $\sum_{t=1}^{\infty} \frac{1}{t}$ diverges, this means $\sum_{t=1}^{\infty} \frac{1}{1+|\Sigma| t}$ must also diverge.

Now, suppose that $k\, z_{\max} \leqslant \log t$ for all $t \geqslant N$. This implies that for $t \geqslant N$ we have $\frac{1}{1+|\Sigma| \exp(k z_{\max})} \geqslant \frac{1}{1+|\Sigma| t}$, which combined with the above and the comparison test, implies that $\sum_{t=N}^{\infty} \frac{1}{1+|\Sigma| \exp(k z_{\max})}$ diverges. This in turn means that $\sum_{t=1}^{\infty} \frac{1}{1+|\Sigma| \exp(k z_{\max})}$ diverges. Hence, if $k\, z_{\max} \leqslant \log t$ for all $t \geqslant N$ for some $N \in \mathbb{N}$, then the language model is tight.

■

Theorem 3.1.6 is a generalization of the following result from Welleck et al. (2020).

Theorem 3.1.7. *A locally-normalized representation-based language model, as defined in Definition 3.1.11, with uniformly bounded $||\mathsf{enc}(\boldsymbol{y})||_p$ (for some $p \geqslant 1$) is tight.*

For most of the language models that we consider, $\mathsf{enc}(\boldsymbol{y})$ is bound due to the choice of activation functions. In turn, $\mathbf{E}\, \mathsf{enc}(\overline{\boldsymbol{y}}_{<t})$ is bounded for all $\overline{\boldsymbol{y}}$. Further, by the definition of the softmax, $\mathbf{f}_{\boldsymbol{\Delta}^{|\overline{\Sigma}|-1}}(\mathbf{E}\, \mathsf{enc}(\overline{\boldsymbol{y}}_{<t}))_{\textsc{eos}} > \eta$ for some constant η.

This concludes our investigation of general representation-based models. The next section discusses *learning* parametrized models (as a special case, also symbol and context representations).

3.2 Estimating a Language Model from Data

The **language modeling task** refers to any attempt to estimate the parameters[6] of a model p_{M} of the ground-truth probability distribution over natural language strings p_{LM} using data $\mathcal{D} = \{\boldsymbol{y}^{(n)}\}_{n=1}^{N}$, where we assume samples $\boldsymbol{y}^{(n)}$ were generated according to p_{LM}. This task is often treated as an optimization problem. Here we will discuss the various components of this optimization problem, primarily the objective and the algorithm used to perform optimization. Note that the material covered here corresponds to what is colloquially referred to as pre-training. The learning paradigm for fine-tuning a language model for a downstream task will be covered later in the course.

3.2.1 Data

In this course, we consider objectives that are defined in terms of data $\mathcal{D}$. Therefore, we will first discuss the nature of this data which, more precisely, is a corpus of texts. Following the notation used throughout the rest of these notes, let Σ be an alphabet. A **corpus** $\mathcal{D} = \{\boldsymbol{y}^{(n)}\}_{n=1}^{N} \subset \Sigma^*$ is a collection of N strings. We will use the terms corpus and dataset interchangeably throughout this section. We make the following assumption about the data-generating process of $\mathcal{D}$:

Assumption 3.2.1. *The strings $\boldsymbol{y}^{(n)}$ in our corpus $\mathcal{D}$ are generated independently and identically distributed (i.i.d.) by some unknown distribution p_{LM}.*

Note that $\boldsymbol{y}^{(n)}$ are strings of an arbitrary length; they can be single words, sentences, paragraphs, or even entire documents depending on how we choose Σ. For example, often our models' architectural designs make them unable to process document-length strings efficiently, e.g., they might not fit into a context window that can be reasonably processed by a transformer language model; we will elaborate on this statement in our discussion of transformers in **??**. Thus in practice, we often chunk documents into paragraphs that we treat as separate data points.[7] This means that our model may not be able to learn properties of language such as discourse structure. We will discuss other factors influencing our choice of unit size—including that of a string $\boldsymbol{y}$ and the symbols that constitute that string y—as well as the implications of these choices in the subsequent section on tokenization.

3.2.2 Language Modeling Objectives

Similarly to many other machine learning tasks, we can cast our problem as the *search* for the best model p_{M} of the ground-truth distribution over strings p_{LM}. In order to make this search tractable, we must limit the models p_{M} that we consider. Explicitly, we make the following assumption:

[6] Most of this course focuses on the parametric case, i.e., where p_{M} is governed by a set of parameters $\boldsymbol{\theta}$. However, we will briefly touch upon various non-parametric language models.

[7] This practice technically breaks Assumption 3.2.1, yet the negative (empirically-observed) effects of this violation are minimal and perhaps outweighed by the additional data it allows us to make use of.

Assumption 3.2.2. *p_{LM} is a member of the parameterized family of models $\{p_{\boldsymbol{\theta}} \mid \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$, the set of all distributions representable by parameters $\boldsymbol{\theta}$ in a given parameter space $\boldsymbol{\Theta}$.*

As concrete examples, $\boldsymbol{\theta}$ could be the conditional probabilities in a simple, standard n-gram model for a given prefix of size $n-1$, i.e., $\boldsymbol{\theta}$ is $n-1$ simplices of size $|\Sigma|$.[8] As another example, $\boldsymbol{\theta}$ could be the weights of a neural network; the set $\boldsymbol{\Theta}$ would then cover all possible valid weight matrices that could parameterize our model.

Assumption 3.2.2 implies that we can equivalently write p_{LM} as $p_{\boldsymbol{\theta}^\star}$ for certain (unknown) parameters $\boldsymbol{\theta}^\star \in \boldsymbol{\Theta}$.[9] Further, an arbitrary model p_{M} from this hypothesis space with parameters $\boldsymbol{\theta}$ can be written as $p_{\boldsymbol{\theta}}$; we will use this notation for the remainder of the chapter to make the parameterization of our distribution explicit. We now turn to the general framework for choosing the best parameters $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ so that our model $p_{\boldsymbol{\theta}}$ serves as a good approximation of $p_{\boldsymbol{\theta}^\star}$.[10]

General Framework

We search for model parameters $\widehat{\boldsymbol{\theta}} \in \boldsymbol{\Theta}$ such that the model induced by those parameters maximizes a chosen objective, or alternatively, minimizes some loss function $\ell : \boldsymbol{\Theta} \times \boldsymbol{\Theta} \to \mathbb{R}_{\geqslant 0}$. This loss can be used to measure the quality of this model as an approximation to $p_{\boldsymbol{\theta}^\star}$. In simple math, we search for the solution.

$$\widehat{\boldsymbol{\theta}} \stackrel{\text{def}}{=} \operatorname*{argmin}_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \ell(\boldsymbol{\theta}^\star, \boldsymbol{\theta}) \tag{3.53}$$

where our loss function is chosen with the following principle in mind

Principle 3.2.1 (Proximity Principle)**.** *We seek a model $p_{\boldsymbol{\theta}}$ that is "close" to $p_{\boldsymbol{\theta}^\star}$.*

That is, we choose our loss function to be a measure M of the difference between a distribution parameterized by $\boldsymbol{\theta}$ and one parameterized by the true $\boldsymbol{\theta}^\star$, i.e., those of our ground-truth distribution. Yet we are immediately faced with a problem: computing an arbitrary M between $\boldsymbol{\theta}$ and $\boldsymbol{\theta}^\star$ (or at least the distributions induced by these sets of parameters) requires knowledge of both, the latter for which we only have samples $\boldsymbol{y}^{(n)} \in \mathcal{D}$. We will therefore use our corpus $\mathcal{D}$ as an approximation to $p_{\boldsymbol{\theta}^\star}$, which is typically implemented by representing $\mathcal{D}$ as an empirical distribution—a collection of Dirac Delta functions—which we will denote as $\widetilde{p_{\boldsymbol{\theta}^\star}}$. Formally, we define

$$\widetilde{p_{\boldsymbol{\theta}^\star}}(\boldsymbol{y}) \stackrel{\text{def}}{=} \frac{1}{N} \sum_{n=1}^{N} \delta_{\boldsymbol{y}^{(n)}}(\boldsymbol{y}) \tag{3.54}$$

[8] One might be tempted to assume we only need $|\Sigma| - 1$ parameters per simplex, but we condition over $\overline{\Sigma}$ classes per prefix position.

[9] We discuss the implications of the case that $p_{\text{LM}} \notin \{p_{\boldsymbol{\theta}} \mid \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$ later in this section.

[10] The modeling paradigms that we will discuss in this section are predominantly generative, i.e., these models try to learn the underlying distribution of the data rather than the boundaries between different classes or categories. The implication is that parameter estimation in language modeling typically makes use of *unannotated* text data, and is therefore sometimes referred to as **self-supervised**.

where the Dirac Delta function $\delta_{x'}(x) = \begin{cases} 1 \textbf{ if } x = x' \\ 0 \textbf{ else} \end{cases}$ is essentially a point mass with all probability on x'. We can decompose this definition over symbols in our strings as well. I.e., we can compute

$$\widetilde{p_{\boldsymbol{\theta}^\star}}(y_t \mid \boldsymbol{y}_{<t}) = \frac{1}{N_{\boldsymbol{y}_{<t}}} \sum_{n=1}^{N} \delta_{y_t{}^{(n)} \mid \boldsymbol{y}_{<t}^{(n)}}(y_t \mid \boldsymbol{y}_{<t}) \tag{3.55}$$

where $N_{\boldsymbol{y}_{<t}} \stackrel{\text{def}}{=} \sum_{n=1}^{N} \mathbb{1}\{\boldsymbol{y}_{<t}^{(n)} = \boldsymbol{y}_{<t}\}$. Note that we can likewise define Eq. (3.55) in terms of the one-hot encodings of symbols, i.e., using the definition in Example 3.1.2: $\widetilde{p_{\boldsymbol{\theta}^\star}}(\cdot \mid \boldsymbol{y}_{<t}) = \frac{1}{N_{\boldsymbol{y}_{<t}}} \sum_{n=1}^{N} \mathbf{o}_{y_t{}^{(n)}} \mathbb{1}\{\boldsymbol{y}_{<t}^{(n)} = \boldsymbol{y}_{<t}\}$. In fact, the empirical distribution is often also referred to in machine learning as the one-hot encoding of a dataset.

Now that we are equipped with methods for representing both $p_{\boldsymbol{\theta}^\star}$ and $p_{\boldsymbol{\theta}}$, we can define a loss function for approximating $p_{\boldsymbol{\theta}^\star}$ using $p_{\boldsymbol{\theta}}$.

Cross-Entropy. A natural choice for a loss function is cross-entropy, a measure of the difference between two probability distributions, which has its roots in information theory (Shannon, 1948). Specifically, in Eq. (3.53), we take $\ell(\boldsymbol{\theta}^\star, \boldsymbol{\theta}) = \mathrm{H}(\widetilde{p_{\boldsymbol{\theta}^\star}}, p_{\boldsymbol{\theta}})$ where the definition of the cross-entropy H between distributions p_1 (with support $\mathcal{Y}$) and p_2 is as follows:

$$\mathrm{H}(p_1, p_2) = -\sum_{\boldsymbol{y} \in \mathcal{Y}} p_1(\boldsymbol{y}) \log p_2(\boldsymbol{y}) \tag{3.56}$$

Further, most of the models that we will encounter in this course are locally normalized. Thus, it is more common to see cross-entropy expressed as

$$\mathrm{H}(p_1, p_2) = -\sum_{\boldsymbol{y} \in \mathcal{Y}} \sum_{t=1}^{T} p_1(y_t{}^{(n)}) \log p_2(y_t \mid \boldsymbol{y}_{<t}). \tag{3.57}$$

Note that cross-entropy is not symmetric, i.e., $\mathrm{H}(p_1, p_2) \neq \mathrm{H}(p_2, p_1)$. To motivate cross-entropy as a loss function, as well as the intuitive difference between the two argument orderings, we turn to coding theory, a sub-field of information theory. In words, the cross-entropy between two probability distributions is the expected number of bits needed to encode an event $\boldsymbol{y} \in \mathcal{Y}$ from p_1 when using the optimal encoding scheme corresponding to distribution p_2. Importantly, the optimal encoding scheme for p_1 uses $\log p_1(\boldsymbol{y})$ bits to encode an event $\boldsymbol{y}$ that occurs with probability $p_1(\boldsymbol{y})$, implying that the minimal cross-entropy is achieved when $p_1 = p_2$. This characteristic of cross-entropy motivates another metric: the KL divergence D_{KL}.

KL **Divergence.** A **divergence measure** is a measure of statistical distance[11] between two probability distributions. The KL divergence is defined as:

$$D_{\mathrm{KL}}(p_1 \mid\mid p_2) = \sum_{\boldsymbol{y} \in \mathcal{Y}} p_1(\boldsymbol{y}) \log p_2(\boldsymbol{y}) - p_1(\boldsymbol{y}) \log p_1(\boldsymbol{y}) \tag{3.58}$$

The KL divergence can intuitively be viewed as the cross-entropy shifted by the expected number of bits used by the optimal encoding scheme for p_1, i.e., it is the *additional* number of expected bits needed to encode events from p_1 when using our encoding scheme from p_2. Indeed, taking $\ell(\boldsymbol{\theta}^\star, \boldsymbol{\theta}) = D_{\mathrm{KL}}(\widetilde{p_{\boldsymbol{\theta}^\star}} \mid\mid p_{\boldsymbol{\theta}})$ should lead to the same solution as taking $\ell(\boldsymbol{\theta}^\star, \boldsymbol{\theta}) = \mathrm{H}(\widetilde{p_{\boldsymbol{\theta}^\star}}, p_{\boldsymbol{\theta}})$ because the $\widetilde{p_{\boldsymbol{\theta}^\star}}(\boldsymbol{y}) \log \widetilde{p_{\boldsymbol{\theta}^\star}}(\boldsymbol{y})$ term is constant with respect to model parameters $\boldsymbol{\theta}$.

Relationship to Maximum Likelihood Estimation

An alternative way that we could frame our search for model parameters $\widehat{\boldsymbol{\theta}} \in \boldsymbol{\Theta}$ is in terms of data likelihood. Formally, the **likelihood** of the corpus $\mathcal{D}$ under the distribution $p_{\boldsymbol{\theta}}$ is the joint probability of all $\boldsymbol{y}^{(n)}$:

$$L(\boldsymbol{\theta}) = \prod_{n=1}^{N} p_{\boldsymbol{\theta}}(\boldsymbol{y}^{(n)}). \tag{3.59}$$

The principle of maximum likelihood then dictates:

Principle 3.2.2 (Maximum Likelihood)**.** *The optimal parameters for a model are those that maximize the likelihood of observing the given data under that model. Formally:*

$$\widehat{\boldsymbol{\theta}}_{\mathrm{MLE}} \stackrel{\mathrm{def}}{=} \operatorname*{argmax}_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \mathcal{L}(\boldsymbol{\theta}) \tag{3.60}$$

Note that in practice, we typically work with the **log-likelihood** $\mathcal{L}(\boldsymbol{\theta}) = \log L(\boldsymbol{\theta})$ rather than the likelihood for a number of reasons, e.g., it is convex and more numerically stable given the small probabilities we encounter when using L and the finite precision of the computing frameworks that we employ. Since log is a monotonically increasing function, this would not change the solution to Eq. (3.60). Further, as is the case with Eq. (3.57), we decompose our loss over symbol-level distributions.

Notably, in our setting, finding parameters that maximize data log-likelihood is equivalent to finding those that minimize cross-entropy. We show this equivalence below.

Proposition 3.2.1. *The optimal parameters under Eq. (3.60) are equivalent to the optimal parameters when solving for Eq. (3.53) with the cross-entropy loss between the empirical distribution $\widetilde{p_{\boldsymbol{\theta}^\star}}$ and the model $p_{\boldsymbol{\theta}}$.*

[11] Divergences are not technically distances because they are not symmetric, i.e., it may be the case for divergence measure D and probability distributions p and q that $D(p \mid\mid q) \neq D(p \mid\mid q)$. However, they do meet the criteria that $D(p \mid\mid q) \geqslant 0 \; \forall p, q$ and $D(p \mid\mid q) = 0 \iff p = q$.

Proof. Under the standard practice of taking $0 \log(0) = 0$, the only elements of $\mathcal{Y}$ that make a nonzero contribution to $\mathrm{H}(\widetilde{p_{\theta^\star}}, p_{\boldsymbol{\theta}})$ are sequences in the support of $\widetilde{p_{\theta^\star}}$, making summing over $\mathcal{Y}$ equivalent to summing over $\mathcal{D}$:

$$\mathrm{H}(\widetilde{p_{\theta^\star}}, p_{\boldsymbol{\theta}}) = -\sum_{\boldsymbol{y} \in \Sigma^*} \widetilde{p_{\theta^\star}}(\boldsymbol{y}) \log p_{\boldsymbol{\theta}}(\boldsymbol{y}) \tag{3.61}$$

$$= -\sum_{\boldsymbol{y} \in \Sigma^*} \frac{1}{N} \sum_{n=1}^{N} \delta_{\boldsymbol{y}^{(n)}}(\boldsymbol{y}) \log p_{\boldsymbol{\theta}}(\boldsymbol{y}) \tag{3.62}$$

$$= -\sum_{\boldsymbol{y} \in \Sigma^*} \frac{1}{N} \mathbb{1}\{\boldsymbol{y}^{(n)} \in \mathcal{D}\} \log p_{\boldsymbol{\theta}}(\boldsymbol{y}) \tag{3.63}$$

$$\propto -\sum_{\boldsymbol{y} \in \mathcal{D}} \log p_{\boldsymbol{\theta}}(\boldsymbol{y}) \tag{3.64}$$

$$= -\mathcal{L}(\boldsymbol{\theta}) \tag{3.65}$$

Thus, we can see that the objectives are equivalent, up to a multiplicative constant that is independent of model parameters. ■

The equivalence of cross-entropy, D_{KL} divergence , and maximum likelihood as learning objectives provides intuition about our many goals when learning $p_{\boldsymbol{\theta}}$: (1) we want a close (w.r.t. a given metric) approximation of the data-generating distribution, and (2) this approximation should place high probability on samples of real language data.

Properties of $\widehat{\boldsymbol{\theta}}$ under the cross-entropy loss. Assumption 3.2.2 may feel quite strong, as it implies we know a great deal about the nature of p_{LM}. However, it allows us to prove the optimality of $p_{\hat{\theta}}$ under certain conditions.

Theorem 3.2.1. *Consider that our loss function $\ell(\boldsymbol{\theta}^\star, \boldsymbol{\theta}) = \mathrm{H}(\widetilde{p_{\theta^\star}}, p_{\boldsymbol{\theta}})$ (or equivalently that $\ell(\boldsymbol{\theta}^\star, \boldsymbol{\theta}) = D_{\mathrm{KL}}(\widetilde{p_{\theta^\star}} \,||\, p_{\boldsymbol{\theta}})$). Given Assumption 3.2.1 and that the minimizer $p_{\hat{\theta}}$ of $\mathrm{H}(\widetilde{p_{\theta^\star}}, p_{\boldsymbol{\theta}})$ is unique, then under certain (quite strong) regularity conditions on $\{p_{\boldsymbol{\theta}} \mid \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$, $\widehat{\boldsymbol{\theta}}$ is a consistent estimator, i.e., it converges to $\boldsymbol{\theta}^\star$ in probability as $n \to \infty$.*

Arguably, in practice, Assumption 3.2.2 does not hold; we often make some incorrect modeling assumptions. Naturally, this raises the following question: If we *mis*specify the family of models that p_{LM} belongs to, i.e., $p_{\mathrm{LM}} \notin \{p_{\boldsymbol{\theta}} \mid \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$, then is our optimal model $p_{\hat{\theta}}$ under the cross-entropy loss at all meaningful? Fortunately, the answer here is yes. In this case, we can interpret $p_{\hat{\theta}}$ as a projection of p_{LM} onto the manifold of parametric models $\{p_{\boldsymbol{\theta}} \mid \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$. This projection is formally known as an **information projection** (Nielsen, 2018), which while we do not cover formally here, we can intuit as a mapping of p_{LM} onto its "closest" point in $\{p_{\boldsymbol{\theta}} \mid \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$. In this setting, using different metrics M leads to different definitions of closeness, which in turn means that optimal models under different M exhibit different properties.

Potential drawbacks of cross-entropy loss. A closer inspection of Eq. (3.56) reveals that, when we use $\mathrm{H}(\widetilde{p_{\theta^\star}}, p_{\boldsymbol{\theta}})$ as our loss function, $p_{\boldsymbol{\theta}}$ must put probability mass on *all* samples $\boldsymbol{y}^{(n)}$ in the support of $\widetilde{p_{\theta^\star}}$; otherwise, our loss is infinite. Since the model is not explicitly penalized for extraneous coverage, it will thus resort to placing mass over all of Σ^* to avoid such gaps;[12] this is sometimes referred to as *mean-seeking* behavior. In practice, this means that sequences of symbols that one might qualitatively describe as gibberish are assigned nonzero probability by $p_{\boldsymbol{\theta}}$. It is unclear whether this is a desirable property under a language model. While perhaps useful when using such a model to assign probabilities to strings—in which case, we might be more interested in how strings' probabilities rank against each other and may not want to write off any string as completely improbable—it could prove problematic when generating strings from these models, a topic covered later in this course.

Teacher Forcing. The loss functions that we have considered thus far are all based on our model's predictions conditioned on prior context. Here we are faced with a choice during training: we could either use the model's predictions from the previous time step(s) $p_{\boldsymbol{\theta}}(\cdot \mid \widehat{\boldsymbol{y}}_{<t})$ (e.g., the most probable symbols) as the prior context or use the ground-truth prior context from our data $p_{\boldsymbol{\theta}}(\cdot \mid \boldsymbol{y}_{<t})$. The latter method is often referred to as **teacher forcing**: Even if our model makes an incorrect prediction at one step of training, we intervene and provide the correct answer for it to make subsequent predictions with.

From a theoretical perspective, training with the cross-entropy loss mandates that we should use the teacher-forcing approach since each conditional distribution is defined with respect to the ground-truth context; this is elucidated, for example, in Eq. (3.57). Yet such meticulous guidance can lead to poor performance in tasks where the model is required to accurately predict an entire sequence of symbols on its own. For example, in language generation, since the model is not exposed to its own generations during training, small errors in predictions can compound, leading to degenerate text. This problem is known as **exposure bias**. Only the other hand, using previous model outputs in order to make subsequent predictions can lead to serious instability during training, especially if implemented from the start of training. Methods for alleviating exposure bias have been proposed with more stable training dynamics, such as scheduled sampling Bengio et al. (2015), which we discuss in §3.2.2.

Alternative Objectives

Masked Language Modeling. So far, our parameter estimation strategies have made use of the decomposition of $p_{\boldsymbol{\theta}}(\boldsymbol{y})$ into individual symbol probabilities, conditioned on *prior* symbols, i.e., $p_{\boldsymbol{\theta}}(\boldsymbol{y}) = \prod_{t=1}^{T} p_{\boldsymbol{\theta}}(y_t \mid \boldsymbol{y}_{<t})$. In other words, we do not give a model both sides of a symbol's context when asking it to estimate the probability distribution over that symbol. While this paradigm might be

[12] This behavior can also be (at least partially) attributed to the softmax used to transform model outputs into a probability distribution over symbols. Since the softmax maps to the interior of the probability simplex, no symbol can be assigned a probability of exactly 0.

more realistic when using a language model for tasks such as generation—for which we may want to generate outputs sequentially to mimic human language production—access to both sides of a symbol's context could be critical when using the model for tasks such as acceptability judgments or classification. This motivates the use of an alternative objective for parameter estimation.

Similarly to the maximum likelihood objective in Eq. (3.59), we can choose model parameters by optimizing for the per-symbol log-likelihood of a dataset $\mathcal{D}$, albeit in this case, using *both* sides of the symbol's context:

$$\mathcal{L}_{\texttt{MLM}}(\boldsymbol{\theta}) = \sum_{n=1}^{N} \sum_{t=1}^{T} \log p_{\boldsymbol{\theta}}({y_t}^{(n)} \mid \boldsymbol{y}_{<t}^{(n)}, \boldsymbol{y}_{>t}^{(n)}) \tag{3.66}$$

Eq. (3.66) is sometimes referred to as the **pseudo(log)likelihood** (Besag, 1975), since it gives us an approximation of the true log-likelihood, i.e., $\sum_{t=1}^{T} \log p_{\boldsymbol{\theta}}(y_t \mid \boldsymbol{y}_{<t}, \boldsymbol{y}_{>t}) \approx \log p_{\boldsymbol{\theta}}(\boldsymbol{y})$. Pseudolikelihood has its origins in thermodynamics, where it was used as an approximate inference technique for parameter estimation in Ising models. In such situations, computing $p_{\boldsymbol{\theta}}(y_t \mid \boldsymbol{y}_{\neq t})$ often proved computationally easier than computing the exact set of of conditional probabilities whose product equaled the marginal.

Using Eq. (3.66) as a model's training objective is also motivated by psychological tests of language understanding—specifically, the Cloze (Taylor, 1953) task in psychology, in which the goal is to predict the omitted symbol from a piece of text that constitutes a logical and coherent completion. For example, in the string

Example 3.2.1. *The students* `[MASK]` *to learn about language models.*

we predict *want* or *like* with high probability for the `[MASK]` position. When used as an objective in NLP, estimating the probability distribution over symbols at the masked position is referred to as masked language modeling; BERT (Devlin et al., 2019) is one well known example of a masked language model. In practice, typically only the distributions over symbols at a percentage of randomly-chosen positions in $\mathcal{D}$ are estimated during training. As mentioned in §2.5, a model whose parameters are estimated with the masked language modeling objective is not a valid language model in the sense of Definition 2.3.7 because it does not provide a valid distribution over Σ^*. Yet, masked language models have become increasingly popular as base models for fine-tuning on certain downstream tasks, where they sometimes lead to superior performance over standard language models.

Other Divergence Measures. From a given hypothesis space (see Assumption 3.2.2), the distribution that minimizes a given divergence measure with $p_{\boldsymbol{\theta}^\star}$ exhibits certain properties with respect to how probability mass is spread over the support of that distribution. For example, the model $p_{\boldsymbol{\theta}}$ that minimizes $D_{\mathrm{KL}}(p_{\boldsymbol{\theta}^\star} \,||\, p_{\boldsymbol{\theta}})$ exhibits *mean-seeking* behavior, as discussed earlier in this section. These properties have been studied in depth by a number of works (Minka, 2005;

Theis et al., 2016; Huszár, 2015; Labeau and Cohen, 2019). The implication of these findings is that, depending on the use case for the model, other divergence measures may be better suited as a learning objective. For example, prior work has noted frequency biases in models estimated using the standard log-likelihood objective, i.e., these models exhibit an inability to accurately represent the tails of probability distributions (Gong et al., 2018). This is particularly relevant in the case of language modeling, as symbol-usage in natural language tends to follow a power-law distribution (Zipf, 1935). Consequently, when we care particularly about accurately estimating the probability of rare words, we may wish to instead use a loss function that prioritizes good estimation of probability distribution tails. On the other hand, in the case of language generation, we may desire models that only assign probability mass to outputs that are highly-likely according to $p_{\theta^\star}$, even if this means assigning probabilities of 0 to some outcomes possible under $p_{\theta^\star}$. In other words, we may want a model with *mode-seeking* behavior, which is characteristic of models trained to minimize $D_{\mathrm{KL}}(p_{\boldsymbol{\theta}} \mid\mid p_{\theta^\star})$. However, there are a number of computational issues with using other divergence measures—such as general power divergences, reverse D_{KL} divergence, and total variation distance—for training neural probabilistic models over large supports, making them difficult to work with in practice. For example, we can compute a Monte Carlo estimate of the forward D_{KL} divergence simply by using samples from $p_{\theta^\star}$, which is exactly what we have in our dataset. However an unbiased estimator of the reverse D_{KL} divergence would require the ability to query $p_{\theta^\star}$ for probabilities, which we do not have.

Scheduled Sampling and Alternative Target Distributions. Scheduled sampling (Bengio et al., 2015) is an algorithm proposed with the goal of alleviating exposure bias: after an initial period of training using the standard teacher forcing approach, some percentage of the models' predictions are conditioned on prior model outputs, rather than the ground-truth context. However, under this algorithm, $\widehat{\boldsymbol{\theta}}$ does not lead to a consistent estimator of $\boldsymbol{\theta}^\star$ (Huszár, 2015). Other methods likewise aim to alleviate the discrepancy between settings during parameter estimation and those at inference time by specifying an alternative target distribution, for example, one that ranks "higher-quality" text as more probable than average-quality text. Ultimately, these methods often make use of techniques developed for reinforcement learning, i.e., the REINFORCE algorithm. These methods fall under the category of fine-tuning criterion, which are discussed later in this course.

Auxiliary Prediction Tasks. Certain works jointly optimize for an additional objective when performing parameter estimation. For example, the parameters for BERT were learned using both the masked language modeling objective as well as a task referred to as next sentence prediction, i.e., given two sentences, estimating the probability that the second sentence followed the first in a document. A number of similar auxiliary tasks have subsequently been proposed, such as symbol frequency prediction or sentence ordering (see Aroca-Ouellette

and Rudzicz (2020) for summary). However, these tasks do not have a formal relationship to language modeling and it is unclear what their effects are on a model's ability to serve as a valid probability distribution over strings. They likely lead to models that no longer fulfill the formal criteria of §2.5.4.

3.2.3 Parameter Estimation

Given a loss function ℓ and a parameter space $\boldsymbol{\Theta}$ from which to choose model parameters, we are now tasked with *finding* the parameters $\widehat{\boldsymbol{\theta}}$, i.e., solving Eq. (3.53). For the class of models that we consider (those parameterized by large neural networks), finding an exact solution analytically would be impractical, if not impossible. Thus, we must resort to numerical methods, where we find approximately optimal parameters by iterating over solutions. This is known as **parameter estimation**, or more colloquially as **training** our model.

Here we will review the various components of training a language model from start to finish. Many of the techniques used for training language models are generally applicable machine learning techniques, e.g., gradient-descent algorithms. Further, these techniques are constantly evolving and often viewed as trade secrets, meaning that entities building and deploying models may not reveal the combination of components that they employed. Thus, we give a more general overview of the design choices involved in parameter estimation, along with the characteristics common to most components.

Data Splitting

In any machine learning setting, we may overestimate model quality if we evaluate solely on its performance w.r.t. the data on which its parameters were estimated. While we can often construct a model that performs arbitrarily well on a given dataset, our goal is to build a model that generalizes to unseen data. Thus, it is important to measure the final performance of a model on data that has had no influence on the choice of model parameters.

This practice can be accomplished simply by splitting the data into several sets. The two basic data splits are a training set $\mathcal{D}_{\text{train}}$ and test set $\mathcal{D}_{\text{test}}$; as the names imply, the training set is used during parameter estimation while the test set is used for evaluating final performance. When samples from $\mathcal{D}_{\text{test}}$ can be found in $\mathcal{D}_{\text{train}}$, we call this **data leakage**. The training set can be further divided to produce a validation set $\mathcal{D}_{\text{val}}$. Typically, $\mathcal{D}_{\text{val}}$ is not used to define the objective for which parameters are optimized. Rather, it serves as a check during training for the generalization abilities of a model, i.e., to see whether the model has started overfitting to the training data. The validation set can be used, e.g., to determine when to stop updating parameters.

Numerical Optimization

From a starting point $\boldsymbol{\theta}_0 \in \boldsymbol{\Theta}$ chosen according to our initialization strategy, we want to find $\widehat{\boldsymbol{\theta}}$ in an efficient manner. This is where numerical **optimization**

algorithms come into play—a precise set of rules for choosing how to move within $\boldsymbol{\Theta}$ in order to find our next set of parameters. The output of a numerical optimization algorithm is a sequence of iterates $\{\boldsymbol{\theta}_s\}_{t=0}^{T}$, with the property that as $T \to \infty$ we find the minimizer of our objective ℓ. Ideally, even after a finite number of iterations, we will be sufficiently close to $\widehat{\boldsymbol{\theta}}$.

The basic algorithm for searching the parameter space for $\widehat{\boldsymbol{\theta}}$ follows a simple formula: starting from $\boldsymbol{\theta}_0 \in \boldsymbol{\Theta}$, we iteratively compute $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots$ as

$$\boldsymbol{\theta}_{s+1} = \boldsymbol{\theta}_s + \text{update magnitude} \times \text{update direction}. \tag{3.67}$$

where the update added to $\boldsymbol{\theta}_s$ to obtain $\boldsymbol{\theta}_{s+1}$ is intended to move us closer to $\widehat{\boldsymbol{\theta}}$. Once some maximum number of updates S or a pre-defined desideratum has been met, e.g., our loss has not improved in subsequent iterations, we stop and return the current set of parameters. Many of the numerical optimization techniques in machine learning are gradient-based, i.e., we use the gradient of the objective with respect to current model parameters (denoted as $\nabla_{\boldsymbol{\theta}_s}\ell(\boldsymbol{\theta}_s)$) to determine our update direction. Standard vanilla gradient descent takes the form of §3.2.3, where the learning rate schedule $\boldsymbol{\eta} = \langle \eta_0, \cdots, \eta_T \rangle$ determines the step size of our parameter update in the loss minimizing direction—there is an inherent trade-off between the rate of convergence and overshooting—and the stopping criterion C determines whether we can terminate parameter updates before our maximum number of iterations S. In vanilla gradient descent, we set

Algorithm 1 Gradient descent for parameter optimization.

Input: ℓ objective
$\boldsymbol{\theta}_0$ initial parameters
$\boldsymbol{\eta}$ learning rate schedule
$C\colon \ell \times \boldsymbol{\Theta} \times \boldsymbol{\Theta} \to \{\text{True}, \text{False}\}$ stopping criterion

1. **for** $s = 0, \cdots, S$:
2. $\quad \boldsymbol{\theta}_{s+1} \leftarrow \boldsymbol{\theta}_s - \eta_s \cdot \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}_s)$
3. $\quad$ **if** $C(\ell, \boldsymbol{\theta}_s, \boldsymbol{\theta}_{s-1})$:
4. $\quad\quad$ **break**
5. **return** $\boldsymbol{\theta}_s$

$\boldsymbol{\eta} = c \cdot \mathbf{1}$ for some constant c and $C(\ell, \boldsymbol{\theta}_s, \boldsymbol{\theta}_{s-1}) = \mathbb{1}\{|\ell(\boldsymbol{\theta}_s) - \ell(\boldsymbol{\theta}_{s-1})| < \epsilon\}$ for user-chosen ϵ—in words, we stop when the change in loss between parameter updates is below a chosen threshold. In practice, more sophisticated learning rate schedules η, e.g., square-root functions of the timestep (Hoffer et al., 2017) or adaptive functions that take into account model parameter values (Duchi et al., 2011), and stopping criterion C are employed.

Modern training frameworks rely on backpropagation—also known as reverse-mode automatic differentiation (Griewank and Walther, 2008)—to compute gradients efficiently (and, as the name implies, automatically!). In fact, gradients can be computed using backpropogation in the same complexity as evaluation of the original function. We do not provide a formal discussion of backpropagation here but see Griewank and Walther (2008) for this material.

Recall that our loss function—and consequently the gradient of our loss function—is defined with respect to the *entire* dataset. Vanilla gradient descent therefore requires iterating through all of $\mathcal{D}_{\text{train}}$ in order to determine the direction to move parameters U, which is an incredibly time-consuming computation for the large datasets employed in modern machine learning settings. Rather, an optimization algorithm would likely take much less time to converge if it could rapidly compute estimates of the gradient at each step. This is the motivation behind perhaps the most widely employed class of optimization algorithms in machine learning: variations of **stochastic gradient descent** (SGD), such as mini-batch gradient descent. Explicitly, these algorithms make use of the fact that $\mathbb{E}_{\mathcal{D}' \overset{\text{i.i.d.}}{\sim} \mathcal{D}} \nabla_{\boldsymbol{\theta}}(\boldsymbol{\theta}, \mathcal{D}') = \nabla_{\boldsymbol{\theta}}(\boldsymbol{\theta}, \mathcal{D})$, where in slight abuse of notation, we use $\mathcal{D}' \overset{\text{i.i.d.}}{\sim} \mathcal{D}$ to signify that the multi-set $\mathcal{D}'$ consists of random i.i.d. samples from $\mathcal{D}$. Thus we can instead base our loss ℓ, and consequently U, off of a randomly selected subset of the data.[13] In practice though, this sample is taken *without* replacement, which breaks the i.i.d. assumption. This in turn implies this our gradient estimates are biased under the mini-batch gradient descent algorithm. However, this bias does not seem to empirically harm the performance of such optimization strategies. Indeed, an entire branch of machine learning called **curriculum learning** focuses on trying to find an optimal data ordering with which to train models to achieve desirable characteristics such as generalization abilities. Even when orderings are randomly selected, the chosen ordering can have a large impact on model performance Dodge et al. (2020).

A number of optimization algorithms have since iterated on SGD, e.g., the momentum algorithm (Polyak, 1964). In short, the momentum algorithm computes an exponentially decaying moving average of past gradients, and continues updating parameters in this direction, which can drastically speed up convergence. A widely-employed optimization algorithm called ADAM (Kingma and Ba, 2015) takes a similar approach. Just as in momentum, it computes update directions using a moving average (first moment) of gradients, albeit it additionally makes use of the variance of gradients (second moment) when computing update directions. ADAM is one of the most popular optimization algorithms used for training large language models in modern ML frameworks.

Parameter Initialization

Our search for (approximately) optimal model parameters must start from some point in the parameter space, which we denote as $\boldsymbol{\theta}_0$. Ideally, starting from any point would lead us to the same solution, or at least to solutions of similar quality. Unfortunately, this is not the case: both training dynamics and the performance

[13] While this logic holds even for samples of size 1 (which is the sample size for standard SGD by definition), basing updates off of single samples can lead to noisy updates. Depending on resource constraints, batch sizes of a few hundred are often used, leading to much more stable training (although in the face of memory constraints, larger batch sizes can be mimicked by accumulating, i.e., averaging, gradients across multiple batches when computing update directions). Batch size itself is often viewed as a model hyperparameter that can have a significant effect on model performance.

of the final model can depend quite heavily on the chosen initialization strategy, and can even have high variance between different runs of the same strategy. This makes sense at some intuitive level though: depending on the learning algorithm, an initial starting point can heavily dictate the amount of searching we will have to do in order to find $\widehat{\boldsymbol{\theta}}$, and how many local optima are on the route to $\widehat{\boldsymbol{\theta}}$. Consequently, a poor initial starting point may lead to models that take longer to train and/or may lead our learning algorithm to converge to sub-optimal solutions (i.e., an alternative local optimum) (Dodge et al., 2020; Sellam et al., 2022). This can be the case even when only estimating the final layer of a network, e.g., when building a classifier by appending a new layer to a pretrained model—a recent, widely-adopted practice in NLP (Dodge et al., 2020).

Methods for initializing the parameters of neural language models are largely the same as those for initializing other neural networks. Perhaps the simplest approach is to randomly initialize all parameters, e.g., using a uniform or normal random variable generator. The parameters of these generators (mean, standard deviation, bounds) are considered hyperparameters of the learning algorithm. Subsequent methods have iterated on this strategy to develop methods that take into account optimization dynamics or model architectures. One consideration that is particularly relevant for language models is that the input and output sizes of the embedding layer and the fully connected layer can be very different; this exacerbate the problem of vanishing or exploding gradients during training. For example, Glorot, Xavier and Bengio, Yoshua (2010) proposed Xavier init, which keeps the variance of the input and output of all layers within a similar range in order to prevent vanishing or exploding gradients; He et al. (2015) proposed a uniform initialization strategy specifically designed to work with ReLU activation units. Using uniform random variables during parameter initialization can likewise alleviate the problem of vanishing gradients. While most deep learning libraries use thoughtfully-selected initialization strategies for neural networks, it is important to internalize the variance in performance that different strategies can cause.

Early Stopping

As previously discussed, performance on $\mathcal{D}_{\text{train}}$ is not always the best indicator of model performance. Rather, even if our objective continues to increase as we optimize over model parameters, performance on held-out data, i.e., $\mathcal{D}_{\text{test}}$ or even $\mathcal{D}_{\text{val}}$, may suffer as the model starts to overfit to the training data. This phenomenon inspires a practice called **early stopping**, where we stop updating model parameters before reaching (approximately) optimal model parameter values w.r.t. $\mathcal{D}_{\text{train}}$. Instead, we base our stopping criterion C off of model performance on $\mathcal{D}_{\text{val}}$ as a quantification of generalization performance, a metric other than that which model parameters are optimized for, or just a general slow down in model improvement on the training objective.

Early stopping sacrifices better training performance for better generalization performance; in this sense, it can also be viewed as a regularization technique,

a topic which we discuss next. As with many regularization techniques, early stopping can have adverse effects as well. Recent work suggests that many models may have another period of learning after an initial period of plateauing train/validation set performance. Indeed, a sub-field has recently emerged studying the "grokking" phenomenon (Power et al., 2022), when validation set performance suddenly improves from mediocre to near perfect after a long period in which it appears that model learning has ceased, or even that the model has overfit to the training data. Thus, it is unclear whether early stopping is always a good practice.

3.2.4 Regularization Techniques

Our goal during learning is to produce a model $p_{\boldsymbol{\theta}}$ that generalizes beyond the observed data; a model that perfectly fits the training data but produces unrealistic estimates for a new datapoint is of little use. Exactly fitting the empirical distribution is therefore perhaps not an ideal goal. It can lead to **overfitting**, which we informally define as the situation when a model uses spurious relationships between inputs and target variables observed in training data in order to make predictions. While this behavior decreases training loss, it generally harms the model's ability to perform on unseen data, for which such spurious relationships likely do not hold.

To prevent overfitting, we can apply some form of regularization.

Principle 3.2.3 (Regularization)**.** *Regularization is a modification to a learning algorithm that is intended to increase a model's generalization performance, perhaps at the cost of training performance.*[14]

There are many ways of implementing regularization, such as smoothing a distribution towards a chosen baseline or adding a penalty to the loss function to reflect a prior belief that we may have about the values model parameters should take on Hastie et al. (2001); Bishop (2006). Further, many regularization techniques are formulated for specific model architecture: for example, the count-based smoothing methods used n-gram language models (**??**). Here we specifically consider the forms of regularization often used in the estimation of neural language models. Most fall into two categories: methods that try to ensure a model's robustness to yet unseen (or rarely seen) inputs—e.g., by introducing noise into the optimization process—or methods that add a term to our loss function that reflects biases we would like to impart on our model. This is by no means a comprehensive discussion of regularization techniques, for which we refer the reader to Ch.7 of Goodfellow et al. (2016).

Weight Decay

A bias that we may wish to impart on our model is that not all the variables available to the model may be necessary for an accurate prediction. Rather, we

[14] Adapted from Goodfellow et al. (2016), Ch. 7.

hope for our model to learn the simplest mapping from inputs to target variables, as this is likely the function that will be most robust to statistical noise.[15] This bias can be operationalized using regularization techniques such as weight decay (Goodfellow et al., 2016)—also often referred to as ℓ_2 regularization. In short, a penalty for the ℓ_2 norm of $\boldsymbol{\theta}$ is added to ℓ. This should in theory discourage the learning algorithm from assigning high values to model parameters corresponding to variables with only a noisy relationship with the output, instead assigning them a value close to 0 that reflects the a non-robust relationship.

Entropy Regularization

One sign of overfitting in a language model $p_{\boldsymbol{\theta}}$ is that it places effectively all of its probability mass on a single symbol.[16] Rather, we may want the distributions output by our model to generally have higher entropy, i.e., following the principle of maximum entropy: "the probability distribution which best represents the current state of knowledge about a system is the one with largest entropy, in the context of precisely stated prior data" Jaynes (1957). Several regularization techniques, which we refer to as **entropy regularizers**, explicitly penalize the model for low entropy distributions.

Label smoothing (Szegedy et al., 2015) and the confidence penalty (Pereyra et al., 2017) add terms to ℓ to penalize the model for outputting peaky distributions. Explicitly, label smoothing reassigns a portion of probability mass in the reference distribution from the ground truth symbol to all other symbols in the vocabulary. It is equivalent to adding a term $D_{\mathrm{KL}}(u \mid\mid p_{\boldsymbol{\theta}})$ to ℓ, where u is the uniform distribution. The confidence penalty regularizes against low entropy distribution by adding a term $\mathrm{H}(p_{\boldsymbol{\theta}})$ to ℓ that encourage high entropy in model outputs. The general class of entropy regularizers have proven effective in training neural models Meister et al. (2020).

Dropout

Regularization also encompasses methods that expose a model to noise that can occur in the data at inference time. The motivation behind such methods is both to penalize a model for being overly dependent on any given variable (whether directly from the input or somewhere further along in the computational graph) for making predictions. Dropout does this explicitly by randomly "dropping" variables from a computation in the network (Srivastava et al., 2014).

More formally, consider a model defined as a series of computational nodes, where any given node is the product of the transformation of previous nodes. When dropout is applied to the module that contains that node, then the node is zeroed out with some percentage chance, i.e., it is excluded in all functions that may make use of it to compute the value of future nodes. In this case, the

[15] This philosophy can be derived from Occam's Razor, i.e., the principle that one should search for explanations constructed using the smallest possible set of elements.

[16] The softmax transformation serves as somewhat of a regularizer against this behavior since it does not allow any symbol be assigned a probability of 0.

model will be penalized if it relied completely on the value of that node for any given computation in the network. Dropout can be applied to most variables within a model, e.g., the inputs to the model itself, the inputs to the final linear projection in a feed-forward layer, or the summands in the attention head of a Transformer. Note that at inference time, all nodes are used to compute the model's output.[17]

Batch and Layer Normalization

Rescaling variables within a network helps with training stability, and further, with generalization by keeping variables within the same range and with unit variance. Specifically, batch normalization helps regularize the problem of covariate shift, where the distribution of features (both the input features and the variables corresponding to transformed features within a network) differs between the training data and the data at inference time. Batch normalization alleviates this problem by recentering (around 0) and scaling (such that data points have unit variance) data points such that the data flowing between intermediate layers of the network follows approximately the same distribution between batches. Layer normalization likewise performs centering and rescaling, albeit across features rather than across data points. Specifically, normalization is performed so that all of the feature values within a data point have mean 0 and unit variance.

[17] Some form of renormalization is typically performed to account for the fact that model parameters are learned with only partial availability of variables. Thus when all variables are used in model computations, the scale of the output will (in expectation) be larger than during training, potentially leading to poor estimates.

Chapter 4

Classical Language Models

Next we turn to two classical language modeling frameworks: finite-state language models (a natural generalization of the well-known n-gram models) in §4.1 and pushdown language models §4.2. Although the most successful approaches to language modeling are based on neural networks, the study of older approaches to language modeling is invaluable. First, due to the simplicity of the models, learning how they work helps distill concepts. And, moreover, they often serve as important baselines in modern NLP and provide very useful insights into the capabilities of modern architectures as we will see when we discuss modern architectures in Chapter 5.

In the spirit of our question-motivated investigation, we will focus on the following two questions.

Question 4.0.1. *How can we tractably represent all conditional distributions of the form $p_{SM}(y \mid \boldsymbol{y})$ in a simple way?*

Question 4.0.2. *How can we tractably represent the hierarchical structure of human language?*

4.1 Finite-state Language Models

After rigorously defining what language models are (and what they are not) and discussing how we can estimate them, it is time to finally introduce our first class of language models—those based on finite-state automata. Language models derived from probabilistic finite-state automata are some of the simplest classes of language models because they definitionally distinguish a *finite* numbers of contexts when modeling the conditional distribution of the next possible symbol $p_{\text{M}}(y \mid \boldsymbol{y})$. We first give an intuitive definition of a finite-state language model and then introduce a more formal definition, which we will use throughout the rest of the section.

Definition 4.1.1 (Informal definition of a finite-state language model)**.** *A language model p_{LM} is **finite-state** if it defines only finitely many unique conditional*

distributions $p_{LM}(y \mid \boldsymbol{y})$. In other words, there are only finitely many contexts $\boldsymbol{y}$ which define the distribution over the next symbol, $p_{LM}(y \mid \boldsymbol{y})$.

Intuitively, this framework might be useful because it *bounds* the number of unique conditional distributions we have to learn. However, as we will see later in this chapter, finite-state language models are not sufficient for modeling human language. Nevertheless, they can still offer a baseline for modeling more complex phenomena. They also offer a useful theoretical tool in the understanding of neural language models, which we will discuss in Chapter 5.

4.1.1 Weighted Finite-state Automata

Before we introduce finite-state *language models*, we go on a brief detour into the theory of finite-state *automata*. As we will see, finite-state automata are a tidy and well-understood formalism. As we will see later in §5.1.5, they also provide a solid and convenient theoretical framework for understanding modern neural language models, e.g., those based on recurrent neural networks and transformers. We, therefore, begin by briefly introducing the theory of finite-state automata with real-valued weights.

Finite-state Automata

In words, finite-state automata are one of the simplest devices for defining a formal language (cf. Definition 2.3.5). We give a formal definition below.

Definition 4.1.2 (Finite-state Automata)**.** *A **finite-state automaton** (FSA) is a 5-tuple $(\Sigma, Q, I, F, \delta)$ where*

- *Σ is an alphabet;*
- *Q is a finite set of states;*
- *$I \subseteq Q$ is the set of initial states;*
- *$F \subseteq Q$ the set of final or accepting states;*
- *A finite multiset $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$.[1] Elements of δ are generally called **transitions**.*

The name, *finite-state* automaton, stems from the requirement that the set of states Q is finite, which stands in contrast to the remaining formalisms we will cover in this course, e.g., pushdown automata and recurrent neural networks. We will denote a general finite-state automaton with a (subscripted) $\mathcal{A}$. We will also adopt a more suggestive notation for transitions by denoting a transition (q_1, a, q_2) as $q_1 \xrightarrow{a} q_2$.

[1] The fact that it is a multiset reflects that it can contain multiple copies of the same element (i.e., transitions between the same pair of states with the same symbol).

An FSA can be graphically represented as a labeled, directed multi-graph.[2] The vertices in the graph represent the states $q \in Q$ and the (labeled) edges between them the transitions in δ. The labels on the edges correspond to the input symbols $a \in \Sigma$ which are consumed when transitioning over the edges. The initial states $q_I \in I$ are marked by a special incoming arrow while the final states $q_F \in F$ are indicated using a *double* circle.

Example 4.1.1. *An example of an FSA can be seen in Fig. 4.1. Formally, we can specify it as*

- $\Sigma = \{a, b, c\}$
- $Q = \{1, 2, 3\}$
- $I = \{1\}$
- $F = \{3\}$
- $\delta = \{(1, a, 2), (1, b, 3), (2, b, 2), (2, c, 3)\}$

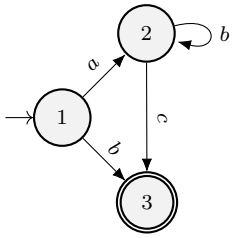


Figure 4.1: Example of a simple FSA.

A finite-state automaton sequentially reads in individual symbols of an **input string** $\boldsymbol{y} \in \Sigma^*$ and transitions from state to state according to the transition function δ. The traversal through the automaton starts in a state $q_I \in I$ (more precisely, it acts as if starting from all of them in parallel). It then transitions from state q into the state q' upon reading the symbol a if and only if $q \xrightarrow{a} q' \in \delta$. ε-labeled transitions, however, allow a finite-state machine to transition to a new state without consuming a symbol. This is in line with ε's definition as an empty string.

A natural question to ask at this point is what happens if for a state–symbol pair (q, a) there is *more than one* possible transition allowed under the relation δ. In such a case, we take *all* implicit transitions simultaneously, which leads us to a pair of definitions.

Definition 4.1.3. *A FSA* $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ *is* ***deterministic*** *if*

[2]The *multi-* aspect of the multi-graph refers to the fact that we can have multiple transitions from any pair of states and labeled refers to the fact that we label those transitions with symbols from the alphabet Σ.

- *it does not have any ε-transitions;*
- *for every $(q, a) \in Q \times \Sigma$, there is at most one $q' \in Q$ such that $q \xrightarrow{a} q' \in \delta$;*
- *there is a single initial state, i.e., $|I| = 1$.*

*Otherwise, $\mathcal{A}$ is **non-deterministic**.*

An important, and perhaps not entirely obvious, result is that the classes of deterministic and non-deterministic FSA are *equivalent*, in the sense that you can always represent a member of one class with a member of the other.

If the automaton ends up, after reading in the last symbol of the input string, in one of the final states $q_F \in F$, we say that the automaton **accepts** that string. A finite-state automaton is therefore a computational device that determines whether a string satisfies a condition (namely, the condition that the automaton, by starting in an initial state and following one of the paths labeled with that string, ends in a final state). A string that satisfies this condition is said to be *recognized* by the automaton and the set of all strings satisfying this condition form the *language* of the automaton.[3]

Definition 4.1.4. *Let $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ be an finite-state automaton. The **language** of $\mathcal{A}$, $L(\mathcal{A})$ is defined as*

$$L(\mathcal{A}) \stackrel{\text{def}}{=} \{\boldsymbol{y} \mid \boldsymbol{y} \text{ is recognized by } \mathcal{A}\} \tag{4.1}$$

Abstractly, a finite-state automaton is hence a specification of a set of *rules* that strings must satisfy to be included in its language. The set of languages that finite-state automata can recognize is known as the class of **regular languages**.

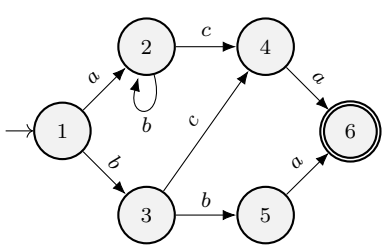
Definition 4.1.5. *A language $L \subseteq \Sigma^*$ is **regular** if and only if it can be recognized by an unweighted finite-state automaton, i.e., if there exists a finite-state automaton $\mathcal{A}$ such that $L = L(\mathcal{A})$.*

Example 4.1.2. *Additional simple examples of FSAs are shown in Fig. 4.2. The FSA in Fig. 4.2a, for example, can formally be defined with*

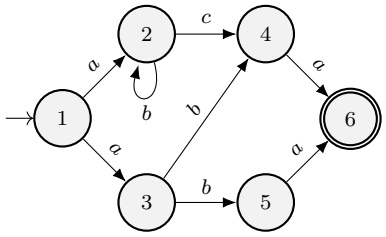
- $\Sigma = \{a, b, c\}$
- $Q = \{1, 2, 3, 4, 5, 6\}$
- $I = \{1\}$
- $F = \{6\}$
- $\delta = \{(1, a, 2), (1, b, 3), (2, b, 2), (2, c, 4), (3, c, 4), (3, b, 5), (4, a, 6), (5, a, 6)\}$

The FSA in Fig. 4.2a is deterministic while the one in Fig. 4.2b is non-deterministic. A few examples of strings accepted by the $\mathcal{A}_1$ include bba, bca,

[3] We also say that the automaton *recognizes* this set of strings (language).



(a) A deterministic FSA, $\mathcal{A}_1$. Each state only has one outgoing transition labeled with the same symbol.



(b) A non-deterministic FSA, $\mathcal{A}_2$. State 1 has two outgoing transitions labeled with a whereas state 3 has two outgoing transitions labeled with b.

Figure 4.2: Examples of a deterministic and a non-deterministic FSA.

aca, abca, abbca, abbbca, ... In fact, due to the self-loop at state 2*, the symbol b can appear an arbitrary number of times at position 2 in the accepted string abca. Notice that, starting from the state* 1 *and following the transitions dictated by any of the accepted strings, we always end up in the only final state, state* 6*. In particular, the string "abbca" is accepted with the following set of transitions in* $\mathcal{A}_1$*:*

$$1 \xrightarrow{a} 2, 2 \xrightarrow{b} 2, 2 \xrightarrow{b} 2, 2 \xrightarrow{yc} 4, 4 \xrightarrow{a} 6.$$

Weighted Finite-state Automata

A common and very useful augmentation to finite-state automata is through the addition of *weights* on the transitions. The general theory of weighted automata makes use of semiring theory, which is beyond the scope of this course.[4] In this course, we will limit ourselves to the study of automata with real-valued weights.

Definition 4.1.6 (Real-weighted Finite-State Automaton)**.** *A* ***real-weighted finite-state automaton*** *(WFSA)* $\mathcal{A}$ *is a 5-tuple* $(\Sigma, Q, \delta, \lambda, \rho)$ *where*

- Σ *is a finite alphabet;*
- Q *is a finite set of states;*
- $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \mathbb{R} \times Q$ *a finite multiset of transitions;*[5]
- $\lambda : Q \to \mathbb{R}$ *a weighting function over* Q*;*
- $\rho : Q \to \mathbb{R}$ *a weighting function over* Q*.*

[4] Semirings and semiring-weighted formal languages are covered in detail in the Advanced Formal Language Theory course offered at ETH as well.

[5] Again, we use the notation $q \xrightarrow{a/w} q'$ to denote $(q, a, w, q') \in \delta$.

Notice that we omit the initial and final state sets from the definition of WFSAs. Those can implicitly be specified by the states given non-zero initial or final weights by the λ and ρ functions, i.e., $I = \{q \in Q \mid \lambda(q) \neq 0\}$ and $F = \{q \in Q \mid \rho(q) \neq 0\}$. We might refer to them in the text later for notational convenience and clarity of exposition. We will also sometimes denote transition weights with $\mathcal{W}\left(q \xrightarrow{a/w} q'\right) \stackrel{\text{def}}{=} w$.

Graphically, we write the transition weights on the edges of the graph representing the WFSA after the output symbol, separated by a "/". The same separator is also used to separate the state name from its initial/final weights, which are written in the node.

Example 4.1.3. *Fig. 4.3 shows a ring-weighted version of the FSA we saw in Fig. 4.2a above.*

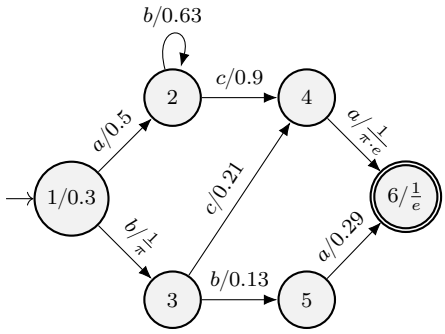


Figure 4.3: The WFSA corresponding to the FSA from Fig. 4.2a.

The connection of WFSAs to graphs makes it natural to define a set of transition matrices specified by a WFSA.

Definition 4.1.7. *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA. For any $a \in \Sigma$, we define the* ***symbol-specific transition matrix*** $\mathbf{T}^{(a)}$ *as the transition matrix of the graph restricted to a-labeled transitions. We also define the (full)* ***transition matrix*** *as $\mathbf{T} \stackrel{\text{def}}{=} \sum_{a \in \Sigma} \mathbf{T}^{(a)}$.*

Example 4.1.4. *Consider the WFSA $\mathcal{A}$ in Fig. 4.3. The (symbol-specific) transition matrices for $\mathcal{A}$ are*

$$\mathbf{T}^{(a)} = \begin{pmatrix} 0 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{\pi \cdot e} \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.29 & 0 \end{pmatrix}$$

$$\mathbf{T}^{(b)} = \begin{pmatrix} 0 & 0 & \frac{1}{\pi} & 0 & 0 & 0 \\ 0 & 0.63 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.13 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\mathbf{T}^{(c)} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.9 & 0 & 0 \\ 0 & 0 & 0 & 0.21 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\mathbf{T} = \mathbf{T}^{(a)} + \mathbf{T}^{(b)} + \mathbf{T}^{(c)} = \begin{pmatrix} 0 & 0.5 & \frac{1}{\pi} & 0 & 0 & 0 \\ 0 & 0.63 & 0 & 0.9 & 0 & 0 \\ 0 & 0 & 0 & 0.21 & 0.13 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{\pi \cdot e} \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.29 & 0 \end{pmatrix}$$

Paths and Path Weights

A path is an important concept when talking about (weighted) finite-state automata as it defines the basic structure by which a string is recognized or weighted. We now give a formal definition of a path and discuss how to weight paths.

Definition 4.1.8 (Path)**.** *A* ***path*** $\boldsymbol{\pi}$ *is an element of* δ^* *with* consecutive transitions*, meaning that it is of the form* $\left(q_1 \xrightarrow{\bullet/\bullet} q_2, q_2 \xrightarrow{\bullet/\bullet} q_3 \cdots q_{n-1} \xrightarrow{\bullet/\bullet} q_n\right)$*, where* $\bullet$ *is a placeholder.*[6] *The* ***length*** *of a path is the number of transition in it; we denote the length as* $|\boldsymbol{\pi}|$*. We use* q_1 *and* $n(\boldsymbol{\pi})$ *to denote the origin and the destination of a path, respectively. The* ***yield*** *of a path is the concatenation of the input symbols on the edges along the path, which we will mark with* $\mathsf{yield}(\boldsymbol{\pi})$*. Furthermore, we denote sets of paths with capital* Π*. Throughout the text, we will use a few different variants involving* Π *to avoid clutter:*

- $\Pi(\mathcal{A})$ *as the set of all paths in automaton* $\mathcal{A}$*;*
- $\Pi(\mathcal{A}, \boldsymbol{y})$ *as the set of all paths in automaton* $\mathcal{A}$ *with yield* $\boldsymbol{y} \in \Sigma^*$*;*
- $\Pi(\mathcal{A}, q, q')$ *as the set of all paths in automaton* $\mathcal{A}$ *from state* q *to state* q'*.*

One of the most important questions when talking about weighted formalisms like weighted finite-state automata is how to *combine* weights of atomic units like

[6] Notice we use the Kleene closure on the set δ here. It thus represents any sequence of transitions $\in \delta$

transitions into weights of complete *structures*.[7] We begin by multiplicatively combining the weights of individual transitions in a path into the weights of the full path.

Definition 4.1.9 (Path Weight)**.** *The* ***inner path weight*** $\mathrm{w}(\boldsymbol{\pi})$ *of a path* $\boldsymbol{\pi} = q_1 \xrightarrow{a_1/w_1} q_2 \cdots q_{N-1} \xrightarrow{a_N/w_N} q_N$ *is defined as*

$$\mathrm{w}_I(\boldsymbol{\pi}) = \prod_{n=1}^{N} w_n. \tag{4.2}$$

The ***(full) path weight*** *of the path* $\boldsymbol{\pi}$ *is then defined as*

$$\mathrm{w}(\boldsymbol{\pi}) = \lambda(p(\boldsymbol{\pi}))\, \mathrm{w}_I(\boldsymbol{\pi})\, \rho(n(\boldsymbol{\pi})). \tag{4.3}$$

A path $\boldsymbol{\pi}$ *is called* ***accepting*** *or* ***successful*** *if* $\mathrm{w}(\boldsymbol{\pi}) \neq \mathbf{0}$.

The inner path weight is therefore the product of the weights of the transitions on the path, while the (full) path weight is the product of the transition weights as well as the initial and final weights of the origin and the destination of the path, respectively.

String Acceptance Weights and Weighted Regular Languages

When we introduced unweighted finite-state automata, we defined the important concept of recognizing a string and recognizing a language. We generalize these concepts to the very natural quantity of the weight assigned by a WFSA to a string $\boldsymbol{y} \in \Sigma^*$, i.e., its acceptance weight, or stringsum, as the sum of the weights of the paths that yield $\boldsymbol{y}$.

Definition 4.1.10 (Stringsum)**.** *The* ***stringsum****, string weight, or acceptance weight of a string* $\boldsymbol{y} \in \Sigma^*$ *under a WFSA* $\mathcal{A}$ *is defined as*

$$\mathcal{A}(\boldsymbol{y}) \stackrel{\text{def}}{=} \sum_{\boldsymbol{\pi} \in \Pi(\mathcal{A}, \boldsymbol{y})} \mathrm{w}(\boldsymbol{\pi}). \tag{4.4}$$

This naturally generalizes the notion of acceptance by an unweighted FSA—whereas an unweighted FSA only makes a binary decision of accepting or rejecting a string, a weighted FSA always accepts a string with a specific weight. This leads to the definition of the *weighted language* of the WFSA.

Definition 4.1.11. *Let* $\mathcal{A}$ *be a WFSA. Its* ***(weighted) language*** *is defined as*

$$L(\mathcal{A}) \stackrel{\text{def}}{=} \{(\boldsymbol{y}, \mathcal{A}(\boldsymbol{y})) \mid \boldsymbol{y} \in \Sigma^*\}. \tag{4.5}$$

We say a language is a weighted regular language if it is a language of some WFSA:

[7] In the case of WFSAs, a structure is a path. In the next section, we will see how to combine weights from basic units into *trees*.

Definition 4.1.12. *A weighted language L is a weighted regular language if there exists a WFSA $\mathcal{A}$ such that $L = L(\mathcal{A})$.*

Lastly, we also define the full and state-specific allsum of the automaton. The latter refers to the total weight assigned to *all* possible strings, or all possible paths whereas the former refer to the sum of the path weights of the paths stemming from a specific state.

Definition 4.1.13 (State-specific allsum)**.** *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA. The **allsum** of a state $q \in Q$ is defined as*

$$Z(\mathcal{A}, q) = \sum_{\substack{\boldsymbol{\pi} \in \Pi(\mathcal{A}) \\ q_1 = q}} \mathrm{w}_I(\boldsymbol{\pi}) \, \rho(n(\boldsymbol{\pi})). \tag{4.6}$$

State-specific allsums are also referred to as the **backward values** in the literature and are often denoted as $\boldsymbol{\beta}_q$.

Definition 4.1.14 (WFSA allsum)**.** *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA. The **allsum** of $\mathcal{A}$ is defined as*

$$Z(\mathcal{A}) = \sum_{\boldsymbol{y} \in \Sigma^*} \mathcal{A}(\boldsymbol{y}) = \sum_{\boldsymbol{y} \in \Sigma^*} \sum_{\boldsymbol{\pi} \in \Pi(\mathcal{A}, \boldsymbol{y})} \mathrm{w}(\boldsymbol{\pi}) = \sum_{\boldsymbol{\pi} \in \Pi(\mathcal{A})} \mathrm{w}(\boldsymbol{\pi}). \tag{4.7}$$

The second equality in Eq. (4.7) comes from the crucial observation that the double sum in the second term sums over precisely *all paths* of the automaton $\mathcal{A}$, which is where the name of the quantity comes from *allsum*.[8] This is easy to see if we consider that by summing over all possible strings, we enumerate all possible path yields, and each path in the automaton has a yield $\in \Sigma^*$. $Z(\mathcal{A})$ is again the result of summing over infinitely many terms (whether the set of strings in Σ^* of the infinitely many paths in a cyclic WFSA), and might therefore not necessarily be finite. For reasons which will become clear shortly, we will say that a WFSA $\mathcal{A}$ is **normalizable** if $Z(\mathcal{A}) < \infty$.

Note that the sum in Eq. (4.4) only contains one term if the automaton is deterministic. Whenever the automaton is non-deterministic, or when we are interested in the sum of paths with *different* yields as in Eq. (4.7), the interactions (namely, the *distributive law*) between the *sum* over the different *paths* and the *multiplications* over the *transitions* in the paths play an important role when designing efficient algorithms. Indeed, many algorithms defined for WFSAs rely on decompositions of such sums enabled by the distributive law.[9]

Accessibility and Probabilistic Weighted Finite-state Automata

An important property of states of a WFSA which we will need when investigating the tightness of finite-state language models is accessibility.

[8] Analogously, given some (implicitly defined) set of paths $\mathcal{S}$, we will name the sum over the weights of the paths in $\mathcal{S}$ the allsum over $\mathcal{S}$

[9] Many such examples are covered in the Advanced Formal Language Theory course.

Definition 4.1.15. *A state $q \in Q$ of a WFSA is **accessible** if there is a non-zero-weighted path to q from some state q_I with $\lambda(q_I) \neq 0$; it is **co-accessible state** if there is a non-zero-weighted path from q to some state q_F with $\rho(q_F) \neq 0$. It is **useful** if it is both accessible and co-accessible, i.e., q appears on some non-zero-weighted accepting path.*

Definition 4.1.16. ***Trimming** a WFSA means removing its useless states.*[10] *Removing the non-useful states means removing their rows and columns from $\mathbf{T}$ as well as their rows from $\overrightarrow{\lambda}$ and $\overrightarrow{\rho}$, yielding possibly smaller $\mathbf{T}'$, $\overrightarrow{\lambda}'$ and $\overrightarrow{\rho}'$.*

We will use WFSAs to specify language models. However, not every WFSA is a language model, i.e., a distribution over strings. Generally, the weight of a string could be negative if we allow arbitrary real weights. Thus, a restriction we will impose on *all* weighted automata that represent finite-state language models is that the weights be *non-negative*.

Furthermore, a special class of WFSAs that will be of particular interest later is probabilistic WFSAs.

Definition 4.1.17 (Probabilistic Weighted Finite-State Automaton)**.** *A WFSA $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ is **probabilistic** (a PFSA) if*

$$\sum_{q \in Q} \lambda(q) = 1 \tag{4.8}$$

and, for all $q \in Q$ and all outgoing transitions $q \xrightarrow{a/w} q' \in \delta$ it holds that

$$\lambda(q) \geqslant 0 \tag{4.9}$$

$$\rho(q) \geqslant 0 \tag{4.10}$$

$$w \geqslant 0 \tag{4.11}$$

and

$$\sum_{q \xrightarrow{a/w} q'} w + \rho(q) = 1. \tag{4.12}$$

This means that the initial weights of all the states of the automaton form a probability distribution (the initial weight of a state corresponds to the *probability* of starting in it), as well as that, for any state q in the WSFA, the weights of its outgoing transitions (with any label) together with its final weight form a valid discrete probability distribution. In a certain way, probabilistic finite-state automata naturally correspond to *locally normalized* language models, as we explore in the next subsection.

The eos symbol and the final weights. Notice that the final weights in a PFSA play an analogous role to the EOS symbol: the probability of ending a path in a specific state q—and therefore ending a string—is q's final weight!

[10] This does not affect the weights of the strings with $\mathrm{w}(\boldsymbol{y}) \neq 0$.

That is, the probability $\rho(q_F)$ for some $q_F \in Q$, representing the probability of ending the path in q_F, is analogous to the probability of ending a string $\boldsymbol{y}$, $p_{\text{SM}}(\text{EOS} \mid \boldsymbol{y})$, where q_F "represents" the string (history) $\boldsymbol{y}$.[11] When modeling language with weighted finite-state automata, we will therefore be able to avoid the need to specify the special symbol and rather rely on the final weights, which are naturally part of the framework.

4.1.2 Finite-state Language Models

We can now formally define what it means for a language model to be finite-state:

Definition 4.1.18 (Finite-state language models)**.** *A language model p_{LM} is **finite-state** if it can be represented by a weighted finite-state automaton, i.e., if there exists a WFSA $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ such that $L(\mathcal{A}) = L(p_{LM})$. Equivalently, we could say that p_{LM} is finite-state if its language is a weighted regular language.*

On the other hand, given a WFSA $\mathcal{A}$, there are two established ways of defining a *probability* of string.

String Probabilities in a Probabilistic Finite-state Automaton

In a probabilistic FSA (cf. Definition 4.1.17), any action from a state $q \in Q$ is associated with a probability. Since the current state completely encodes all the information of the input seen so far in a finite-state automaton, it is intuitive to see those probabilities as conditional probabilities of the next symbol given the input seen so far. One can, therefore, define the probability of a path as the product of these individual "conditional" probabilities.

Definition 4.1.19 (Path probability in a PFSA)**.** *We call the weight of a path $\boldsymbol{\pi} \in \Pi(\mathcal{A})$ in a probabilistic FSA the **probability** of the path $\boldsymbol{\pi}$.*

This alone is not enough to define the probability of any particular string $\boldsymbol{y} \in \Sigma^*$ since there might be multiple accepting paths for $\boldsymbol{y}$. Naturally, we define the probability of $\boldsymbol{y}$ as the sum of the individual paths that recognize it:

Definition 4.1.20 (String probability in a PFSA)**.** *We call the stringsum of a string $\boldsymbol{y} \in \Sigma^*$ in a probabilistic FSA the **probability** of the string $\boldsymbol{y}$:*

$$p_{\mathcal{A}}(\boldsymbol{y}) \stackrel{\text{def}}{=} \mathcal{A}(\boldsymbol{y}). \tag{4.13}$$

Crucially, notice that these two definitions did not require any *normalization* over all possible paths or strings. This closely resembles the way we defined locally normalized models based on the conditional probabilities of a sequence model. Again, such definitions of string probabilities are attractive as the summation over all possible strings is avoided. However, a careful reader might then ask themself: do these probabilities actually sum to 1, i.e., is a probabilistic FSA *tight*? As you might guess, they might *not*.[12] We explore this question in §4.1.4.

[11] Due to the possible non-determinism of WFSAs, the connection is of course not completely straightforward, but the point still stands.

[12] Notice that, however, whenever a PFSA is tight, its allsum is 1.

String Probabilities in a General Weighted Finite-state Automaton

To define string probabilities in a general weighted FSA, we use the introduced notions of the stringsum and the allsum. The allsum allows us to tractably *normalize* the stringsum to define the globally normalized probability of a string $\boldsymbol{y}$ as the *proportion* of the total weight assigned to all strings that is assigned to $\boldsymbol{y}$.[13]

Definition 4.1.21 (String probability in a WFSA)**.** *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a normalizable WFSA with non-negative weights. We define the probability of a string $\boldsymbol{y} \in \Sigma^*$ under $\mathcal{A}$ as*

$$p_{\mathcal{A}}(\boldsymbol{y}) \stackrel{\text{def}}{=} \frac{\mathcal{A}(\boldsymbol{y})}{Z(\mathcal{A})}. \tag{4.14}$$

Language Models Induced by a WFSA

With the notions of string probabilities in both probabilistic and general weighted FSAs, we can now define the language model induced by $\mathcal{A}$ as follows.

Definition 4.1.22 (A language model induced by a WFSA)**.** *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA. We define the **language model induced by** $\mathcal{A}$ as the following probability distribution over Σ^**

$$p_{LM\mathcal{A}}(\boldsymbol{y}) \stackrel{\text{def}}{=} p_{\mathcal{A}}(\boldsymbol{y}). \tag{4.15}$$

It is easy to see that while global normalization requires the computation of the allsum, language models induced by weighted FSAs through Eq. (4.14) are *globally normalized* and thus always tight. In the next subsection, we consider *how* the quantities needed for computing Eq. (4.14) can be computed. Of particular interest will be the quantity $Z(\mathcal{A})$, as it involves the summation over possibly infinitely many terms and therefore requires some clever tricks to be computed.

4.1.3 Normalizing Finite-state Language Models

In this subsection, we develop an algorithm for normalizing a globally normalized language model (cf. Definition 2.4.2) defined by a WFSA, i.e., an algorithm for computing the allsum $Z(\mathcal{A})$ whenever this quantity is finite. Moreover, the derivation will also reveal necessary and sufficient conditions for WFSAs to be normalizable.

Converting a matrix of pairwise pathsums to the allsum. Before we consider how to compute $Z(\mathcal{A})$, let us first consider a much simpler problem. Suppose we had a matrix $\mathbf{M}$, which contained at the entry $\mathbf{M}_{ij}$ the sum of all the inner weights over all paths between the states i and j, i.e.,

$$\mathbf{M}_{ij} = \sum_{\boldsymbol{\pi} \in \Pi(\mathcal{A}, i, j)} \mathrm{w}_{\mathrm{I}}(\boldsymbol{\pi}).$$

[13] We will see how the allsum can be computed tractably in §4.1.3.

How could we then compute the quantity $Z(\mathcal{A})$?

$$
\begin{aligned}
Z(\mathcal{A}) &= \sum_{\boldsymbol{\pi} \in \Pi(\mathcal{A})} \mathrm{w}(\boldsymbol{\pi}) && (4.16)\\
&= \sum_{\boldsymbol{\pi} \in \Pi(\mathcal{A})} \lambda(p(\boldsymbol{\pi}))\, \mathrm{w}_{\mathrm{I}}(\boldsymbol{\pi})\, \rho(n(\boldsymbol{\pi})) && (4.17)\\
&= \sum_{i,j \in Q} \sum_{\boldsymbol{\pi} \in \Pi(\mathcal{A},i,j)} \lambda(p(\boldsymbol{\pi}))\, \mathrm{w}_{\mathrm{I}}(\boldsymbol{\pi})\, \rho(n(\boldsymbol{\pi})) && (4.18)\\
&= \sum_{i,j \in Q} \sum_{\boldsymbol{\pi} \in \Pi(\mathcal{A},i,j)} \lambda(i)\, \mathrm{w}_{\mathrm{I}}(\boldsymbol{\pi})\, \rho(j) && (4.19)\\
&= \sum_{i,j \in Q} \lambda(i) \left(\sum_{\boldsymbol{\pi} \in \Pi(\mathcal{A},i,j)} \mathrm{w}_{\mathrm{I}}(\boldsymbol{\pi}) \right) \rho(j) && (4.20)\\
&= \sum_{i,j \in Q} \lambda(i)\, \mathbf{M}_{ij} \rho(j) && (4.21)\\
&= \overrightarrow{\lambda} \mathbf{M} \overrightarrow{\rho}, && (4.22)
\end{aligned}
$$

where $\overrightarrow{\lambda}$ and $\overrightarrow{\rho}$ denote the vectors resulting from the "vectorization" of the functions λ and ρ, i.e., $\overrightarrow{\lambda}_n = \lambda(n)$ and $\overrightarrow{\rho}_n = \rho(n)$. This also explains the naming of the functions λ and ρ: the initial weights function λ, "lambda" appears on the left side of the closed form expression for $Z(\mathcal{A})$ and the definition of the path weight (cf. Eq. (4.3)), whereas the final weights function ρ, rho, appears on the right side of the expression and the definition of the path weight.

Computing the matrix of pairwise pathsums. Let $\mathbf{T}$ be the transition matrix of the automaton $\mathcal{A}$. Notice that the entry $\mathbf{T}_{ij}$ by definition contains the sum of the inner weights of all paths of length exactly 1 (individual transitions) between the states i and j. We also define $\mathbf{T}^0 = \mathbf{I}$, meaning that the sum of the weights of the paths between i and j of length zero is 0 if $i \neq i$ and 1, which is multiplicative unity. This corresponds to not transitioning if $i = j$. We next state a basic result from graph theory.

Lemma 4.1.1. *Let* $\mathbf{T}$ *be the transition matrix of some weighted directed graph* $\mathcal{G}$*. Then the matrix* $\mathbf{T}^d$ *contains the allsum of all paths of length* exactly d*, i.e.,*

$$
\mathbf{T}^d_{i,j} = \sum_{\substack{\boldsymbol{\pi} \in \Pi(\mathcal{A},i,j) \\ |\boldsymbol{\pi}| = d}} \mathrm{w}_I(\boldsymbol{\pi})\,. \tag{4.23}
$$

Proof. By induction on the path length. Left as an exercise for the reader. ■

It follows directly that the matrix

$$
\mathbf{T}^{\leqslant d} \stackrel{\text{def}}{=} \sum_{k=1}^{d} \mathbf{T}^k
$$

contains the pairwise pathsums of paths of length *at most* d.

In general, the WFSA representing a n-gram language model can of course be cyclic. This means that the number of paths in $\Pi(\mathcal{A})$ might be infinite and they might be of arbitrary length (which is the result of looping in a cycle arbitrarily many times). To compute the pairwise pathsums over *all* possible paths, we, therefore, have to compute

$$\mathbf{T}^* \stackrel{\text{def}}{=} \lim_{d\to\infty} \mathbf{T}^{\leqslant d} = \sum_{d=0}^{\infty} \mathbf{T}^d. \tag{4.24}$$

This is exactly the matrix form of the *geometric sum*. Similarly to the scalar version, we can manipulate the expression Eq. (4.24) to arrive to a closed-form expression for computing it:

$$\begin{aligned}
\mathbf{T}^* &= \sum_{d=0}^{\infty} \mathbf{T}^d && (4.25)\\
&= \mathbf{I} + \sum_{d=1}^{\infty} \mathbf{T}^d && (4.26)\\
&= \mathbf{I} + \sum_{d=1}^{\infty} \mathbf{T}\mathbf{T}^{d-1} && (4.27)\\
&= \mathbf{I} + \mathbf{T} \sum_{d=1}^{\infty} \mathbf{T}^{d-1} && (4.28)\\
&= \mathbf{I} + \mathbf{T} \sum_{d=0}^{\infty} \mathbf{T}^{d} && (4.29)\\
&= \mathbf{I} + \mathbf{T}\mathbf{T}^*. && (4.30)
\end{aligned}$$

If the inverse of $(\mathbf{I} - \mathbf{T})$ exists, we can further rearrange this equation to arrive at

$$\begin{aligned}
\mathbf{T}^* &= \mathbf{I} + \mathbf{T}\mathbf{T}^* && (4.31)\\
\mathbf{T}^* - \mathbf{T}\mathbf{T}^* &= \mathbf{I} && (4.32)\\
\mathbf{T}^* - \mathbf{T}^*\mathbf{T} &= \mathbf{I} && (4.33)\\
\mathbf{T}^* (\mathbf{I} - \mathbf{T}) &= \mathbf{I} && (4.34)\\
\mathbf{T}^* &= (\mathbf{I} - \mathbf{T})^{-1}. && (4.35)
\end{aligned}$$

This means that, if $(\mathbf{I} - \mathbf{T})$ exists, we can compute the pairwise pathsums by simply inverting it! Using the remark above on how to convert a matrix of pairwise pathsums into the full allsum, we can therefore see that we can globally normalize an n-gram language model by computing a matrix inversion! Since the runtime of inverting a $N \times N$ matrix is $\mathcal{O}\left(N^3\right)$, and $N = |Q|$ for a transition matrix of a WFSA with states Q, we can globally normalize a n-gram language model in time cubic in the number of its states. This is a special case of the

general algorithm by Lehmann (1977). Note, however, that this might still be prohibitively expensive: as we saw, the number of states in a n-gram model grows exponentially with n, and even small n's and reasonable alphabet sizes might result in a non-tractable number of states in the WFSA with the cubic runtime.

We still have to determine when the infinite sum in Eq. (4.24) converges. One can see by writing out the product $\mathbf{T}^d$ in terms of its eigenvalues that the entries of $\mathbf{T}^d$ diverge towards $\pm\infty$ as soon as the magnitude of any of $\mathbf{T}$'s eigenvalues is larger than 1. This means that $\|\mathbf{T}\|_2 < 1$ (spectral norm) is a necessary condition for the infinite sum to exist. This is, however, also a *sufficient* condition: if $\|\mathbf{T}\|_2 < 1$, all of $\mathbf{T}$'s eigenvalues are smaller than 1 in magnitude, meaning that the eigenvalues of $\mathbf{I} - \mathbf{T}$ are strictly positive and the matrix $\mathbf{I} - \mathbf{T}$ is invertible.[14]

Speed-ups of the Allsum Algorithm

The introduced algorithm for computing the allsum in a WFSA can, therefore, be implemented as a matrix inverse. This means that its runtime is $\mathcal{O}\left(|Q|^3\right)$, which can be relatively expensive. Fortunately, faster algorithms exist for WFSAs with more structure (in their transition functions)—for example, the allsum can be computed in time *linear* in the number of transitions if the automaton is acyclic using a variant of the Viterbi algorithm (Eisner, 2016). Furthermore, if the automaton "decomposes" into many smaller strongly connected components (i.e., subgraphs that are cyclic), but the components are connected sparsely and form an acyclic graph of components, the allsum can also be computed more efficiently using a combination of the algorithms described above and the algorithm for acyclic WFSA, resulting in a possibly large speedup over the original algorithm.

Importantly, the allsum algorithm and all the speed-ups are *differentiable*, meaning that they can be used during the *gradient-based training* (cf. §3.2.3) of a finite-state language model, where the weights are parametrized using some learnable parameters—we will return to this point shortly.

Locally Normalizing a Globally Normalized Finite-state Language Model

As shown in Theorem 2.4.2, any language model (and thus, any globally-normalized model with a normalizable energy function) can also be locally normalized. In the case of finite-state language models, we can actually explicitly construct the WFSA representing the locally normalized variant using a procedure that is conceptually similar to the allsum algorithm described here. In contrast to the procedure we presented here, however, the local normalization algorithm computes the pathsums of the paths stemming from every possible state q *individually* and then "reweights" the *transitions* depending on the pathsums of their target states r. You can think of this as computing the contributions to the entire allsum from q made by all the individual outgoing transitions from q and then normalizing those contributions. This is an instance of the more

[14] $1 - \lambda$ is an eigenvalues of $\mathbf{I} - \mathbf{T}$ iff λ is an eigenvalue of $\mathbf{T}$.

general **weight pushing** algorithm.[15] This can be summarized by the following theorem:

Theorem 4.1.1 (PFSAs and WFSAs are equally expressive)**.** *Normalizable weighted finite-state automata with non-negative weights and tight probabilistic finite-state automata are equally expressive.*

In the proof of this theorem, we will make use of the following lemma.

Lemma 4.1.2. *Let* $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ *and* $q \in Q$*. Then*

$$Z(\mathcal{A}, q) = \sum_{q \xrightarrow{a/w} q' \in \delta_{\mathcal{A}_L}} \mathcal{W}\left(q \xrightarrow{a/\cdot} q'\right) Z(\mathcal{A}, q') + \rho(q) \tag{4.36}$$

Proof. You are asked to show this in Exercise 4.1. ■

We can now prove Theorem 4.1.1

Proof. To prove the theorem, we have to show that any WFSA can be written as a PFSA and vice versa.[16]

⇐ Since any tight probabilistic FSA is simply a WFSA with $Z(\mathcal{A}) = 1$, this holds trivially.

⇒ Local normalization is a general property of automata resulting from weight pushing. Here, we describe the construction in the special case of working with real-valued weights. See Mohri et al. (2008) for a general treatment.

Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a normalizable WFSA with non-negative weights. We now show that, for any WFSA, there exists a PFSA encoding the same language model. Let $\mathcal{A}_G = (\Sigma, Q, \delta, \lambda, \rho)$ be a trim WFSA that encodes a distribution over Σ^* using Eq. (4.14). We now construct a tight probabilistic finite-state automaton $\mathcal{A}_L = (\Sigma, Q, \delta_{\mathcal{A}_L}, \lambda_{\mathcal{A}_L}, \rho_{\mathcal{A}_L})$ whose language is identical. We define the initial and final weights of the probabilistic FSA as follows.

$$\lambda_{\mathcal{A}_L}(q) \stackrel{\text{def}}{=} \lambda(q) \frac{Z(\mathcal{A}, q)}{Z(\mathcal{A})} \tag{4.37}$$

$$\rho_{\mathcal{A}_L}(q) \stackrel{\text{def}}{=} \frac{\rho(q)}{Z(\mathcal{A}, q)} \tag{4.38}$$

We define the transitions of the probabilistic FSA as follows.

$$\mathcal{W}_{\mathcal{A}_L}\left(q \xrightarrow{a/\cdot} q'\right) \stackrel{\text{def}}{=} \frac{\mathcal{W}\left(q \xrightarrow{a/\cdot} q'\right) Z(\mathcal{A}, q')}{Z(\mathcal{A}, q)} \tag{4.39}$$

This means that $\mathcal{A}_L$ contains the same transitions as $\mathcal{A}$, they are simply reweighted. Note that the assumption that $\mathcal{A}$ is trimmed means that all the quantities in the denominators are non-zero.

[15] See Mohri et al. (2008) for a more thorough discussion of weight pushing.

[16] By "written as", we mean that the weighted language is the same.

It is easy to see that the weights defined this way are non-negative due to the non-negativity of $\mathcal{A}$'s weights. Furthermore, the weights of all outgoing arcs from any $q \in Q$ and its final weight sum to 1:

$$\sum_{q \xrightarrow{a/w} q' \in \delta_{\mathcal{A}_L}} w + \rho_{\mathcal{A}_L}(q) \tag{4.40}$$

$$= \sum_{q \xrightarrow{a/w} q' \in \delta_{\mathcal{A}_L}} \frac{\mathcal{W}\left(q \xrightarrow{a/\cdot} q'\right) Z(\mathcal{A}, q')}{Z(\mathcal{A}, q)} + \frac{\rho(q)}{Z(\mathcal{A}, q)} \qquad \text{(definition of } \delta_{\mathcal{A}_L}\text{)} \tag{4.41}$$

$$= \frac{1}{Z(\mathcal{A}, q)} \left(\sum_{q \xrightarrow{a/w} q' \in \delta_{\mathcal{A}_L}} \mathcal{W}\left(q \xrightarrow{a/\cdot} q'\right) Z(\mathcal{A}, q') + \rho(q) \right) \tag{4.42}$$

$$= 1 \qquad \text{(Lemma 4.1.2)} \tag{4.43}$$

It is also easy to see that the initial weights form a probability distribution over the states of the constructed automaton.

$$\sum_{q \in Q} \lambda_{\mathcal{A}_L}(q) = \sum_{q \in Q} \lambda(q) \frac{Z(\mathcal{A}, q)}{Z(\mathcal{A})} \tag{4.44}$$

$$= \frac{1}{Z(\mathcal{A})} \sum_{q \in Q} \lambda(q) Z(\mathcal{A}, q) \tag{4.45}$$

$$= \frac{1}{Z(\mathcal{A})} Z(\mathcal{A}) = 1 \tag{4.46}$$

We now have to show that the probabilities assigned by these two automata match. We will do that by showing that the probabilities assigned to individual *paths* match, implying that stringsums match as well. The probability of a path is defined analogously to a probability of a string, i.e., $p_{\mathcal{A}}(\boldsymbol{\pi}) = \frac{\mathrm{w}(\boldsymbol{\pi})}{Z(\mathcal{A})}$ (where $Z(\mathcal{A}) = 1$ for tight probabilistic FSAs). Let then $\boldsymbol{\pi} = \left(q_1 \xrightarrow{a_1/w_1} q_2, \ldots, q_{N-1} \xrightarrow{a_{N-1}/w_{N-1}} q_N\right) \in \Pi(\mathcal{A}) = \Pi(\mathcal{A}_L)$. Then, by the definitions of $\mathcal{W}_{\mathcal{A}_L}$, $\lambda_{\mathcal{A}_L}$, and $\rho_{\mathcal{A}_L}$

$$p_{\mathcal{A}_L}(\boldsymbol{\pi}) = \lambda_{\mathcal{A}_L}(q_1) \left(\prod_{n=1}^{N-1} w_n \right) \rho_{\mathcal{A}_L}(q_N) \tag{4.47}$$

$$= \lambda(q_1) \frac{Z(\mathcal{A}, q_1)}{Z(\mathcal{A})} \prod_{n=1}^{N-1} \frac{\mathcal{W}\left(q_n \xrightarrow{a/\cdot} q_{n+1}\right) Z(\mathcal{A}, q_{n+1})}{Z(\mathcal{A}, q_n)} \frac{\rho(q_N)}{Z(\mathcal{A}, q_N)}. \tag{4.48}$$

Notice that the state-specific allsums of all the inner states of the path (all states apart from q_1 and q_N) *cancel out* as the product moves over the transitions of the path. Additionally, the terms $Z(\mathcal{A}, q_1)$ and $Z(\mathcal{A}, q_N)$ cancel out with the definitions of $\lambda_{\mathcal{A}_L}$ and $\rho_{\mathcal{A}_L}$. This leaves us with

$$p_{\mathcal{A}_L}(\boldsymbol{\pi}) = \lambda(q_1) \frac{1}{Z(\mathcal{A})} \prod_{n=1}^{N-1} \mathcal{W}\left(q_n \xrightarrow{a/\cdot} q_{n+1}\right) \rho(q_N) = p_{\mathcal{A}}(\boldsymbol{\pi}), \tag{4.49}$$

finishing the proof. ■

While Theorem 2.4.2 shows that any language model can be locally normalized, Theorem 4.1.1 shows that in the context of finite-state language models, the locally normalized version of a globally-normalized model is *also* a finite-state model.

Defining a Parametrized Globally Normalized Language Model

Having learned how an arbitrary normalizable finite-state language model can be normalized, we now discuss how models in this framework can be *parametrized* to enable fitting them to some training data. Crucial for parameterizing a globally normalized model is a **score function** $f^{\delta}_{\boldsymbol{\theta}} : Q \times \Sigma \times Q \to \mathbb{R}$, which parametrizes the transitions between the states and thus determines the weights of the (accepting) paths. Additionally, we also parameterized the initial and final functions $f^{\lambda}_{\boldsymbol{\theta}}$ and $f^{\rho}_{\boldsymbol{\theta}}$. These parametrized functions then define the automaton $\mathcal{A}_{\boldsymbol{\theta}} \stackrel{\text{def}}{=} (\Sigma, Q, \delta_{\boldsymbol{\theta}}, \lambda_{\boldsymbol{\theta}}, \rho_{\boldsymbol{\theta}})$, where $\delta_{\boldsymbol{\theta}} \stackrel{\text{def}}{=} \left\{q_1 \xrightarrow{y/f^{\delta}_{\boldsymbol{\theta}}(q_1,y,q_2)} q_2\right\}$, $\lambda_{\boldsymbol{\theta}}(q_I) \stackrel{\text{def}}{=} f^{\lambda}_{\boldsymbol{\theta}}(q_I)$, and $\rho_{\boldsymbol{\theta}}(q_F) \stackrel{\text{def}}{=} f^{\rho}_{\boldsymbol{\theta}}(q_F)$. Note that we can parametrize the function $f_{\boldsymbol{\theta}}$ in any way we want; for example, the function could be a neural network using distributed representations (we will see a similar example at the end of this section), or it could simply be a lookup table of weights. The fact that the function $f_{\boldsymbol{\theta}} : (q_1, y, q_2)$ can only "look at" the identities of the states and the symbol might seem limiting; however, the states alone can encode a lot of information: for example, in n-gram models we describe below, they will encode the information about the previous $n-1$ symbols and the transitions will then encode the probabilities of transitioning between such sequences of symbols.

The globally parametrized model then simply takes in any string $\boldsymbol{y} \in \Sigma^*$ and computes its stringsum value under the parametrized automaton, which in turn, as per Eq. (4.15), defines probabilities of the strings. The quantity $Z(\mathcal{A}_{\boldsymbol{\theta}})$ can be computed with the allsum algorithm discussed in §4.1.3. Importantly, since the algorithms for computing the string probabilities are differentiable, the model defined this way can also be *trained* with gradient-based learning as described in §3.2.3.

You might notice that this formulation does not exactly match the formulation of globally normalized models from Definition 2.4.2—the function $\mathcal{A} : \Sigma^* \to \mathbb{R}$ does not exactly match the form of an energy function as its values are not exponentiated as in Eq. (2.11). However, we tie this back to the definition of

globally normalized models by defining an actual energy function as a simple *transformation* of the stringsum given by $\mathcal{A}_{\boldsymbol{\theta}}$. We can define the globally normalizing energy function $\widehat{p}_{\text{GN}\mathcal{A}_{\boldsymbol{\theta}}}$ as

$$\widehat{p}_{\text{GN}\mathcal{A}_{\boldsymbol{\theta}}}(\boldsymbol{y}) \stackrel{\text{def}}{=} -\log\left(\mathcal{A}(\boldsymbol{y})\right), \tag{4.50}$$

which can be easily seen to, after exponentiating it as in Eq. (2.11), result in the same expression as Eq. (4.15). With this, we have formulated finite-state language models as general globally normalized models.

Having introduced WFSAs as a formal and abstract computational model which can define a set of weighted strings, we now show how it can be used to explicitly model a particularly simple family of languages. We arrive at this family of language models when we impose a specific assumption on the set of *conditional distributions* of the language models that ensures that they are finite-state: the n-gram assumption.

4.1.4 Tightness of Finite-state Models

Any normalizable globally normalized finite-state language model is tight by definition because the sum of the scores over all *finite* strings is finite, and since they are normalized, they sum to 1. We, therefore, focus on *locally* normalized finite-state models and provide necessary and sufficient conditions for their tightness. Locally normalized finite-state models are exactly probabilistic WFSAs (Definition 4.1.17). Luckily, the tightness of probabilistic WFSAs can be easily characterized, as the following theorem shows.

Theorem 4.1.2. *A probabilistic WFSA is tight if and only if all accessible states are also co-accessible.*

Proof. We prove each direction in turn.

(⇒): Assume the WFSA is tight. Let $q \in Q$ be an accessible state, which means q can be reached after a finite number of steps with positive probability. By tightness assumption, then there must be a positive probability path from q to termination, or else the WFSA will not be able to terminate after reaching q, resulting in non-tightness. This means that q is also co-accessible. So, assuming that the WFSA is tight, every accessible state is also co-accessible.

(⇐): Assume that all accessible states are co-accessible. First, one may consider a Markov chain consisting only of the set of accessible states $Q_A \subseteq Q$, since all other states will have probability 0 at every step. Recall a fundamental result in finite-state Markov chain theory which states that, if there exists a unique absorbing state which is reachable from every state, then the Markov process is absorbed by this state with probability 1 (see, e.g., Theorem 11.3 in Grinstead and Snell, 1997). We already have that

- EOS is an absorbing state, and that

- by assumption, every state in Q_A is co-accessible which implies that they can reach EOS.

Hence, it remains to show that EOS is the *unique* absorbing state. Suppose there is another state (or group of states) in Q_A distinct from EOS that is absorbing, i.e., cannot leave once entered. Then, these states cannot reach EOS by assumption, which means they are not co-accessible, contradicting the assumption that every state in Q_A is co-accessible. Hence, EOS is the only absorbing state in Q_A and by the property of an absorbing Markov chain, the process is absorbed by EOS with probability 1. In other words, the WFSA is tight. ■

Notice that trimming a PFSA results in a model that satisfies $\rho(q) + \sum_{q \xrightarrow{a/w} q'} w \leqslant 1$, but might no longer achieve equality as required by Definition 4.1.17. We call such models substochastic WFSAs.

Definition 4.1.23 (Substochastic Weighted Finite-State Automaton)**.** *A WFSA $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ is **substochastic** if for all $q \in Q$ and all outgoing transitions $q \xrightarrow{a/w} q' \in \delta$ it holds that*

$$\lambda(q) \geqslant 0 \tag{4.51}$$

$$\rho(q) \geqslant 0 \tag{4.52}$$

$$w \geqslant 0 \tag{4.53}$$

and

$$\rho(q) + \sum_{q \xrightarrow{a/w} q'} w \leqslant 1. \tag{4.54}$$

We can then express the termination probability of a WFSA in simple linear algebra terms.

Theorem 4.1.3. *Let $\mathbf{T}'$ be the transition sum matrix of a trimmed substochastic WFSA. Then $\mathbf{I} - \mathbf{T}'$ is invertible and $p(\mathbf{x} \in \Sigma^*) = \overrightarrow{\lambda}'^{\top}(\mathbf{I} - \mathbf{T}')^{-1}\overrightarrow{\rho}' \leqslant 1$.*

In the following, we will make use of the spectral radius of a matrix.

Definition 4.1.24 (Spectral radius)**.** *The **spectral radius** of a matrix $\mathbf{M} \in \mathbb{C}^{N \times N}$ with eigenvalues $\lambda_1, \ldots, \lambda_N$ is defined as*

$$\rho_s(\mathbf{M}) \stackrel{\text{def}}{=} \max\{|\lambda_1|, \ldots, |\lambda_N|\}. \tag{4.55}$$

To prove Theorem 4.1.3, we will make use of the following useful lemma.

Lemma 4.1.3. *Let $\mathbf{T}'$ be the transition sum matrix of a trimmed substochastic WFSA, then $\rho_s(\mathbf{T}') < 1$.*

Proof. To begin with, we wish to apply the following result which connects the row sums of a matrix to its spectral radius. Below, $\mathcal{M}_N$ denotes the set of $N \times N$ matrices, and $\|\mathbf{A}\|_\infty = \max_{1 \leqslant n \leqslant N} \sum_{i=1}^{N} |\mathbf{A}_{ni}|$ denotes the infinity matrix norm.

Proposition 4.1.1 (§6.2.P8; Horn and Johnson, 2012)**.** *For any* $\mathbf{A} \in \mathcal{M}_N$, $\rho_s(\mathbf{A}) \leqslant \|\mathbf{A}\|_\infty$. *Additionally, if* $\mathbf{A}$ *is irreducible and not all absolute row sums of* $\mathbf{A}$ *are equal, then* $\rho_s(\mathbf{A}) < \|\mathbf{A}\|_\infty$.

However, the transition sum matrix $\mathbf{P}$ of a substochastic WFSA may be reducible whereas the irreducibility condition in Proposition 4.1.1 cannot be dropped. Hence, we need to "decompose" $\mathbf{T}'$ in a way to recover irreducibility. We use the *Frobenius normal form* (also known as *irreducible normal form*) to achieve this.

Proposition 4.1.2 (§8.3.P8; Horn and Johnson, 2012)**.** *Let* $\mathbf{A} \in \mathcal{M}_N$ *be non-negative. Then, either* $\mathbf{A}$ *is irreducible or there exists a permutation matrix* $\mathbf{P}$ *such that*

$$\mathbf{P}^\top \mathbf{A} \mathbf{P} = \begin{bmatrix} \mathbf{A}_1 & & * \\ & \ddots & \\ \mathbf{0} & & \mathbf{A}_K \end{bmatrix} \tag{4.56}$$

is block upper triangular, and each diagonal block is irreducible (possibly a 1-by-1 zero matrix). This is called an ***Frobenius normal form*** *(or* ***irreducible normal form****) of* A. *Additionally,* $\Lambda(A) = \Lambda(\mathbf{A}_1) \cup \cdots \cup \Lambda(\mathbf{A}_K)$ *where* $\Lambda(\cdot)$ *denotes the set of eigenvalues of a matrix.*

Notice that, by way of a similarity transformation via a permutation matrix, the Frobenius normal form is equivalent to a relabeling of the states in the trimmed WFSA in the sense of

$$\begin{aligned} (\mathbf{P}^\top \overrightarrow{\lambda}')^\top (\mathbf{P}^\top \mathbf{T}' \mathbf{P})^K (\mathbf{P}^\top \overrightarrow{\rho}') &= (\overrightarrow{\lambda}'^\top \mathbf{P})(\mathbf{P}^\top \mathbf{T}'^K \mathbf{P})(\mathbf{P}^\top \overrightarrow{\rho}') && \text{(4.57a)} \\ &= \overrightarrow{\lambda}'^\top \mathbf{T}'^K \overrightarrow{\rho}' && \text{(4.57b)} \end{aligned}$$

where the equalities follow from the fact that the inverse of a permutation matrix $\mathbf{P}$ is its transpose. Hence, with an appropriate relabeling, we may assume without loss of generality that $\mathbf{P}$ is already put into a Frobenius normal form

$$\mathbf{T}' = \begin{bmatrix} \mathbf{T}'_1 & & * \\ & \ddots & \\ \mathbf{0} & & \mathbf{T}'_K \end{bmatrix} \tag{4.58}$$

where each $\mathbf{T}'_k$ is irreducible.

Since the transition sum matrix $\mathbf{T}'$ of a trimmed substochastic WFSA is a substochastic matrix, each $\mathbf{T}'_k$ is also substochastic. In fact, each $\mathbf{T}'_k$ is *strictly* substochastic, meaning that there is at least a row that sums to less than 1. To see this, suppose to the contrary that there is a probabilistic $\mathbf{T}'_k$. Since the WFSA is trimmed, every state is both accessible and co-accessible. Being accessible implies that there is a positive probability of reaching every state in $\mathbf{T}'_k$. However, the probabilisticity of $\mathbf{T}'_k$ forces the corresponding $\overrightarrow{\rho}'$ entries to be 0. Hence, none of these states can transition to EOS, meaning that they're

not co-accessible, contradicting the assumption. Hence, every $\mathbf{T}'_k$ is strictly substochastic and has at least one strictly less than 1 row sum. Then, either all row sums of $\mathbf{T}'_k$ are less than 1 or some row sums are 1 and some are less than 1. In either cases, Proposition 4.1.1 implies that $\rho_{\text{s}}(\mathbf{T}'_k) < 1$ for all $1 \leqslant k \leqslant K$. Finally, as Proposition 4.1.2 entails, $\rho_{\text{s}}(\mathbf{T}') = \max\{\rho_{\text{s}}(\mathbf{T}'_1), \ldots, \rho_{\text{s}}(\mathbf{T}'_K)\}$ where each $\rho_{\text{s}}(\mathbf{T}'_k) < 1$. Hence, $\rho_{\text{s}}(\mathbf{T}') < 1$. ■

We now use the stated results to finally prove Theorem 4.1.3.

Proof. By Lemma 4.1.3, $\rho_{\text{s}}(\mathbf{T}') < 1$, in which case $\mathbf{I} - \mathbf{T}'$ is invertible and the Neumann series $\mathbf{I} + \mathbf{T}' + \mathbf{T}'^2 + \cdots$ converges to $(\mathbf{I} - \mathbf{T}')^{-1}$ (§5.6, Horn and Johnson, 2012). Hence, we can write $(\mathbf{I} - \mathbf{T}')^{-1} = \sum_{k=0}^{\infty} \mathbf{T}'^k$. Then,

$$p(\Sigma^*) = \sum_{k=0}^{\infty} P(\Sigma^k) \tag{4.59a}$$

$$= \sum_{k=0}^{\infty} \overrightarrow{\lambda}'^{\top} \mathbf{T}'^k \overrightarrow{\rho}' \tag{4.59b}$$

$$= \overrightarrow{\lambda}'^{\top} \left(\sum_{k=0}^{\infty} \mathbf{T}'^k \right) \overrightarrow{\rho}' \tag{4.59c}$$

$$= \overrightarrow{\lambda}'^{\top} (\mathbf{I} - \mathbf{T}')^{-1} \overrightarrow{\rho}'. \tag{4.59d}$$

■

4.1.5 The *n*-gram Assumption and Subregularity

We now turn our attention to one of the first historically significant language modeling frameworks: n-gram models. While they are often taught completely separately from (weighted) finite-state automata, we will see shortly that they are simply a special case of finite-state language models and thus all results for the more general finite-state language models also apply to the specific n-gram models as well.

As we saw in Theorem 2.4.2, we can factorize the language model p_{LM} for $\boldsymbol{y} = y_1 \ldots y_T \in \Sigma^*$ as

$$p_{\text{LM}}(\boldsymbol{y}) = p_{\text{LN}}(\boldsymbol{y}) = p_{\text{SM}}(\text{EOS} \mid \boldsymbol{y}) \prod_{t=1}^{T} p_{\text{SM}}(y_t \mid \boldsymbol{y}_{<t}), \tag{4.60}$$

where $p_{\text{SM}}(y \mid \boldsymbol{y})$ are specified by a locally normalized model (Definition 2.4.5).

Recall that SMs specify individual conditional distributions of the next symbol y_t given the previous $t-1$ symbols for *all possible* t. However, as t grows and the history of seen tokens accumulates, the space of possible histories (sequences of strings to condition on) grows very large (and indeed infinite as $t \to \infty$). This makes the task of modeling individual conditional distributions for large t computationally infeasible. One way to make the task more manageable is by using the n-gram assumption.

Assumption 4.1.1 (n-gram assumption)**.** *In words, the* ***n-gram assumption*** *states that the probability of a word y_t only depends on $n-1$ previous words $y_{t-1}, \ldots, y_{t-n+1}$ where $y_0 \stackrel{\text{def}}{=} \text{BOS}$. Notationally, we can write the n-gram assumption as a conditional independence assumption i.e.,*

$$p_{SM}(y_t \mid \boldsymbol{y}_{<t}) \stackrel{\text{def}}{=} p_{SM}(y_t \mid y_{t-1} \cdots y_{t-n+1}) \stackrel{\text{def}}{=} p_{SM}(y_t \mid \boldsymbol{y}_{t-n-1:t-1}) \tag{4.61}$$

The sequence $y_{t-1} \cdots y_{t-n+1}$) is often called the ***history*** *or the* ***context****.*

In plain English, this means that the probability of a token only depends on the previous $n-1$ tokens. n-gram assumption is, therefore, an alias of $(n-1)^{\text{th}}$ order Markov assumption in the language modeling context.

Handling edge cases by padding. Given our definition in Eq. (4.61) where the conditional probability $p_{\text{SM}}(y_t \mid \boldsymbol{y}_{t-n-1:t-1})$ depends on exactly $n-1$ previous symbols, we could run into an issue with negative indices for $t < n$. To handle edge cases for $t < n$, we will **pad** the sequences with the BOS symbols at the beginning, that is, we will assume that the sequences $\overline{y}_1 \ldots \overline{y}_t$ for $t < n-1$ are "transformed" as

$$\overline{y}_1 \overline{y}_2 \ldots \overline{y}_t \mapsto \underbrace{\text{BOS} \ldots \text{BOS}}_{n-1-t \text{ times}} \overline{y}_1 \overline{y}_2 \ldots \overline{y}_t \tag{4.62}$$

Notice that with such a transformation, we always end up with strings of length $n-1$, which is exactly what we need for conditioning in an n-gram model. In the following, we will assume that all such sequences are already transformed, but at the same time, we will assume that

$$p_{\text{SM}}\left(\overline{y} \mid \underbrace{\text{BOS} \ldots \text{BOS}}_{n-1-t \text{ times}} \overline{y}_1 \overline{y}_2 \ldots \overline{y}_t\right) = \overline{y}_0 \overline{y}_1 \overline{y}_2 \ldots \overline{y}_t \tag{4.63}$$

By definition, n-gram language models can only model dependencies spanning n tokens or less. By limiting the length of the relevant context when determining $p_{\text{SM}}(y_t \mid \boldsymbol{y}_{<t})$ to the previous n tokens, the n-gram assumption limits the number of possible probability *distributions* that need to be tracked to $\mathcal{O}\left(|\Sigma|^{n-1}\right)$.

A particularly simple case of the n-gram model is the **bigram** model where $n = 2$, which means that the probability of the next word only depends on the previous one, i.e., $p_{\text{SM}}(y_t \mid \boldsymbol{y}_{<t}) = p_{\text{SM}}(y_t \mid y_{t-1})$.[17]

Example 4.1.5 (A simple bigram model)**.** *Let us look at a specific example of a simple bigram model. Suppose our vocabulary consists of the words "large", "language", and "models", thus, $|\Sigma| = 3$. To specify the bigram model, we have to define the conditional probabilities $p_M(y_j \mid y_i)$ for $y_i \in \Sigma \cup \{\text{BOS}, \text{EOS}\}$ and $y_j \in \Sigma \cup \{\text{EOS}\}$ (remember that we do not have to model the probability of the next token being* BOS*). In the case of bigrams, we can represent those in a table, where the entry at position i, j represents the probability $p_M(y_j \mid y_i)$:*

[17] What would the uni-gram ($n = 1$) model look like? What conditional dependencies between words in a sentence could be captured by it?

	"large"	*"language"*	*"models"*	"EOS"
BOS	0.4	0.2	0.2	0.2
"large"	0.1	0.4	0.2	0.3
"language"	0.1	0.1	0.4	0.4
"models"	0.2	0.2	0.1	0.5
"EOS"	0.4	0.2	0.2	0.2

Under our model, the probability of the sentence "large language models" would be

$$\begin{aligned} & p_{SM}(\textit{"large"} \mid \text{BOS}) \\ & \cdot p_{SM}(\textit{"language"} \mid \textit{"large"}) \\ & \cdot p_{SM}(\textit{"models"} \mid \textit{"language"}) \\ & \cdot p_{SM}(\text{EOS} \mid \textit{"models"}) \\ & = 0.4 \cdot 0.4 \cdot 0.4 \cdot 0.5 = 0.032 \end{aligned}$$

while the probability of the sentence "large large large" would be

$$\begin{aligned} & p_{SM}(\textit{"large"} \mid \text{BOS}) \\ & \cdot p_{SM}(\textit{"large"} \mid \textit{"large"}) \\ & \cdot p_{SM}(\textit{"large"} \mid \textit{"large"}) \\ & \cdot p_{SM}(\text{EOS} \mid \textit{"large"}) \\ & = 0.4 \cdot 0.1 \cdot 0.1 \cdot 0.3 = 0.0012. \end{aligned}$$

Note that the probabilities in the above table are made up and not completely reasonable. A real n-gram model would not allow for probabilities of exactly 0 *to avoid pathological behavior.*

Representing *n*-gram Models as WFSAs

We define n-gram language models as models which only consider a finite amount of context when defining the conditional probabilities of the next token. This means that the set of possible conditional distributions $p_{\text{SM}}(y \mid \boldsymbol{y})$ is also finite which very naturally connects them to weighted finite-state automata—indeed, every n-gram language model is a WFSA—specifically, a probabilistic finite-state automaton (or a substochastic one). We will make this connection more formal in this subsection, thus formally showing that n-gram models are indeed finite-state. Note that this is different from §4.1.3, where we discussed how to parametrize a general WFSA and use it as a globally normalized model—in contrast, in this section, we consider how to fit a (locally normalized) n-gram model into the finite-state framework.

The intuition behind the connection is simple: the finite length of the context implies a finite number of histories we have to model. These histories represent the different states the corresponding automaton can reside in at any point.

Given any history $\boldsymbol{y}$ with $|\boldsymbol{y}| < n$ and the state $q \in Q$ representing $\boldsymbol{y}$, then, the conditional distribution of the next token given $\boldsymbol{y}$ dictate the transition weights into the next states in the WFSA, representing the new, updated history of the input.

Importantly, since we want PFSAs to represent globally-normalized models, we will also remove the EOS symbol from the n-gram model before transforming it into a PFSA—as the remark above about the relationship between the EOS symbol and the final states hints, the latter will fill in the role of the EOS symbol. The way we do that is the following. From the semantics of the EOS symbol discussed in the section on tightness (cf. Eq. (2.41)), we also know that to model the probability distribution over finite strings in Σ^*, we only require to keep track of strings up to the first occurrence of the EOS symbol. Therefore, when converting a given n-gram model to a WFSA, we will only model sequences up to the first occurrence of the special symbol, meaning that EOS will never occur in the *context* of any conditional distribution $p_{\text{SM}}(\overline{y} \mid \boldsymbol{y})$. We now detail this construction.

Let p_{LN} be a well-defined n-gram language model specified by conditional distributions p_{SM} as defined by Assumption 4.1.1. We will now construct a WFSA representing p_{LN}. Intuitively, its states will represent all possible sequences of words of length n while the transitions between the states q_1 and q_2 will correspond to the possible transitions between the n-grams which those represent. This means that the only possible (positively weighted) transitions will be between the n-grams which can follow each other, i.e. $\boldsymbol{y}_{t-n:t-1}$ and $\boldsymbol{y}_{t-n+2:t}$ for some $y_{t-n}, y_t \in \overline{\Sigma}$ (until the first occurrence of EOS). The transition's weight will depend on the probability of observing the "new" word y_0 in the second n-gram given the starting n-gram $y_{-n}y_{-(n-1)}\dots y_{-1}$. Further, the final weights of the states will correspond to *ending* the string in them. In p_{LN}, this is modeled as the probability of observing EOS given the context $\boldsymbol{y}_{t-n:t-1}$—this, therefore, is set as the final weight of the state representing the history $\boldsymbol{y}_{t-n:t-1}$. Formally, we can map a n-gram model into a WFSA $\mathcal{A} = (\Sigma_{\mathcal{A}}, Q_{\mathcal{A}}, \delta_{\mathcal{A}}, \lambda_{\mathcal{A}}, \rho_{\mathcal{A}})$ by constructing $\mathcal{A}$ as follows.

- Automaton's alphabet:

$$\Sigma_{\mathcal{A}} \stackrel{\text{def}}{=} \Sigma \tag{4.64}$$

- The set of states:

$$Q_{\mathcal{A}} \stackrel{\text{def}}{=} \bigcup_{t=0}^{n-1} \{\text{BOS}\}^{n-1-t} \times \Sigma^t \tag{4.65}$$

- The transitions set

$$\delta_{\mathcal{A}} \stackrel{\text{def}}{=} \{\boldsymbol{y}_{t-n:t-1} \xrightarrow{y_t/p_{\text{SM}}(y_t|\boldsymbol{y}_{t-n:t-1})} \boldsymbol{y}_{t-n+1:t} \mid \boldsymbol{y}_{t-n+1:t-1} \in \bigcup_{t=0}^{n-2} \{\text{BOS}\}^{n-2-t} \times \Sigma^t; y_{t-n}, y_t \in \Sigma\} \tag{4.66}$$

- The initial function:

$$\lambda_{\mathcal{A}} : \boldsymbol{y} \mapsto \begin{cases} \mathbf{1} & \text{if } \boldsymbol{y} = \underbrace{\text{BOS} \ldots \text{BOS}}_{n-1 \text{ times}} \\ \mathbf{0} & \text{otherwise} \end{cases} \tag{4.67}$$

- The final function

$$\rho_{\mathcal{A}} : \boldsymbol{y} \mapsto p_{\text{SM}}\left(\text{EOS} \mid \boldsymbol{y}\right), \ \boldsymbol{y} \in Q_{\mathcal{A}} \tag{4.68}$$

The definition of the states set $Q_{\mathcal{A}}$ captures exactly the notion of padding with the BOS symbol for handling the edge cases we described above. This shows that n-gram language models are indeed finite-state (we leave the formal proof showing that $L\left(\mathcal{A}\right) = L\left(p_{\text{LN}}\right)$ to the reader.

Defining a n-gram language model through a parametrized WFSA. We now consider how we can use the framework of WFSA to define a more "flexible" *parametrized globally* normalized model. In this case, we do not start from an existing locally normalized set of distributions forming p_{SM}. Rather, we would like to model the "suitability" of different n-grams following each other—that is, we would like to somehow *parametrize* the probability that some n-gram $\boldsymbol{y}'$ will follow an n-gram $\boldsymbol{y}$ without having to worry about normalizing the model at every step. This will allow us to then fit the probability distributions of the model to those in the data, e.g., with techniques described in §3.2.3. Luckily, the flexibility of the WFSA modeling framework allows us to do exactly that.

Subregularity

We saw that language models implementing the very natural n-gram assumption can be represented using weighted finite-state automata. However, n-gram models do not "need the full expressive power" of WFSAs—they can actually be modeled using even simpler machines than finite-state automata. This, along with several other examples of simple families of formal languages, motivates the definition of *subregular* languages.

Definition 4.1.25. *A language is **subregular** if it can be recognized by a finite-state automaton or any weaker machine.*

Most subregular languages can indeed be recognized by formalisms which are much simpler than FSAs. Many useful and interesting classes of subregular languages have been identified—recently, especially in the field of phonology. Naturally, due to their simpler structure, they also allow for more efficient algorithms—this is why we always strive to represent a language with the simplest formalism that still captures it adequately. See Jäger and Rogers (2012); Avcu et al. (2017) for comprehensive overviews of subregular languages.

Subregular languages actually form multiple hierarchies of complexity within regular languages. Interestingly, n-gram models fall into the simplest level of

complexity in one of the hierarchies, directly above *finite* languages. This class of subregular languages is characterized by patterns that depend solely on the blocks of symbols that occur *consecutively* in the string, which each of the blocks considered independently of the others—it is easy to see that n-gram models intuitively fall within such languages. This family of subregular languages is suggestively called *strictly local languages*.

Definition 4.1.26 (Strictly local languages)**.** *A language L is **strictly n-local** (SL_n) if, for every string $\boldsymbol{y}$ of length $|\boldsymbol{y}| = n{-}1$, and all strings $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{z}_1, \boldsymbol{z}_2 \in \Sigma^*$, it holds that if $\boldsymbol{x}_1\boldsymbol{y}\boldsymbol{z}_1 \in L$ and $\boldsymbol{x}_2\boldsymbol{y}\boldsymbol{z}_2 \in L$, then also $\boldsymbol{x}_1\boldsymbol{y}\boldsymbol{z}_2 \in L$ (and $\boldsymbol{x}_2\boldsymbol{y}\boldsymbol{z}_1 \in L$).*

*A language is **strictly local** (SL) if it is strictly n-local for any n.*

Note that we could of course also define this over with the EOS-augmented alphabet $\overline{\Sigma}$. You can very intuitively think of this definition as postulating that the history more than n symbols back does not matter anymore for determining or specifying whether a string is in a language (or its weight, in the weighted case)—this is exactly what the n-gram assumption states.

4.1.6 Representation-based n-gram Models

So far, we have mostly talked about the conditional probabilities and the WFSA weights defining a language model very abstractly. Apart from describing how one can generally parametrize the weights of the underlying WFSA with the scoring function in §4.1.5, we only discussed what values the weights can take for the language model to be well-defined and what implications that has on the distribution defined by the WFSA. In this section, we consider for the first time what an actual implementation of a finite-state, or more precisely, a n-gram language model might look like. Concretely, we will define our first parameterized language model in our General language modeling framework (cf. §3.1) by defining a particular form of the encoding function enc as a simple multi-layer feed-forward neural network.[18]

However, before we dive into that, let us consider as an alternative possibly the simplest way to define a (locally normalized) n-gram language model: by directly parametrizing the probabilities of each of the symbols y in the distribution $p_{\text{SM}}(\overline{y} \mid \overline{\boldsymbol{y}})$ for any context $\overline{\boldsymbol{y}}$, that is

$$\boldsymbol{\theta} \stackrel{\text{def}}{=} \left\{ \theta_{y|\boldsymbol{y}} \stackrel{\text{def}}{=} p_{\text{SM}}(y \mid \boldsymbol{y}) \mid y \in \overline{\Sigma}, \boldsymbol{y} \in \overline{\Sigma}^{n-1}, \theta_{y|\boldsymbol{y}} \geqslant 0, \sum_{y' \in \overline{\Sigma}} \theta_{y|\boldsymbol{y}} = 1 \right\}. \tag{4.69}$$

The following proposition shows that the maximum likelihood solution (Eq. (3.60)) to this parametrization is what you would probably expect.

[18] While we introduce particular architectures of neural networks, for example, recurrent neural networks and transformers later in **??**, we assume some familiarity with neural networks in general. See Chapter 6 of Goodfellow et al. (2016) for an introduction.

Proposition 4.1.3. *The MLE solution of Eq. (4.69) is*

$$p_{SM}\left(y_n \mid \boldsymbol{y}_{<n}\right) = \frac{C\left(y_1, \ldots, y_n\right)}{C\left(y_1, \ldots, y_{n-1}\right)} \tag{4.70}$$

whenever the denominator > 0*, where* $C\left(y_1, \ldots, y_n\right)$ *denotes the number of occurrences of all possible strings of the form* $y_1, \ldots, y_n$ *and* $C\left(y_1, \ldots, y_n\right)$ *denotes the number of occurrences of all possible strings of the form* $y_1, \ldots, y_{n-1}$*.*

Proof. Let $\mathcal{D} \stackrel{\text{def}}{=} \left\{\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(M)}\right\}$ be the training dataset. The log-likelihood of a single example $\boldsymbol{y}^{(m)}$ is

$$\log\left(p_{\text{LN}}\left(\boldsymbol{y}\right)\right) = \log\left(\prod_{t=1}^{|\boldsymbol{y}^{(m)}|} p_{\text{SM}}\left(y_t^{(m)} \mid y_{t-n:t-1}^{(m)}\right)\right) \tag{4.71}$$

$$= \sum_{t=1}^{|\boldsymbol{y}^{(m)}|} \log p_{\text{SM}}\left(y_t^{(m)} \mid y_{t-n:t-1}^{(m)}\right) \tag{4.72}$$

which means that the log-likelihood of the entire dataset is

$$\ell\ell\left(\mathcal{D}\right) = \sum_{m=1}^{M} \sum_{t=1}^{|\boldsymbol{y}^{(m)}|} \log p_{\text{SM}}\left(y_t^{(m)} \mid y_{t-n:t-1}^{(m)}\right) \tag{4.73}$$

$$= \sum_{m=1}^{M} \sum_{t=1}^{|\boldsymbol{y}^{(m)}|} \log \theta_{y_n|\boldsymbol{y}<n}. \tag{4.74}$$

Exercise 4.2 asks you to show that this can be rewritten with the **token to type switch** as

$$\ell\ell\left(\mathcal{D}\right) = \sum_{\substack{\boldsymbol{y} \\ |\boldsymbol{y}|=n}} C\left(\boldsymbol{y}\right) \theta_{y_n|\boldsymbol{y}<n}. \tag{4.75}$$

The maximum likelihood parameters can then be determined using Karush–Kuhn–Tucker (KKT) conditions[19] to take into account the non-negativity and local normalization constraints:

$$\nabla_{\boldsymbol{\theta}}\left(\ell\ell\left(\mathcal{D}\right) - \sum_{\substack{\boldsymbol{y}\in\Sigma^* \\ |\boldsymbol{y}|=n-1}} \lambda_{\boldsymbol{y}y}\left(\sum_{y\in\Sigma} \theta_{y|\boldsymbol{y}} - 1\right) - \sum_{\substack{\boldsymbol{y}\in\Sigma^* \\ |\boldsymbol{y}|=n-1}} \eta_{\boldsymbol{y}y}\theta_{y|\boldsymbol{y}}\right) = 0. \tag{4.76}$$

Recall that the KKT conditions state that a $\boldsymbol{\theta}$ is an optimal solution of $\ell\ell$ if and only if $\left(\boldsymbol{\theta}, \left\{\lambda_{\boldsymbol{y}y'}\right\}_{\boldsymbol{y}\in\Sigma^{n-1}, y\in\Sigma}, \left\{\eta_{\boldsymbol{y}y'}\right\}_{\boldsymbol{y}\in\Sigma^{n-1}, y\in\Sigma}\right)$ satisfy Eq. (4.76). Since this is simply a sum over the dataset with no interactions of parameters for individual contexts $\boldsymbol{y}$ with $|\boldsymbol{y}| = n-1$ in $\theta_{y|\boldsymbol{y}}$, it can be solved for each context $\boldsymbol{y}$ individually.

[19] See https://en.wikipedia.org/wiki/Karush%E2%80%93Kuhn%E2%80%93Tucker_conditions.

Moreover, as you are asked to show Exercise 4.3, it holds that

$$\sum_{y' \in \Sigma} C\left(y_1 \dots y_{n-1} y'\right) = C\left(y_1 \dots y_{n-1}\right) \tag{4.77}$$

for any $\boldsymbol{y} = y_1 \dots y_{n-1} \in \Sigma^{n-1}$. This leaves us with the following system for each $\boldsymbol{y} \in \Sigma^{n-1}$:

$$\sum_{y' \in \Sigma} C\left(\boldsymbol{y}\boldsymbol{y}'\right) \log \theta_{y'|\boldsymbol{y}} - \lambda_{\boldsymbol{y}} \left(\sum_{\boldsymbol{y}' \in \Sigma} \theta_{y'|\boldsymbol{y}} - 1 \right) - \sum_{y' \in \Sigma} \eta_{\boldsymbol{y}y'} \theta_{y'|\boldsymbol{y}}. \tag{4.78}$$

It is easy to confirm that $\theta_{y|\boldsymbol{y}} = \frac{C(\boldsymbol{y}y)}{C(\boldsymbol{y})}$ with $\lambda_{\boldsymbol{y}} = C\left(\boldsymbol{y}\right)$ and $\eta_{\boldsymbol{y}y'} = 0$ is a saddle point of Eq. (4.76). This means that $\theta_{y|\boldsymbol{y}} = \frac{C(\boldsymbol{y}y)}{C(\boldsymbol{y})}$ is indeed the maximum likelihood solution. ■

This results in a locally normalized n-gram model. To avoid issues with division-by-zero and assigning 0 probability to unseen sentences, we can employ methods such as *smoothing* and *backoff*, which are beyond the scope of the course.[20]

While this model might seem like an obvious choice, it comes with numerous drawbacks. To see what can go wrong, consider the following example.

Example 4.1.6. *Suppose we have a large training corpus of sentences, among which sentences like "We are going to the shelter to adopt a dog.", "We are going to the shelter to adopt a puppy.", and "We are going to the shelter to adopt a kitten.", however, without the sentence "We are going to the shelter to adopt a cat." Fitting an n-gram model using the count statistics and* individual *tables of conditional probabilities* $p_{SM}(y \mid \boldsymbol{y})$*, we would assign the probability*

$$p_{SM}(y_t = \textit{cat} \mid \boldsymbol{y}_{<t} = \textit{We are going to the shelter to adopt a})$$

the value 0 *(or some "default" probability if we are using smoothing). However, the words "dog", "puppy", "kitten", and "cat" are semantically very similar—they all describe pets often found in shelters. It would therefore be safe to assume that the word "cat" is similarly probable given the context "We are going to the shelter to adopt a" as the other three words observed in the training dataset. However, if we estimate all the conditional probabilities* independently*, we have no way of using this information—the words have no relationship in the alphabet, they are simply different indices in a lookup table. Additionally, statistics gathered for the sentences above will not help us much when encountering very similar sentences, such as "We went to a nearby shelter and adopted a kitten." The issue is that there are simply many ways of expressing similar intentions. We would thus like our language models to be able to generalize across different surface forms and make use of more "semantic" content of the sentences and words. However, if the model is parametrized as defined in Eq.* (4.69)*, it is not able to take advantage of any such relationships.*

[20] See (Chen and Goodman, 1996) and Chapter 4 in (Jurafsky and Martin, 2009).

The model defined by Eq. (4.69) is therefore unable to take into account the relationships and similarities between words. The general modeling framework defined in §3.1 allows us to remedy this using the **distributed word representations**. Recall that, in that framework, we associate each word y with its vector representation $\mathbf{e}(y)$ (its *embedding*), and we combine those into the embedding matrix $\mathbf{E}$. Importantly, word embeddings are simply additional parameters of the model and can be *fit* on the training dataset *together* with the language modeling objective. One of the first successful applications of $\mathsf{enc}(\cdot)$ is due to Bengio et al. (2003), which we discuss next.

To be able to use the embeddings in our general framework, we now just have to define the concrete form of the context-encoding function enc. In the case of the neural n-gram model which we consider here and as defined by (Bengio et al., 2003), the representations of the context $\boldsymbol{y}_{<t}$, $\mathsf{enc}\left(\boldsymbol{y}_{<t}\right)$, are defined as the output of a neural network which looks at the previous $n-1$ words in the context:

$$\mathsf{enc}(\boldsymbol{y}_{<t}) \stackrel{\text{def}}{=} \mathsf{enc}(y_{t-1}, y_{t-2}, \ldots, y_{t-n+1}), \tag{4.79}$$

where enc is a neural network we define in more detail shortly. The full language model is therefore defined through the conditional distributions

$$p_{\text{SM}}\left(\overline{y}_t \mid \overline{\boldsymbol{y}}_{<t}\right) \stackrel{\text{def}}{=} \text{softmax}\left(\mathsf{enc}\left(\overline{y}_{t-1}, \overline{y}_{t-2}, \ldots, \overline{y}_{t-n+1}\right)^\top \mathbf{E} + \mathbf{b}\right)_{\overline{y}_t} \tag{4.80}$$

resulting in the locally normalized model

$$p_{\text{LN}}\left(\boldsymbol{y}\right) = \text{softmax}\left(\mathsf{enc}\left(\overline{y}_T, \overline{y}_{T-1}, \ldots, \overline{y}_{T-n+2}\right)^\top \mathbf{E} + \mathbf{b}\right)_{\text{EOS}} \tag{4.81}$$

$$\cdot \prod_{t=1}^{T} \text{softmax}\left(\mathsf{enc}\left(y_{t-1}, y_{t-2}, \ldots, y_{t-n+1}\right)^\top \mathbf{E} + \mathbf{b}\right)_{y_t} \tag{4.82}$$

for $\boldsymbol{y} \in \Sigma^*$.

Importantly, notice that although this is a *neural* model, it is nonetheless still an n-gram model with finite context—Eq. (4.79) is simply a restatement of the n-gram assumption in terms of the neural encoding function enc. It therefore still suffers from some of the limitations of regular n-gram models, such as the inability to model dependencies spanning more than n words. However, it solves the problems encountered in Example 4.1.6 by considering word similarities and *sharing* parameters across different contexts in the form of an encoding function rather than a lookup table.

While encoding function enc in Eq. (4.79) could in principle take any form, the original model defined in Bengio et al. (2003) defines the output as for the string $\boldsymbol{y} = y_t, y_{t-1}, \ldots, y_{t-n+1}$ as

$$\mathsf{enc}\left(y_t, y_{t-1}, \ldots, y_{t-n+1}\right) \stackrel{\text{def}}{=} \mathbf{b} + \mathbf{W}\mathbf{x} + \mathbf{U}\tanh\left(\mathbf{d} + \mathbf{H}\mathbf{x}\right), \tag{4.83}$$

where $\mathbf{x} \stackrel{\text{def}}{=} \text{concat}\left(\mathbf{e}(y_t), \mathbf{e}(y_{t-1}), \ldots, \mathbf{e}(y_{t-n+1})\right)$ denotes the concatenation of the context symbol embeddings into a long vector of size $(n-1) \cdot R$, and $\mathbf{b}$, $\mathbf{d}$, $\mathbf{W}$, and $\mathbf{U}$ define the parameters of the encoding function. This completes our

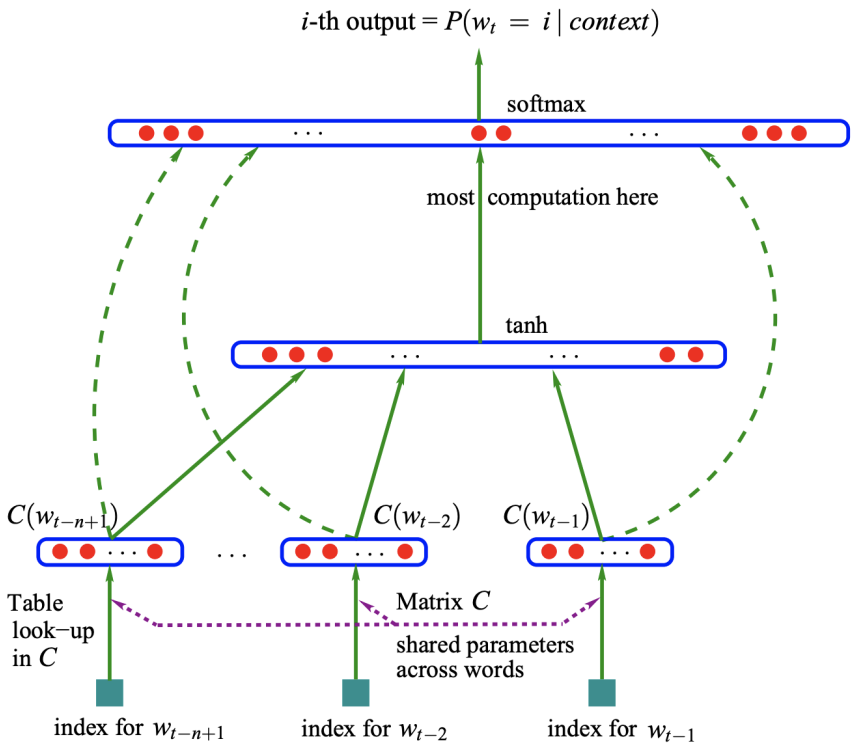


Figure 4.4: A pictorial depiction of the n-gram neural language model from the original publication (Bengio et al., 2003). Note that the quantity $C(w)$ corresponds to $\mathbf{e}(y)$ for a word y in our notation.

definition of the model in the general language modeling framework—the model can then simply be trained on the language modeling objective as defined in §3.2.2.

We can also see that such a model also reduces the number of parameters required to specify a n-gram model: whereas a lookup-table-based n-gram model with no parameter sharing requires $\mathcal{O}(|\Sigma|^n)$ parameters to be defined, the number of parameters required by a representation-based n-gram model scales *linearly* with n—all we have to do is add additional rows to the matrices defined in Eq. (4.83). We will later see how this can be reduced to a *constant* number of parameters w.r.t. the sequence length in the case of recurrent neural networks in §5.1.2.

Pictorially, we can imagine the model as depicted in Fig. 4.4 (taken from the original publication). This shows that the n-gram modeling framework is not limited to counting co-occurrence statistics. The model from Eq. (4.79) can also be represented by a WFSA just like the simpler models we discussed above, with the weights on the transitions parametrized by the neural network. This

allows us to both understand well with insights from formal language theory, as well as to train it in a flexible way allowed for by the non-linear encoding function. However, the model from Eq. (4.79) is still limited to statistics of the last n tokens or less. If we want to model arbitrarily long dependencies and hierarchical structures, we have to leave the space of finite-state languages behind and develop formalisms capable of modeling more complex languages. The next section explores the first of such frameworks: context-free languages with the computational models designed to model them.

4.2 Pushdown Language Models

An strong limitation of finite-state language models is that they can definitionally only distinguish a finite set of contexts. However, human language has inherently more structure than what a finite set of contexts can encode. For example, human language contains arbitrarily deep recursive structures which cannot be captured by a finite set of possible histories—we will see an example of this soon in §4.2.1.

To be able to model these structures we are climbing a rung higher on the ladder of the hierarchy of formal languages: we are going to consider **context-free languages**, a larger class of languages than regular languages. Luckily, we will see that a lot of the formal machinery we introduce in this section closely follows analogs from the finite-state section and we invite the reader to pay close attention to the parallels. For example, similarly to how we weighted a string in a regular language by summing over the weights of the paths labeled with that string, we will weight strings in context-free languages by summing over analogous structures.

To be able to recognize context-free languages, will have to extend finite-state automata from §4.1.1 with an additional data structure—the stack. Finite-state automata augmented with a stack are called pushdown automata. We introduce them in §4.2.7. Before giving a formal treatment of pushdown automata, however, we will discuss an arguably more natural formalism for generating the context-free languages—context-free grammars.[21]

In the last part of the section, we will then further extend the regular pushdown automaton with an *additional* stack. Interestingly, this will make it more powerful: as we will see, it will raise its expressive power from context-free languages to all computable languages, as it is Turing complete. While this augmentation will not be immediately useful from a language modeling perspective, we will then later use this machine to prove some theoretical properties of other modern language models we consider later in the course.

4.2.1 Human Language is not Finite-state

As hinted above, human language contains structures that cannot be modeled by finite-state automata. Before we introduce ways of modeling context-free languages, let us, therefore, first motivate the need for a more expressive formalism by more closely considering a specific phenomenon often found in human language: recursive hierarchical structure. We discuss it through an example, based on Jurafsky and Martin (2009).

Example 4.2.1 (Center embeddings)**.** *Consider the sentence:*

"The cat likes to cuddle."

[21] You might wonder what non-context-free grammars are: a superclass of context-free grammars is that of context-sensitive grammars, in which a production rule may be surrounded by a left and right context. They are still however a set of restricted cases of general grammars, which are grammars that can emulate Turing machines.

It simply describes a preference of a cat. However, we can also extend it to give additional information about the cat:

"The cat the dog barked at likes to cuddle."

This sentence, in turn, can be extended to include additional information about the dog:

"The cat the dog the mouse startled barked at likes to cuddle."

Of course, we can continue on:

"The cat the dog the mouse the rat frightened startled barked at likes to cuddle."

and on:

"The cat the dog the mouse the rat the snake scared frightened startled barked at likes to cuddle."

In theory, we could continue like this for as long as we wanted—all these sentences are grammatically correct *—this is an instance of the so-called* ***center embeddings****.*

Crucially, such sentences cannot be captured by a regular language, i.e., a language based on an automaton with finitely many states. While we would need formal machinery beyond the scope of this course to formally prove this, the intuition is quite simple. By adding more and more "levels" of recursion to the sentences (by introducing more and more animals in the chain), we unboundedly increase the amount of information the model has to "remember" about the initial parts of the sentence while processing it sequentially, to be able to process or generate the matching terms on the other end of the sentence correctly. Because such hierarchies can be arbitrarily deep (and thus the sentences arbitrarily long), there is no bound on the number of states needed to remember them, which means they cannot be captured by a finite-state automaton.

Note that this example also touches upon the distinction of the grammatical competence *versus grammatical* performance *(Chomsky, 1959; Chomsky and Schützenberger, 1963; Chomsky, 1965). The former refers to the purely theoretical properties of human language, for example, the fact that such hierarchical structures can be arbitrarily long and still grammatically correct. Grammatical performance, on the other hand, studies language grounded more in the way people actually use it. For example, nested structures like the one above are never very deep in day-to-day speech—indeed, you probably struggled to understand the last few sentences above. We rarely come across nestings of depth more than three in human language (Miller and Chomsky, 1963; Jin et al., 2018; Karlsson, 2007).*

4.2.2 Context-free Grammars

How can we capture recursive structures like those in Example 4.2.1 and the long-term dependencies arising from them? The first formalism modeling such phenomena we will introduce is context-free grammars: a *generative* formalism

which can tell us how to generate or "compose" strings in the language it describes. Later in the section (§4.2.7), we will introduce the context-free analog of finite-state automata, which will tell us how to *recognize* whether a string is in a context-free language (rather than generate a string): pushdown automata.

Definition 4.2.1 (Context-free Grammar)**.** *A **context-free grammar** (CFG) is a 4-tuple $\mathcal{G} = (\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P})$ where Σ is an alphabet of terminal symbols, $\mathcal{N}$ is a non-empty set of non-terminal symbols with $\mathcal{N} \cap \Sigma = \varnothing$, $\mathrm{S} \in \mathcal{N}$ is the designated start non-terminal symbol and $\mathcal{P}$ is the set of production rules, where each rule $p \in \mathcal{P}$ is of the form $\mathrm{X} \to \boldsymbol{\alpha}$ with $\mathrm{X} \in \mathcal{N}$ and $\boldsymbol{\alpha} \in (\mathcal{N} \cup \Sigma)^*$.*[22]

Example 4.2.2 (A simple context-free grammar)**.** *Let $\mathcal{G} = (\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P})$ be defined as follows:*

- $\Sigma = \{a, b\}$
- $\mathcal{N} = \{\mathrm{X}\}$
- $\mathrm{S} = \mathrm{X}$
- $\mathcal{P} = \{\mathrm{X} \to a\mathrm{X}b, \mathrm{X} \to \varepsilon\}$

This defines a simple context-free grammar. We will return to it later, when we will formally show that it generates the language $L = \{a^n b^n \mid n \in \mathbb{N}_{\geqslant 0}\}$.

Rule Applications and Derivations

Context-free grammars allow us to generate strings $\boldsymbol{y} \in \Sigma^*$ by *applying* production rules on its non-terminals. We apply a production rule $\mathrm{X} \to \boldsymbol{\alpha}$ to $\mathrm{X} \in \mathcal{N}$ in a rule p by taking X on the right-hand side of p and replacing it with $\boldsymbol{\alpha}$.[23]

Definition 4.2.2 (Rule Application)**.** *A production rule $\mathrm{Y} \to \boldsymbol{\beta}, \boldsymbol{\beta} \in (\mathcal{N} \cup \Sigma)^*$, is **applicable** to Y in a rule p, if p takes the form*

$$\mathrm{X} \to \boldsymbol{\alpha}\,\mathrm{Y}\,\boldsymbol{\gamma}, \quad \boldsymbol{\alpha}, \boldsymbol{\gamma} \in (\mathcal{N} \cup \Sigma)^*.$$

*The **result of applying** $\mathrm{Y} \to \boldsymbol{\beta}$ to $\boldsymbol{\alpha}\,\mathrm{Y}\,\boldsymbol{\gamma}$ is $\boldsymbol{\alpha}\,\boldsymbol{\beta}\,\boldsymbol{\gamma}$.*

Starting with S, we apply $\mathrm{S} \to \boldsymbol{\alpha}$ to S for some $(\mathrm{S} \to \boldsymbol{\alpha}) \in \mathcal{P}$, then take a non-terminal in $\boldsymbol{\alpha}$ and apply a new production rule.[24] To generate a string we follow this procedure until all non-terminal symbols have been transformed into terminal symbols. The resulting string, i.e., the **yield**, will be the string taken by concatenating all terminal symbols read from left to right. More formally, a derivation can be defined as follows.

[22] As is the case for initial states in FSAs, multiple start symbols could be possible. However we consider only one for the sake of simplicity.

[23] We say that X is on the right-hand side of a rule p if p takes the form $p = (\mathrm{Y} \to \boldsymbol{\alpha}\,\mathrm{X}\boldsymbol{\gamma})$, where $\boldsymbol{\alpha}, \boldsymbol{\gamma} \in (\mathcal{N} \cup \Sigma)^*$. We will sometimes refer to X as the *head* of the production rule $\mathrm{X} \to \boldsymbol{\alpha}$, and the right-hand side $\boldsymbol{\alpha}$ as the *body* of the production rule.

[24] We will write $\mathrm{X} \in \boldsymbol{\alpha}$, which formally means a substring of $\boldsymbol{\alpha}$ with length 1. Unless otherwise stated, X can be either a non-terminal or a terminal.

Definition 4.2.3 (Derivation)**.** *A **derivation** in a grammar $\mathcal{G}$ is a sequence $\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_M$, where $\boldsymbol{\alpha}_1 \in \mathcal{N}$, $\boldsymbol{\alpha}_2, \ldots, \boldsymbol{\alpha}_{M-1} \in (\mathcal{N} \cup \Sigma)^*$ and $\boldsymbol{\alpha}_M \in \Sigma^*$, in which each $\boldsymbol{\alpha}_{m+1}$ is formed by applying a production rule in $\mathcal{P}$ to $\boldsymbol{\alpha}_m$.*

We say that $\boldsymbol{\alpha} \in (\mathcal{N} \cup \Sigma)^*$ is derived from $\mathrm{X} \in \mathcal{N}$ if we can apply a finite sequence of production rules to generate $\boldsymbol{\alpha}$ starting from X. We will denote this as $\mathrm{X} \overset{*}{\Rightarrow}_{\mathcal{G}} \boldsymbol{\alpha}$. See the following formal definition.

Definition 4.2.4 (Derives)**.** *Let $\mathcal{G} \overset{\text{def}}{=} (\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P})$ be a CFG. We say that* X ***derives*** *$\boldsymbol{\beta}$ under the grammar $\mathcal{G}$, denoted as $\mathrm{X} \Rightarrow_{\mathcal{G}} \boldsymbol{\beta}$ if $\exists p \in \mathcal{P}$ such that $p = (\mathrm{X} \rightarrow \boldsymbol{\alpha}\,\boldsymbol{\beta}\,\boldsymbol{\gamma})$, $\boldsymbol{\alpha}, \boldsymbol{\gamma} \in (\mathcal{N} \cup \Sigma)^*$ and $\boldsymbol{\beta} \in (\mathcal{N} \cup \Sigma)^* \backslash \{\varepsilon\}$. The special case $\mathrm{X} \Rightarrow_{\mathcal{G}} \varepsilon$ holds iff $\mathrm{X} \rightarrow \varepsilon \in \mathcal{P}$. We denote the reflexive transitive closure of the $\Rightarrow_{\mathcal{G}}$ relation as $\overset{*}{\Rightarrow}_{\mathcal{G}}$. We say that $\boldsymbol{\beta}$ is **derived from** X if $\mathrm{X} \overset{*}{\Rightarrow}_{\mathcal{G}} \boldsymbol{\beta}$.*

The (context-free) language of a CFG $\mathcal{G}$ is defined as all the strings $\boldsymbol{y} \in \Sigma^*$ that can be derived from the start symbol S of $\mathcal{G}$, or alternatively, the set of all yields possible from derivations in $\mathcal{G}$ that start with S. We will denote the language generated by $\mathcal{G}$ as $L(\mathcal{G})$.

Definition 4.2.5 (Language of a Grammar)**.** *The **language** of a context-free grammar $\mathcal{G}$ is*

$$L(\mathcal{G}) = \{\boldsymbol{y} \in \Sigma^* \mid \mathrm{S} \overset{*}{\Rightarrow}_{\mathcal{G}} \boldsymbol{y}\} \tag{4.84}$$

Parse Trees and Derivation Sets

A natural representation of a derivation in a context-free grammar is a **derivation tree** $\boldsymbol{d}$ (also known as a parse tree). A derivation tree represents the sequence of applied rules in a derivation with a directed tree. The tree's internal nodes correspond to the non-terminals in the derivation, and each of their children corresponds to a symbol (from $\Sigma \cup \mathcal{N}$) on the right side of the applied production in the derivation. The leaves, representing terminal symbols, "spell out" the derived string—the tree's yield. More formally, for each production rule $\mathrm{X} \rightarrow \boldsymbol{\alpha}$, the node corresponding to the specific instance of the non-terminal X in the derivation is connected to the nodes corresponding to $\mathrm{Y} \in \boldsymbol{\alpha}$ where $\mathrm{Y} \in \Sigma \cup \mathcal{N}$.

We will mostly be interested in representing derivations starting with S—the root node of a tree representing any such derivation will correspond to S. We will denote the string generated by a tree $\boldsymbol{d}$—its yield—by $\mathsf{yield}\,(\boldsymbol{d})$. See Fig. 4.5 for examples of parse trees for the grammar from Example 4.2.2.

Importantly, a grammar may in fact admit *multiple* derivations and hence multiple derivation trees for any given string.

Example 4.2.3 (Multiple derivation strings)**.** *It is relatively easy to see that in the grammar $\mathcal{G}$, each string $a^n b^n$ is only generated by a single derivation tree—each new pair of symbols a and b can only be added by applying the rule $\mathrm{X} \rightarrow a\mathrm{X}b$ and the string $a^n b^n$ can only be generated by the application of the rule $\mathrm{X} \rightarrow a\mathrm{X}b$ n times and the rule $\mathrm{X} \rightarrow \varepsilon$ once in this order.*

However, we can modify $\mathcal{G}$ by adding, for instance, a non-terminal Y *and rules $\mathrm{X} \rightarrow \mathrm{Y}, \mathrm{Y} \rightarrow \varepsilon$. The empty string ε may then be derived either by $(\mathrm{X} \rightarrow \varepsilon)$,*

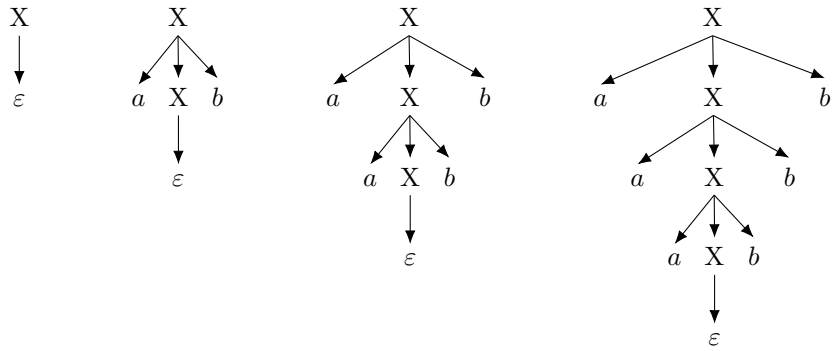


Figure 4.5: A sequence of derivation trees for the strings in $\{a^n b^n \mid n = 0, 1, 2, 3\}$ in the grammar from Example 4.2.2.

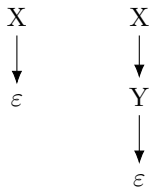


Figure 4.6: Two parse trees in the modified grammar $\mathcal{G}$ yielding ε.

or $(\mathrm{Y} \to \mathrm{Y}), (\mathrm{Y} \to \varepsilon)$*, corresponding to two separate derivation trees, as shown in Fig. 4.6. The set of these two trees comprises what we call the derivation set of* ε*.*

We denote a derivation set of a string $\boldsymbol{y}$, generated by the grammar $\mathcal{G}$, as $\mathcal{D}_{\mathcal{G}}(\boldsymbol{y})$.

Definition 4.2.6 (String derivation set)**.** *Let* $\boldsymbol{y} \in \Sigma^*$*. Its* ***derivation set****, denoted by* $\mathcal{D}_{\mathcal{G}}(\boldsymbol{y})$ *is defined as*

$$\mathcal{D}_{\mathcal{G}}(\boldsymbol{y}) \stackrel{\text{def}}{=} \{\boldsymbol{d} \mid \mathit{yield}(\boldsymbol{d}) = \boldsymbol{y}\}. \tag{4.85}$$

We say that a grammar is **unambiguous** if, for every string that can be generated by the grammar, there is only one associated derivation tree.

Definition 4.2.7 (Unambiguity)**.** *A grammar* $\mathcal{G}$ *is* ***unambiguous*** *if for all* $\boldsymbol{y} \in L(\mathcal{G})$*,* $|\mathcal{D}_{\mathcal{G}}(\boldsymbol{y})| = 1$*.*

The converse holds for **ambiguous** grammars.

Definition 4.2.8 (Ambiguity)**.** *A grammar* $\mathcal{G}$ *is* ***ambiguous*** *if* $\exists \boldsymbol{y} \in L(\mathcal{G})$ *such that* $|\mathcal{D}_{\mathcal{G}}(\boldsymbol{y})| > 1$*.*

The set of all derivation trees in a grammar is its derivation set.

Definition 4.2.9 (Grammar derivation set)**.** *The* ***derivation set of a grammar****,* $\mathcal{D}_\mathcal{G}$*, is the set of all derivations possible under the grammar. More formally, it can be defined as the union over the derivation set for the strings in its language,*

$$\mathcal{D}_\mathcal{G} \stackrel{\text{def}}{=} \bigcup_{\boldsymbol{y}' \in L(\mathcal{G})} \mathcal{D}_\mathcal{G}(\boldsymbol{y}') \tag{4.86}$$

Definition 4.2.10 (Non-terminal derivation set)**.** *The* ***derivation set of a non-terminal*** $\mathrm{Y} \in \mathcal{N}$ *in* $\mathcal{G}$*, denoted* $\mathcal{D}_\mathcal{G}(\mathrm{Y})$*, is defined as the set of derivation subtrees with root node* Y*.*

Note that $\mathcal{D}_\mathcal{G}$ could be defined as $\mathcal{D}_\mathcal{G}(\mathrm{S})$. For a terminal symbol $a \in \Sigma$, we trivially define the derivation set $\mathcal{D}_\mathcal{G}(a)$ to be empty.[25]

In cases where it is irrelevant to consider the order of the production rules in a derivation tree, we will write $(\mathrm{X} \to \boldsymbol{\alpha}) \in \boldsymbol{d}$ to refer to specific production rules in the tree—viewing trees as multisets (or bags) over the production rules they include.

Example 4.2.4 (Nominal Phrases)**.** *CFGs are often used to model natural languages. Terminals would then correspond to words in the natural language, strings would be text sequences and non-terminals would be abstractions over words. As an example, consider a grammar* $\mathcal{G}$ *that can generate a couple of nominal phrases. We let* $\mathcal{N} = \{\mathrm{Adj}, \mathrm{Det}, \mathrm{N}, \mathrm{Nominal}, \mathrm{NP}\}$*,* $\Sigma = \{a, \textit{big}, \textit{female}, \textit{giraffe}, \textit{male}, \textit{tall}, \textit{the}\}$*,* $\mathrm{S} = \mathrm{Nominal}$ *and define the following production rules:*

$$\begin{aligned}
&\mathrm{Nominal} \to \mathrm{Det\ NP}\\
&\mathrm{NP} \to \mathrm{N} \mid \mathrm{Adj\ NP}\\
&\mathrm{Det} \to a \mid \textit{the}\\
&\mathrm{N} \to \textit{female} \mid \textit{giraffe} \mid \textit{male}\\
&\mathrm{Adj} \to \textit{big} \mid \textit{female} \mid \textit{male} \mid \textit{tall}
\end{aligned}$$

See Fig. 4.7 for a few examples of derivation trees in this grammar.

Example 4.2.5 (The geralized Dyck languages $\mathrm{D}(k)$)**.** *A very widely studied family of context-free languages are the Dyck-k languages,* $\mathrm{D}(k)$*, the languages of well-nested brackets of k types. They are, in some ways, archetypal context-free languages (Chomsky and Schützenberger, 1963). Formally, we can define them as follows.*

Definition 4.2.11 ($\mathrm{D}(k)$ languages)**.** *Let* $k \in \mathbb{N}$*. The* $\mathrm{D}(k)$ *language is the language of the following context-free grammar* $\mathcal{G} \stackrel{\text{def}}{=} (\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P})$

- $\Sigma \stackrel{\text{def}}{=} \{\langle_n \mid n = 1, \ldots, k\} \cup \{\rangle_n \mid n = 1, \ldots, k\}$
- $\mathcal{N} \stackrel{\text{def}}{=} \{\mathrm{S}\}$

[25] Empty derivation sets for terminal symbols is defined solely for ease of notation later.

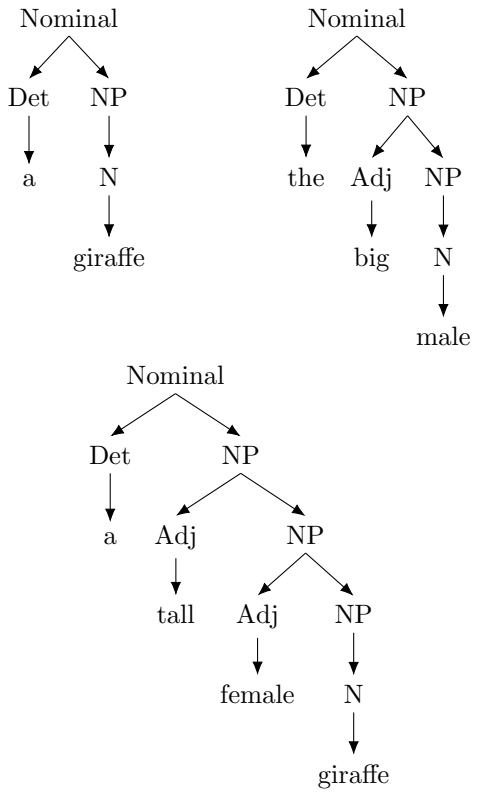


Figure 4.7: Derivation trees for natural language nominal phrases.

- $\mathrm{S} \stackrel{\text{def}}{=} \mathrm{S}$
- $\mathcal{P} \stackrel{\text{def}}{=} \{\mathrm{S} \to \varepsilon, \mathrm{S} \to \mathrm{SS}\} \cup \{\mathrm{S} \to \langle_n \mathrm{S} \rangle_n \mid n = 1, \ldots, k\}$

Examples of strings in the language $\mathrm{D}(3)$ *would be* $\langle_3\rangle_3\langle_2\rangle_2\langle_1\rangle_1$, $\langle_3\rangle_3\langle_1\langle_2\langle_2\rangle_2\rangle_2\rangle_1$, *and* $\langle_1\langle_2\langle_2\rangle_2\rangle_2\langle_3\langle_1\rangle_1\rangle_3\rangle_1$. *The string* $\langle_2\langle_2\rangle_1\rangle_2$ *is not in the language* $\mathrm{D}(3)$.

To give you a taste of what formally working with context-free grammars might look like, we now formally show that the grammar from Example 4.2.2 really generates the language $L = \{a^n b^n \mid n \in \mathbb{N}_{\geqslant 0}\}$, as we claimed.

Example 4.2.6 (Recognizing $a^n b^n$)**.** *The language* $L = \{a^n b^n \mid n \in \mathbb{N}\}$ *is not regular.*[26] *However, we can show that it is context-free and recognized exactly by the simple grammar from Example 4.2.2. We restate it here for convenience:* $\mathcal{G} = (\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P})$ *with* $\mathcal{N} = \{\mathrm{X}\}$, $\Sigma = \{a, b\}$, $\mathrm{S} = \mathrm{X}$, $\mathcal{P} = \{\mathrm{X} \to a\mathrm{X}b, \mathrm{X} \to \varepsilon\}$.

Lemma 4.2.1. *Given the grammar* $\mathcal{G}$ *defined above, we have* $L(\mathcal{G}) = \{a^n b^n \mid n \in \mathbb{N}\}$.

[26] Again, while the intuition behind this is similar to our reasoning from Example 4.2.1, this would have to be proven using the so-called pumping lemma for regular languages.

Proof. We will show that $L = L(\mathcal{G})$ in two steps: (i) showing that $L \subseteq L(\mathcal{G})$ and (ii) showing that $L(\mathcal{G}) \subseteq L$. Define $\boldsymbol{y}_n = a^n b^n$.

(i) We first need to show that each $\boldsymbol{y} \in L$ can be generated by $\mathcal{G}$, which we will do by induction.

Base case ($n = 0$) We have that $\boldsymbol{y}_0 = \varepsilon$, which is generated by $\boldsymbol{d} = (\mathrm{X} \to \varepsilon)$.

Inductive step ($n > 1$) We have that $\boldsymbol{y}_n$ is generated by

$$\boldsymbol{d} = \underbrace{(\mathrm{X} \to a\mathrm{X}b) \cdots (\mathrm{X} \to a\mathrm{X}b)}_{n \text{ times}}(\mathrm{X} \to \varepsilon).$$

It is then easy to see that $\boldsymbol{y}_{n+1}$ is generated by the derivation we get by replacing the last rule $(\mathrm{X} \to \varepsilon)$ with $(\mathrm{X} \to a\mathrm{X}b)(\mathrm{X} \to \varepsilon)$—they are exactly the trees illustrated in Fig. 4.5.

(ii) Next, we show that for each $\boldsymbol{d} \in \mathcal{D}_{\mathcal{G}}$, we have that $\boldsymbol{y}(\boldsymbol{d}) \in L$.

Base case ($\boldsymbol{d} = (\mathrm{X} \to \varepsilon)$) It is trivial to see that the derivation $\boldsymbol{d} = (\mathrm{X} \to \varepsilon)$ yields $\boldsymbol{y}(\boldsymbol{d}) = \varepsilon$.

Inductive step Now observe that $\mathcal{P}$ only contains two production rules and one non-terminal. Starting with X, we can either apply $\mathrm{X} \to a\mathrm{X}b$ to get one new non-terminal X, or apply $\mathrm{X} \to \varepsilon$ to terminate the process. Hence, if we fix the length of the sequence of production rules, there is no ambiguity in which string will be generated. Thus, by induction, we conclude that if we have a derivation tree given by $\underbrace{(\mathrm{X} \to a\mathrm{X}b), \ldots, (\mathrm{X} \to a\mathrm{X}b)}_{n \text{ times}}, (\mathrm{X} \to \varepsilon)$ generating $a^n b^n$, the derivation tree given by $\underbrace{(\mathrm{X} \to a\mathrm{X}b), \ldots, (\mathrm{X} \to a\mathrm{X}b)}_{n+1 \text{ times}}, (\mathrm{X} \to \varepsilon)$ will generate $a^{n+1} b^{n+1}$. ■

Reachable Non-terminals and Pruning

Similarly to how some states in a WFSA can be useless in the sense that they are not accessible from an initial state or might not lead to a final state, so too can non-terminals in a CFG be useless by not beaing reachable from the start symbol or might not lead to any string of terminals. In the context of CFGs, we typically use a different terminology: "reachable" instead of "accessible" and "generating" instead of "co-accessible".

Definition 4.2.12 (Accessibility for CFGs)**.** *A symbol* $\mathrm{X} \in \mathcal{N} \cup \Sigma$ *is* ***reachable*** *(or accessible) if* $\overset{*}{\Rightarrow}\mathrm{SX}$.

Definition 4.2.13 (Co-accessibility for CFGs)**.** *A non-terminal* Y *is* ***generating*** *(or co-accessible) if* $\exists \boldsymbol{y} \in \Sigma^*$ *such that* $\overset{*}{\Rightarrow}\mathrm{Y}\boldsymbol{y}$.

In words, reachable symbols are those that can be derived from the start symbol, whereas generating non-terminals are those from which at least one string (including the empty string) can be derived. Note that we define reachable for both non-terminals and terminals while generating is only defined for non-terminals.

This allows us to define a pruned context-free grammar, which is the CFG version of a trimmed WFSA.

Definition 4.2.14 (Pruned CFG)**.** *A CFG is **pruned** (or trimmed) if it has no useless non-terminals, i.e. all non-terminals are both reachable and generating. **Pruning** (or trimming) refers to the removal of useless non-terminals.*

4.2.3 Weighted Context-free Grammars

As we did with finite-state automata, we will augment the classic, unweighted context-free grammars with real-valued weights. We do that by associating with each rule $\mathrm{X} \to \boldsymbol{\alpha}$ a weight $\mathcal{W}(\mathrm{X} \to \boldsymbol{\alpha}) \in \mathbb{R}$.

Definition 4.2.15 (Weighted Context-free Grammar)**.** *A **real-weighted context-free grammar** is a 5-tuple $(\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P}, \mathcal{W})$ where Σ is an alphabet of terminal symbols, $\mathcal{N}$ is a non-empty set of non-terminal symbols with $\mathcal{N} \cap \Sigma = \varnothing$, $\mathrm{S} \in \mathcal{N}$ is the designated start non-terminal symbol, $\mathcal{P}$ is the set of production rules, and $\mathcal{W}$ a function $\mathcal{W} \colon \mathcal{P} \to \mathbb{R}$, assigning each production rule a real-valued weight.*

For notational brevity, we will denote rules $p \in \mathcal{P}$ as $p = \mathrm{X} \xrightarrow{w} \boldsymbol{\alpha}$ for $\mathrm{X} \in \mathcal{N}$, $\boldsymbol{\alpha} \in (\mathcal{N} \cup \Sigma)^*$ and $w = \mathcal{W}(\mathrm{X} \to \boldsymbol{\alpha}) \in \mathbb{R}$.

Example 4.2.7 (A simple weighted context-free grammar)**.** *Consider the grammar $\mathcal{G} = (\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P}, \mathcal{W})$ defined as follows:*

- $\Sigma = \{a, b\}$
- $\mathcal{N} = \{\mathrm{X}\}$
- $\mathrm{S} = \mathrm{X}$
- $\mathcal{P} = \{\mathrm{X} \to a\mathrm{X}b, \mathrm{X} \to \varepsilon\}$
- $\mathcal{W} = \left\{\mathrm{X} \to a\mathrm{X}b \mapsto \frac{1}{2}, \mathrm{X} \to \varepsilon \mapsto \frac{1}{2}\right\}$

This defines a simple weighting of the CFG from Example 4.2.2.

Weights assigned to productions by WFCGs can be arbitrary real numbers. Analogous to probabilistic WFSAs (Definition 4.1.17) describing locally normalized finite-state language models, we also define probabilistic WCFGs, where the weights of *applicable production rules* to any non-terminal form a probability distribution.

Definition 4.2.16. *A weighted context-free grammar* $\mathcal{G} = (\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P}, \mathcal{W})$ *is* ***probabilistic*** *if the weights of the productions of every non-terminal are non-negative and sum to* 1*, i.e., for all* $\mathrm{X} \in \mathcal{N}$*, it holds that*

$$\forall\ \mathrm{X} \to \boldsymbol{\alpha} \in \mathcal{P},\ \mathcal{W}(\mathrm{X} \to \boldsymbol{\alpha}) \geqslant 0 \tag{4.87}$$

and

$$\sum_{\mathrm{X} \to \boldsymbol{\alpha} \in \mathcal{P}} \mathcal{W}(\mathrm{X} \to \boldsymbol{\alpha}) = 1 \tag{4.88}$$

Intuitively, this means that all the production weights are non-negative and that, for any left side of a production rule X, the weights over all production rules $\mathrm{X} \to \boldsymbol{\alpha}$ sum to 1. The grammar from Example 4.2.7 is, therefore, also probabilistic.

Again analogously to the WFSA case, we say that a string $\boldsymbol{y}$ is in the language of WCFG $\mathcal{G}$ if there exists a derivation tree $\boldsymbol{d}$ in $\mathcal{G}$ containing only non-zero weights with yield $\mathsf{yield}(\boldsymbol{d}) = \boldsymbol{y}$.

Tree Weights, String Weights, and Allsums

In the case of regular languages, we discussed how individual strings are "produced" by paths in the automaton (in the sense that each path *yields* a string). As Example 4.2.4 showed, the structures that "produce" or yield strings in a context-free grammar are *trees*—those, therefore, play an analogous role in context-free grammars to paths in finite-state automata.

Just like we asked ourselves how to combine individual transition weights in a WFSA into weights of entire paths and later how to combine those into weights of strings, we now consider the questions of how to combine the weights of individual production rules into the weight of entire trees and later also individual strings. We start by giving a definition of the weight of a tree as the product over the weights of all the rules in the tree, i.e., as a *multiplicatively decomposable* function over the weights of its rules. As you can probably foresee, we will then define the weight of a string as the *sum* over all the trees which yield that string.

Definition 4.2.17. *The* ***weight of a derivation tree*** $\boldsymbol{d} \in \mathcal{D}_{\mathcal{G}}$ *defined by a WCFG* $\mathcal{G}$ *is*

$$\mathrm{w}(\boldsymbol{d}) = \prod_{(\mathrm{X} \to \boldsymbol{\alpha}) \in \boldsymbol{d}} \mathcal{W}(\mathrm{X} \to \boldsymbol{\alpha}). \tag{4.89}$$

The stringsum or the string acceptance weight of a particular string under a grammar is then defined as follows:

Definition 4.2.18. *The* ***stringsum*** $\mathcal{G}(\boldsymbol{y})$ *of a string* $\boldsymbol{y}$ *generated by a WCFG* $\mathcal{G}$ *is defined by*

$$\mathcal{G}(\boldsymbol{y}) = \sum_{\boldsymbol{d} \in \mathcal{D}_{\mathcal{G}}(\boldsymbol{y})} \mathrm{w}(\boldsymbol{d}) \tag{4.90}$$

$$= \sum_{\boldsymbol{d} \in \mathcal{D}_{\mathcal{G}}(\boldsymbol{y})} \prod_{(\mathrm{X} \to \boldsymbol{\alpha}) \in \boldsymbol{d}} \mathcal{W}(\mathrm{X} \to \boldsymbol{\alpha}) \tag{4.91}$$

Lastly, analogously to the allsum in WFSAs, an **allsum** is the sum of the weights of all the trees in a WCFG. We first define the allsum for symbols (non-terminals and terminals).

Definition 4.2.19 (Allsum)**.** *The **allsum** for a non-terminal* Y *in a grammar* $\mathcal{G}$ *is defined by*

$$Z(\mathcal{G}, \mathrm{Y}) = \sum_{\boldsymbol{d} \in \mathcal{D}_{\mathcal{G}}(\mathrm{Y})} \mathrm{w}(\boldsymbol{d}) \tag{4.92}$$

$$= \sum_{\boldsymbol{d} \in \mathcal{D}_{\mathcal{G}}(\mathrm{Y})} \prod_{(\mathrm{X} \to \boldsymbol{\alpha}) \in \boldsymbol{d}} \mathcal{W}(\mathrm{X} \to \boldsymbol{\alpha}) \tag{4.93}$$

The allsum for a terminal $a \in \Sigma \cup \{\varepsilon\}$ *is defined to be*

$$Z(a) \stackrel{\text{def}}{=} \mathbf{1}. \tag{4.94}$$

The allsum for a grammar is then simply the allsum for its start symbol.

Definition 4.2.20 (Allsum WCFG)**.** *The **allsum** of a weighted context-free grammar* $\mathcal{G} = (\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P}, \mathcal{W})$ *is*

$$Z(\mathcal{G}) = Z(\mathcal{G}, \mathrm{S}) \tag{4.95}$$

$$= \sum_{\boldsymbol{d} \in \mathcal{D}_{\mathcal{G}}(\mathrm{S})} \mathrm{w}(\boldsymbol{d}) \tag{4.96}$$

$$= \sum_{\boldsymbol{d} \in \mathcal{D}_{\mathcal{G}}(\mathrm{S})} \prod_{(\mathrm{X} \to \boldsymbol{\alpha}) \in \boldsymbol{d}} \mathcal{W}(\mathrm{X} \to \boldsymbol{\alpha}) \tag{4.97}$$

When the grammar $\mathcal{G}$ we refer to is clear from context, we will drop the subscript and write e.g. $Z(\mathrm{S})$.

Although we can in some cases compute the allsum of a WCFG in closed form, as we will see in the example below, we generally require some efficient algorithm to be able to do so.

Example 4.2.8 (Geometric Series as an Allsum)**.** *Consider the WCFG* $\mathcal{G} = (\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P}, \mathcal{W})$*, given by* $\mathcal{N} = \{\mathrm{X}\}$*,* $\Sigma = \{a\}$*,* $\mathrm{S} = \mathrm{X}$*, and the rules:*

$$\mathrm{X} \xrightarrow{1/2} a\,\mathrm{X}$$

$$\mathrm{X} \xrightarrow{1} \varepsilon$$

The language generated by $\mathcal{G}$ *is* $L(\mathcal{G}) = \{a^n \mid n \geqslant 0\}$*. Further note that this grammar is unambiguous – each string* $\boldsymbol{y} = a^m$*, for some* $m \geqslant 0$*, is associated with the derivation tree given by* $\underbrace{(\mathrm{X} \xrightarrow{1/3} a\,\mathrm{X}), \ldots, (\mathrm{X} \xrightarrow{1/3} a\,\mathrm{X})}_{m \text{ times}}, (\mathrm{X} \xrightarrow{1} \varepsilon)$*. Due*

to the multiplicative decomposition over the weights of the rules, the weight associated with each derivation tree $\boldsymbol{d}$ *will hence be*

$$\mathrm{w}(\boldsymbol{d}) = \left(\frac{1}{3}\right)^m \times 1 = \left(\frac{1}{3}\right)^m$$

Accordingly, we can compute the allsum of $\mathcal{G}$ *using the closed-form expression for geometric series:*

$$Z(\mathcal{G}) = \sum_{m=0}^{\infty} \left(\frac{1}{3}\right)^m = \frac{1}{1-1/3} = \frac{3}{2}$$

Just like we defined normalizable WFSAs, we also define normalizable WCFSs in terms of their allsum.

Definition 4.2.21 (Normalizable Weighted Context-free Grammar)**.** *A weighted context-free grammar* $\mathcal{G}$ *is* ***normalizable*** *if* $Z(\mathcal{G})$ *is finite, i.e.,* $Z(\mathcal{G}) < \infty$*.*

4.2.4 Context-free Language Models

This brings us to the definition of context-free language models.

Definition 4.2.22. *A language model* p_{LM} *is* ***context-free*** *if its weighted language equals the language of some weighted context-free grammar, i.e., if there exists a weighted context-free grammar* $\mathcal{G}$ *such that* $L(\mathcal{G}) = L(p_{LM})$*.*

Going the other way—defining string probabilities given a weighted context-free grammar—there are again two established ways of defining the probability of a string in its language.

String Probabilities in a Probabilistic Context-free Grammar

In a probabilistic CFG (cf. Definition 4.2.16), any production from a non-terminal $\mathrm{X} \in \mathcal{N}$ is associated with a probability. As the probabilities of continuing a derivation (and, therefore, a derivation tree) depend solely on the individual terminals (this is the core of *context-free* grammars!), it is intuitive to see those probabilities as conditional probabilities of the new symbols given the output generated so far. One can, therefore, define the probability of a path as the product of these individual "conditional" probabilities.

Definition 4.2.23 (Tree probability in a PCFG)**.** *We call the weight of a tree* $\boldsymbol{d} \in \mathcal{D}_{\mathcal{G}}$ *in a probabilistic CFG the* ***probability*** *of the tree* $\boldsymbol{d}$*.*

This alone is not enough to define the probability of any particular string $\boldsymbol{y} \in \Sigma^*$ since there might be multiple derivations of $\boldsymbol{y}$. Naturally, we define the probability of $\boldsymbol{y}$ as the sum of the individual trees that generate it:

Definition 4.2.24 (String probability in a PCFG)**.** *We call the stringsum of a string* $\boldsymbol{y} \in \Sigma^*$ *in a probabilistic CFG* $\mathcal{G}$ *the* ***probability*** *of the string* $\boldsymbol{y}$*:*

$$p_{\mathcal{G}}(\boldsymbol{y}) \stackrel{\text{def}}{=} \mathcal{G}(\boldsymbol{y}). \tag{4.98}$$

These definitions and their affordances mirror the ones in probabilistic finite-state automata (cf. §4.1.2): they again do not require any *normalization* and are therefore attractive as the summation over all possible strings is avoided. Again, the question of *tightness* of such models comes up: we explore it question in §4.2.5.

String Probabilities in a General Weighted Context-free Grammar

To define string probabilities in a general weighted CFG, we use the introduced notions of the stringsum and the allsum—we *normalize* the stringsum to define the globally normalized probability of a string $\boldsymbol{y}$ as the *proportion* of the total weight assigned to all strings that is assigned to $\boldsymbol{y}$.

Definition 4.2.25 (String probability in a WCFG)**.** *Let $\mathcal{G} = (\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P}, \mathcal{W})$ be a normalizable WCFG with non-negative weights. We define the probability of a string $\boldsymbol{y} \in \Sigma^*$ under $\mathcal{G}$ as*

$$p_{\mathcal{G}}(\boldsymbol{y}) \stackrel{\text{def}}{=} \frac{\mathcal{G}(\boldsymbol{y})}{Z(\mathcal{G})}. \tag{4.99}$$

Language Models Induced by a Weighted Context-free Grammar

With the notions of string probabilities in both probabilistic and general weighted CFGs, we can now define the language model induced by $\mathcal{G}$ as follows.

Definition 4.2.26 (A language model induced by a WCFG)**.** *Let $\mathcal{G} = (\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P}, \mathcal{W})$ be a WCFG. We define the **language model induced by** $\mathcal{G}$ as the following probability distribution over Σ^**

$$p_{LM\mathcal{G}}(\boldsymbol{y}) \stackrel{\text{def}}{=} p_{\mathcal{G}}(\boldsymbol{y}). \tag{4.100}$$

Again, it is easy to see that while global normalization requires the computation of the allsum, language models induced by weighted FSAs through Eq. (4.99) are *globally normalized* and thus always tight. The tightness of *probabilistic* WCFGs is discussed next, after which we investigate the relationship between globally- and locally-normalized context-free grammars.

4.2.5 Tightness of Context-free Language Models

Again, an advantage of globally normalized context-free language models (grammars) is that they are always tight, as the derivation trees are explicitly normalized with the global normalization constant such that they sum to 1 over the set of possible sentences.

In this section, we, therefore, consider the tightness of probabilistic context-free grammars. We follow the exposition from Booth and Thompson (1973). The proof requires the use of multiple new concepts, which we first introduce below.

Definition 4.2.27 (Generation level)**.** *We define the* ***level of a generation sequence*** *inductively as follows. The zeroth level $\boldsymbol{\gamma}_0$ of a generation sequence is defined as* S*. Then, for any $n > 0$, $\boldsymbol{\gamma}_n$ corresponds to the string is obtained by applying the applicable productions onto* all *nonterminals of $\boldsymbol{\gamma}_{n-1}$.*

Example 4.2.9. *Let $\mathcal{G} = (\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P})$ with $\Sigma = \{a, b\}$, $\mathcal{N} = \{\mathrm{S}, \mathrm{X}, \mathrm{Y}\}$, and $\mathcal{P} = \{\mathrm{S} \to a\,\mathrm{X}\,\mathrm{Y}, \mathrm{X} \to \mathrm{Y}\,\mathrm{X}, \mathrm{X} \to b\mathrm{Y}\,\mathrm{Y}, \mathrm{Y} \to a\,a\,\mathrm{Y}, \mathrm{Y} \to a\}$. Then the generation sequence of the string aabaaaaa would be*

$$
\begin{aligned}
\boldsymbol{\gamma}_0 &= \mathrm{S} && \textit{(definition)} \\
\boldsymbol{\gamma}_1 &= a\mathrm{XY} && \textit{(applying } \mathrm{S} \to a\,\mathrm{X}\,\mathrm{Y}\textit{)} \\
\boldsymbol{\gamma}_2 &= a\mathrm{YX}aa\mathrm{Y} && \textit{(applying } \mathrm{X} \to \mathrm{Y}\,\mathrm{X},\ \mathrm{Y} \to a\,a\,\mathrm{Y}\textit{)} \\
\boldsymbol{\gamma}_3 &= aab\mathrm{YY}aaaa\mathrm{Y} && \textit{(applying } \mathrm{Y} \to a,\ \mathrm{X} \to b\,\mathrm{Y}\,\mathrm{Y},\ \mathrm{X} \to a\,a\,\mathrm{Y}\textit{)} \\
\boldsymbol{\gamma}_3 &= aabaaaaaaa && \textit{(applying } \mathrm{Y} \to a,\ \mathrm{Y} \to a,\ \mathrm{Y} \to a\textit{)}
\end{aligned}
$$

We will also rely heavily on generating functions. A generating function is simply a way of *representing* an infinite sequence by encoding its elements as the coefficients of a *formal power series*. Unlike ordinary series such as the geometric power series from Example 4.2.8, a formal power series does not need to converge: in fact, at its core a generating function is not actually regarded as a *function*—its "variables" are indeterminate and they simply serve as "hooks" for the numbers in the sequence.

Definition 4.2.28 (Production generating function)**.** *Let $\mathcal{G} \stackrel{\text{def}}{=} (\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P}, \mathcal{W})$ be a PCFG and $N \stackrel{\text{def}}{=} |\mathcal{N}|$. For each $\mathrm{X}_n \in \mathcal{N}$, define its* ***production generating function*** *as*

$$
g\left(s_1, \ldots, s_N\right) \stackrel{\text{def}}{=} \sum_{\mathrm{X}_n \to \boldsymbol{\alpha}} \mathcal{W}\left(\mathrm{X}_n \to \boldsymbol{\alpha}\right) s_1^{\mathrm{r}_1(\boldsymbol{\alpha})} s_2^{\mathrm{r}_2(\boldsymbol{\alpha})} \cdot \cdots \cdot s_N^{\mathrm{r}_N(\boldsymbol{\alpha})}, \tag{4.101}
$$

where $\mathrm{r}_m\left(\boldsymbol{\alpha}\right)$ denotes the number of times the nonterminal $\mathrm{X}_m \in \mathcal{N}$ appears in $\boldsymbol{\alpha} \in (\Sigma \cup \mathcal{N})^*$.

Example 4.2.10. *Let $\mathcal{G} = (\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P})$ with $\Sigma = \{a, b\}$, $\mathcal{N} = \{\mathrm{S}, \mathrm{X}\}$, and $\mathcal{P} = \{\mathrm{S} \to a\,\mathrm{S}\,\mathrm{X}, \mathrm{S} \to b, \mathrm{X} \to a\mathrm{X}\,\mathrm{X}, \mathrm{X} \to aa\}$. Then*

$$
\begin{aligned}
g_1\left(s_1, s_2\right) &= \mathcal{W}\left(\mathrm{S} \to a\mathrm{SX}\right) s_1 s_2 + \mathcal{W}\left(\mathrm{S} \to b\right) \\
g_2\left(s_1, s_2\right) &= \mathcal{W}\left(\mathrm{X} \to a\mathrm{XX}\right) s_2^2 + \mathcal{W}\left(\mathrm{X} \to aa\right)
\end{aligned}
$$

Definition 4.2.29 (Generating function)**.** *The* ***generating function*** *of the l^{th} level is defined as*

$$
G_0\left(s_1, \ldots, s_N\right) \stackrel{\text{def}}{=} s_1 \tag{4.102}
$$

$$
G_1\left(s_1, \ldots, s_N\right) \stackrel{\text{def}}{=} g_1\left(s_1, \ldots, s_N\right) \tag{4.103}
$$

$$
G_l\left(s_1, \ldots, s_N\right) \stackrel{\text{def}}{=} G_{l-1}\left(g_1\left(s_1, \ldots, s_N\right), \ldots, g_N\left(s_1, \ldots, s_N\right)\right), \tag{4.104}
$$

that is, the l^{th}-level generating function is defined as the $l-1^{st}$-level generating function applied to production generating functions as arguments.

Example 4.2.11. *For the grammar from Example 4.2.10, we have*

$$\begin{aligned}
G_0(s_1, s_2) &= s_1 \\
G_1(s_1, s_2) &= g(s_1, s_2) = \mathcal{W}(\mathrm{S} \to a\mathrm{SX})\, s_1 s_2 + \mathcal{W}(\mathrm{S} \to b) \\
G_2(s_1, s_2) &= \mathcal{W}(\mathrm{S} \to a\mathrm{SX})\, [g_1(s_1, s_2)]\, [g_2(s_1, s_2)] + \mathcal{W}(\mathrm{S} \to b) \\
&= \mathcal{W}(\mathrm{S} \to a\mathrm{SX})^2\, \mathcal{W}(\mathrm{X} \to a\mathrm{XX})\, s_1 s_2^3 \\
&\quad + \mathcal{W}(\mathrm{S} \to a\mathrm{SX})^2\, \mathcal{W}(\mathrm{X} \to aa)\, s_1 s_2 \\
&\quad + \mathcal{W}(\mathrm{S} \to a\mathrm{SX})\, \mathcal{W}(\mathrm{S} \to b)\, \mathcal{W}(\mathrm{X} \to a\mathrm{XX})\, s_2^2 \\
&\quad + \mathcal{W}(\mathrm{S} \to a\mathrm{SX})\, \mathcal{W}(\mathrm{S} \to b)\, \mathcal{W}(\mathrm{X} \to aa) \\
&\quad + \mathcal{W}(\mathrm{S} \to b)
\end{aligned}$$

We can see that a generating function $G_l(s_1, \ldots, s_N)$ can be expressed as

$$G_l(s_1, \ldots, s_N) = D_l(s_1, \ldots, s_N) + C_l \tag{4.105}$$

where the polynomial $D_l(s_1, \ldots, s_N)$ does not contain any constant terms. It is easy to see that the constant C_l then corresponds to the probability of all strings that can be derived in l levels or fewer. This brings us to the following simple lemma.

Lemma 4.2.2. *A PCFG is tight if and only if*

$$\lim_{l \to \infty} C_l = 1. \tag{4.106}$$

Proof. Suppose that $\lim_{l\to\infty} C_l < 1$. This means that the generation process can enter a generation sequence that has a non-zero probability of not terminating—this corresponds exactly to it not being tight.

On the other hand, $\lim_{l\to\infty} C_l = 1$ implies that no such sequence exists, since the limit represents the probability of all strings that can be generated by derivations of a finite number of production rules. ■

The rest of the section considers necessary and sufficient conditions for Eq. (4.106) to hold. For this, we first define the first-moment matrix of a PCFG.

Definition 4.2.30. *Let $\mathcal{G} \stackrel{\text{def}}{=} (\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P}, \mathcal{W})$ be a PCFG. We define its **first-moment matrix** (its mean matrix) $\mathbf{E} \in \mathbb{R}^{N \times N}$ as*

$$E_{nm} \stackrel{\text{def}}{=} \left. \frac{\partial g_n(s_1, \ldots, s_N)}{\partial s_m} \right|_{s_1, \ldots, s_N = 1}. \tag{4.107}$$

Note that E_{nm} represents the *expected number of occurrences* of the non-terminal X_m in the set of sequences $\boldsymbol{\alpha}$ with $\mathrm{X}_n \Rightarrow_{\mathcal{G}} \boldsymbol{\alpha}$, i.e., the set of sequences X_n can be rewritten into:

$$E_{nm} = \sum_{\mathrm{X}_n \to \boldsymbol{\alpha}} \mathcal{W}(\mathrm{X}_n \to \boldsymbol{\alpha})\, \mathrm{r}_m(\boldsymbol{\alpha}). \tag{4.108}$$

The informal intuition behind this is the following: each of the terms $\mathcal{W}\left(\mathrm{X}_n \to \boldsymbol{\alpha}\right) s_1^{\mathrm{r}_1(\boldsymbol{\alpha})} s_2^{\mathrm{r}_2(\boldsymbol{\alpha})} \cdot \ldots \cdot s_N^{\mathrm{r}_N(\boldsymbol{\alpha})}$ in g_n contains the information about *how many times* any non-terminal X_m appears in the production rule $\mathrm{X}_n \to \boldsymbol{\alpha}$ as well as what the probability of "using" or applying that production rule to X_n is. Differentiating $\mathcal{W}\left(\mathrm{X}_n \to \boldsymbol{\alpha}\right) s_1^{\mathrm{r}_1(\boldsymbol{\alpha})} s_2^{\mathrm{r}_2(\boldsymbol{\alpha})} \cdot \ldots \cdot s_N^{\mathrm{r}_N(\boldsymbol{\alpha})}$ w.r.t. s_m then "moves" the coefficient r_m corresponding to the number of occurrences of X_m in $\mathrm{X}_n \to \boldsymbol{\alpha}$ in front of the term $\mathcal{W}\left(\mathrm{X}_n \to \boldsymbol{\alpha}\right) s_1^{\mathrm{r}_1(\boldsymbol{\alpha})} s_2^{\mathrm{r}_2(\boldsymbol{\alpha})} \cdot \ldots \cdot s_N^{\mathrm{r}_N(\boldsymbol{\alpha})}$ in g_n, effectively multiplying the probability of the occurrence of the rule with the number of terms X_m in the rule—this is exactly the expected number of occurrences of X_m for this particular rule, averaging over all possible rules that could be applied. Summing over all applicable production rules for X_n (which form a probability distribution) gives us the total expected number of occurrences of X_m. This brings us to the core theorem of this section characterizing the tightness of PCFGs.

Theorem 4.2.1 (Tightness of PCFGs)**.** *A PCFG is tight if $|\lambda_{max}| < 1$ and is non-tight if $|\lambda_{max}| > 1$, where λ_{max} is the eigenvalue of $\mathbf{E}$ with the largest absolute value.*

Proof. The coefficient of the term $s_1^{\mathrm{r}_1} s_2^{\mathrm{r}_2} \cdot \ldots \cdot s_N^{\mathrm{r}_N}$ in the generating function $G_l\left(s_1, \ldots, s_N\right)$ corresponds to the probability that there will be r_1 non-terminal symbols X_1, …, r_N non-terminal symbols X_N in the l^{th} level of the generation sequence. In particular, if the grammar is tight, this means that

$$\lim_{l \to \infty} G_l\left(s_1, \ldots, s_N\right) = \lim_{l \to \infty} \left[D_l\left(s_1, \ldots, s_N\right) + C_l\right] = 1. \tag{4.109}$$

This, however, is only true if

$$\lim_{l \to \infty} D_l\left(s_1, \ldots, s_N\right) = 0 \tag{4.110}$$

and this, in turn, can only be true if $\lim_{l \to \infty} \mathrm{r}_n = 0$ for all $n = 1, \ldots N$. The expected value of r_n at level l is

$$\bar{\mathrm{r}}_{l,n} = \left.\frac{\partial G_l\left(s_1, \ldots, s_N\right)}{\partial s_n}\right|_{s_1, \ldots, s_N = 1}. \tag{4.111}$$

Reasoning about this is similar to the intuition behind the first-moment matrix, with the difference that we are now considering the number of occurrences after a sequence of l applications. Denoting

$$\bar{\mathrm{r}}_l \stackrel{\text{def}}{=} \left[\bar{\mathrm{r}}_{l,1}, \ldots, \bar{\mathrm{r}}_{l,N}\right] \tag{4.112}$$

we have

$$\bar{\mathrm{r}}_l = \left[\sum_{j=1}^{N} \frac{\partial G_{l-1}\left(g_1\left(s_1, \ldots, s_N\right), \ldots, g_N\left(s_1, \ldots, s_N\right)\right)}{\partial g_j}\right. \tag{4.113}$$

$$\left. \cdot \frac{\partial g_j}{\partial s_n}\left(s_1, \ldots, s_N\right) \mid n = 1, \ldots, N\right]\Bigg|_{s_1, \ldots, s_N = 1} \tag{4.114}$$

$$= \bar{\mathrm{r}}_{l-1} \mathbf{E}. \tag{4.115}$$

Applying this relationship repeatedly, we get

$$\bar{\mathrm{r}}_l = \bar{\mathrm{r}}_0 \mathbf{E}^l = [1, 0, \ldots, 0]\, \mathbf{E}^l, \tag{4.116}$$

meaning that

$$\lim_{l \to \infty} \bar{\mathrm{r}}_l = \mathbf{0} \text{ iff } \lim_{l \to \infty} \mathbf{E}_l = \mathbf{0}. \tag{4.117}$$

The matrix $\mathbf{E}$ satisfies this condition if $|\lambda_{\max}| < 1$. On the other hand, if $|\lambda_{\max}| > 1$, the limit diverges. ■

Note that the theorem does not say anything about the case when $|\lambda_{\max}| = 1$.

We conclude the subsection by noting that, interestingly, weighted context-free grammars *trained* on data with maximum likelihood are *always* tight (Chi and Geman, 1998; Chi, 1999). This is not the case for some models we consider later, e.g., recurrent neural networks (cf. §5.1.2).

4.2.6 Normalizing Weighted Context-free Grammars

Having investigated probabilistic context-free grammars in terms of their tightness, we now turn out attention to general weighted context-free grammars, which define string probabilities using global normalization (cf. Eq. (4.99)). To be able to compute these probabilities, require a way to compute the normalizing constant $Z(\mathcal{G})$ and the stringsum $\mathcal{G}(\boldsymbol{y})$. In the section on finite-state automata, we explicitly presented an algorithm for computing the normalizing constant $Z(\mathcal{A})$. The derivation of a general allsum algorithm for weighted context-free grammars, on the other hand, is more involved and beyond the scope of this course.[27] Here, we simply assert that there are ways of computing the quantities in Eq. (4.99) and only consider the following result:

Theorem 4.2.2 (PCFGs and WCFGs are equally expressive (Smith and Johnson, 2007))**.** *Normalizable weighted context-free grammars with non-negative weights and tight probabilistic context-free grammars are equally expressive.*

Proof. To prove the theorem, we have to show that any WCFG can be written as a PCFG and vice versa.[28]

⇐ Since any tight probabilistic context-free grammar is simply a WCFG with $Z(\mathcal{G}) = 1$, this holds trivially.

⇒ We now show that, for any WCFG, there exists a PCFG encoding the same language model. Let $\mathcal{G}_G = (\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P}, \mathcal{W})$ be a pruned WCFG that encodes a distribution over Σ^* using Eq. (4.99). We now construct a tight probabilistic context-free grammar $\mathcal{G}_L = (\Sigma, \mathcal{N}, \mathrm{S}, \mathcal{P}, \mathcal{W}_L)$ whose language is identical. Notice that all components of the grammar remain identical apart from the weighting function. This means that the derivations of the strings in the grammars remain the same (i.e., $\mathcal{D}_{\mathcal{G}_G} = \mathcal{D}_{\mathcal{G}_L}$)—only the weights of the derivations change, as

[27] The allsums of individual non-terminals can be expressed as solutions to a nonlinear set of equations. Again, the interested reader should have a look at the Advanced Formal Language Theory course.

[28] Again, by "written as", we mean that the weighted language is the same.

we detail next. We define the production weights of the probabilistic CFG as follows.

$$\mathcal{W}_{\mathcal{G}_L}(\mathrm{X} \to \boldsymbol{\alpha}) \stackrel{\text{def}}{=} \frac{\mathcal{W}(\mathrm{X} \to \boldsymbol{\alpha}) \prod_{\mathrm{Y} \in \boldsymbol{\alpha}} Z(\mathcal{G}, \mathrm{Y})}{Z(\mathcal{G}, \mathrm{X})} \tag{4.118}$$

Remember that $Z(a) = 1$ for $a \in \Sigma$. Note that the assumption that $\mathcal{G}$ is pruned means that all the quantities in the denominators are non-zero.

It is easy to see that the weight defined this way are non-negative due to the non-negativity of $\mathcal{G}$'s weights. Furthermore, the weights of all production rules for any non-terminal $\mathrm{X} \in \mathcal{N}$ sum to 1, as by the definitions of $\mathcal{W}_{\mathcal{G}_L}$ and $Z(\mathcal{G}, \mathrm{X})$ we have

$$\sum_{\mathrm{X} \to \boldsymbol{\alpha}} \mathcal{W}_{\mathcal{G}_L}(\mathrm{X} \to \boldsymbol{\alpha}) = \sum_{\mathrm{X} \to \boldsymbol{\alpha}} \frac{\mathcal{W}(\mathrm{X} \to \boldsymbol{\alpha}) \prod_{\mathrm{Y} \in \boldsymbol{\alpha}} Z(\mathcal{G}, \mathrm{Y})}{Z(\mathcal{G}, \mathrm{X})} \tag{4.119}$$

$$= \frac{1}{Z(\mathcal{G}, \mathrm{X})} \sum_{\mathrm{X} \to \boldsymbol{\alpha}} \mathcal{W}(\mathrm{X} \to \boldsymbol{\alpha}) \prod_{\mathrm{Y} \in \boldsymbol{\alpha}} Z(\mathcal{G}, \mathrm{Y}) \tag{4.120}$$

$$= \frac{1}{Z(\mathcal{G}, \mathrm{X})} Z(\mathcal{G}, \mathrm{X}) \tag{4.121}$$

$$= 1 \tag{4.122}$$

We now have to show that the probabilities assigned by these two grammars match. We will do that by showing that the probabilities assigned to individual *derivations* match, implying that stringsums match as well. The probability of a derivation is defined analogously to a probability of a string, i.e., $p_{\mathcal{G}}(\boldsymbol{d}) = \frac{\mathrm{w}(\boldsymbol{d})}{Z(\mathcal{G})}$ (where $Z(\mathcal{G}) = 1$ for tight probabilistic grammars). Let then $\boldsymbol{d} \in \mathcal{D}_{\mathcal{G}} = \mathcal{D}_{\mathcal{G}_L}$. Then

$$p_{\mathcal{G}_L}(\boldsymbol{d}) = \prod_{\mathrm{X} \to \boldsymbol{\alpha} \in \boldsymbol{d}} \mathcal{W}_{\mathcal{G}_L}(\mathrm{X} \to \boldsymbol{\alpha}) \tag{4.123}$$

$$= \prod_{\mathrm{X} \to \boldsymbol{\alpha} \in \boldsymbol{d}} \frac{\mathcal{W}(\mathrm{X} \to \boldsymbol{\alpha}) \prod_{\mathrm{Y} \in \boldsymbol{\alpha}} Z(\mathcal{G}, \mathrm{Y})}{Z(\mathcal{G}, \mathrm{X})} \quad \text{(definition of } \mathcal{W}_{\mathcal{G}_L}\text{)}. \tag{4.124}$$

Notice that by multiplying over the internal nodes of the derivation tree, Eq. (4.123) includes the non-terminal allsum of each *internal* (non-root and non-leaf) non-terminal in the derivation *twice*: once as a parent of a production in the denominator, and once as a child in the numerator. These terms, therefore, all cancel out in the product. The only terms which are left are the allsums of the leaf nodes—the terminals—which are 1, and the allsum of the root node—S—which equals $Z(\mathcal{G}_G)$ and the weights of the individual productions, which multiply into the weight assigned to $\boldsymbol{d}$ by the original grammar $\mathcal{G}_G$. This means that

$$p_{\mathcal{G}_L}(\boldsymbol{d}) = \frac{1}{Z(\mathcal{G}, \mathrm{X})} \prod_{\mathrm{X} \to \boldsymbol{\alpha} \in \boldsymbol{d}} \mathcal{W}(\mathrm{X} \to \boldsymbol{\alpha}) = \frac{1}{Z(\mathcal{G}, \mathrm{X})} \mathrm{w}(\boldsymbol{d}) = p_{\mathcal{G}_G}(\boldsymbol{d}), \tag{4.125}$$

finishing the proof. ■

This means that the classes of probabilistic and weighted context-free grammars are in fact *equally expressive*. In other words, this result is analogous to Theorem 4.1.1 in WFSAs: it shows that in the context of context-free language models, the locally normalized version of a globally-normalized model is *also* context-free.

4.2.7 Pushdown Automata

We presented context-free grammars as a formalism for specifying and representing context-free languages. Many algorithms for processing context-free languages, for example, the allsum algorithms and their generalizations, can also be directly applied to context-free grammars. However, it is also convenient to talk about processing context-free languages in terms of computational models in the form of automata, i.e., the *recognizer* of the language.[29] As we mentioned, the types of automata we considered so far, (weighted) finite-state automata, can only recognize regular languages. To recognize context-free languages, we must therefore extend finite-state automata.[30] We do that by introducing **pushdown automata** (PDA), a more general and more expressive type of automata.

Single-stack Pushdown Automata

Pushdown automata augment finite-state automata by implementing an additional *stack* memory structure for storing *arbitrarily long* strings from a designated alphabet, which allows them to work with unbounded memory effectively. Abstractly, this unbounded memory is the only difference to finite-state automata. However, the definition looks slightly different:

Definition 4.2.31. *A **pushdown automaton** (PDA) is a tuple* $\mathcal{P} \stackrel{\text{def}}{=} (Q, \Sigma, \Gamma, \delta, (q_I, \boldsymbol{\gamma}_I), (q_F, \boldsymbol{\gamma}_F))$*, where:*

- Q *is a finite set of states;*
- Σ *is a finite set of input symbols called the input alphabet;*
- Γ *is a finite set of stack symbols called the stack alphabet;*
- $\delta \subseteq Q \times \Gamma^* \times (\Sigma \cup \{\varepsilon\}) \times Q \times \Gamma^*$ *is a multiset representing the transition function;*
- $(q_I, \boldsymbol{\gamma}_I)$ *is called the initial configuration and* $(q_F, \boldsymbol{\gamma}_F)$ *is called the final configuration, where* $q_I, q_F \in Q$ *and* $\boldsymbol{\gamma}_I, \boldsymbol{\gamma}_F \in \Gamma^*$ *.*

[29] This relationship between a formalism specifying how to *generate* (i.e., a grammar) and a model of *recognizing* a language can be seen in multiple levels of the hierarchy of formal languages. In the case of context-free languages, the former are context-free grammars, while the latter are pushdown automata discussed in this subsection. Regular languages as introduced in the previous section, however, are simply defined in terms of their recognizers—finite-state automata.

[30] Formally, we would of course have to prove that finite-state automata cannot model context-free languages. This can be done with the so-called pumping lemma, which are outside the scope of this class.

The initial and final configurations in pushdown play analogous roles to the sets of initial and final sets of finite-state automata. Compared to the latter, they also allow for different starting configurations of the stack coupled with each possible initial or final state.

Stacks are represented as strings over Γ, from bottom to top. Thus, in the stack $\boldsymbol{\gamma} = X_1 X_2 \cdots X_n$, the symbol X_1 is at the bottom of the stack, while X_n is at the top. $\boldsymbol{\gamma} = \varnothing$ denoses the empty stack.

Definition 4.2.32. *A **configuration** of a PDA is a pair $(q, \boldsymbol{\gamma})$, where $q \in Q$ is the current state and $\boldsymbol{\gamma} \in \Gamma^*$ is the current contents of the stack.*

The initial and final configurations of a PDA are examples of configurations; it is possible to generalize the initial and final stacks to (say) regular expressions over Γ, but the above definition suffices for our purposes.

A PDA moves from configuration to configuration by following transitions of the form $q \xrightarrow{a, \boldsymbol{\gamma}_1 \to \boldsymbol{\gamma}_2} r$, which represents a move from the state q to state r, while popping the sequence of symbols $\boldsymbol{\gamma}_1 \in \Gamma^*$ from the top of the stack and pushing the sequence $\boldsymbol{\gamma}_2 \in \Gamma^*$. The PDA transition function therefore not only depends on the current state q and input symbol a, but also on some *finite* sequence of symbols on the *top* of the stack. The stack hence determines the behavior of the automaton, and since the set of possible configurations of the stack is infinite, the set of configurations of the automaton is infinite, in contrast to finite-state automata.

To describe how pushdown automata process strings, we introduce the concepts of scanning and runs.

Definition 4.2.33. *We say that $\tau = (p, \boldsymbol{\gamma}_1, a, q, \boldsymbol{\gamma}_2) \in \delta$ **scans** a, and if $a \neq \varepsilon$, we call τ **scanning**; otherwise, we call it **non-scanning**.*

Definition 4.2.34. *If $(q_1, \boldsymbol{\gamma}\boldsymbol{\gamma}_1)$ and $(q_2, \boldsymbol{\gamma}\boldsymbol{\gamma}_2)$ are configurations, and τ is a transition $q_1 \xrightarrow{a, \boldsymbol{\gamma}_1 \to \boldsymbol{\gamma}_2} q_2$, we write $(q_1, \boldsymbol{\gamma}\boldsymbol{\gamma}_1) \Rightarrow_\tau (q_2, \boldsymbol{\gamma}\boldsymbol{\gamma}_2)$.*

Since the behavior of a pushdown automaton does not only depend on the states encountered by it but also on the content of the stack, we generalize the notion of a path to include the *configuration* of the automaton. This is called a run.

Definition 4.2.35. *A **run** of a PDA $\mathcal{P}$ is a sequence of configurations and transitions*

$$\boldsymbol{\pi} = (q_0, \boldsymbol{\gamma}_0), \tau_1, (q_1, \boldsymbol{\gamma}_1), \ldots, \tau_n, (q_N, \boldsymbol{\gamma}_N)$$

where, for $n = 1, \ldots, N$, we have $(q_{n-1}, \boldsymbol{\gamma}_{n-1}) \Rightarrow_{\tau_n} (q_n, \boldsymbol{\gamma}_n)$.[31] *A run is called **accepting** if $(q_0, \boldsymbol{\gamma}_0)$ is the initial configuration and $(q_N, \boldsymbol{\gamma}_N)$ is the final configuration. If, for $n = 1, \ldots, N$, τ_n scans a_n, then we say that $\boldsymbol{\pi}$ scans the string $a_1 \cdots a_N$. We write $\Pi(\mathcal{P}, \boldsymbol{y})$ for the set of runs that scan $\boldsymbol{y}$ and $\Pi(\mathcal{P})$ for the set of all accepting runs of $\mathcal{P}$.*

[31] Sometimes it will be convenient to treat $\boldsymbol{\pi}$ as a sequence of only configurations or only transitions.

Definition 4.2.36. *We say that the PDA $\mathcal{P}$ **recognizes** the string $\boldsymbol{y}$ if $\Pi(\mathcal{P}, \boldsymbol{y}) \neq \varnothing$, i.e., if there exists an accepting run with the yield $\boldsymbol{y}$. The set of all strings recognized by $\mathcal{P}$ is the **language recognized by** $\mathcal{P}$, which we denote by $L(\mathcal{P})$, i.e.,*

$$L(\mathcal{P}) \stackrel{\text{def}}{=} \{\boldsymbol{y} \mid \Pi(\mathcal{P}, \boldsymbol{y}) \neq \varnothing\}. \tag{4.126}$$

Example 4.2.12. *Fig. 4.8 shows an example of a pushdown automaton $\mathcal{P}$ accepting the language $L(\mathcal{P}) = \{a^n b^n \mid n \in \mathbb{N}\}$. $\left(1 \xrightarrow{a,\varepsilon\to\mathrm{X}} 1, 1 \xrightarrow{\varepsilon,\varepsilon\to\varepsilon} 2\right)$ is a run of $\mathcal{P}$; $\left(1 \xrightarrow{a,\varepsilon\to\mathrm{X}} 1, 1 \xrightarrow{\varepsilon,\varepsilon\to\varepsilon} 2, 2 \xrightarrow{b,\mathrm{X}\to\varepsilon} 2\right)$ is an* accepting *run of $\mathcal{P}$.*

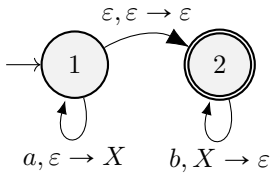


Figure 4.8: The PDA that accepts the language $\{a^n b^n \mid n \in \mathbb{N}\}$.

Lastly, we define deterministic pushdown automata, analogously to their finite-state version (Definition 4.1.3).

Definition 4.2.37. *A PDA $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, (q_I, \boldsymbol{\gamma}_I), (q_F, \boldsymbol{\gamma}_F))$ is **deterministic** if*

- *it does not have any ε-transitions;*
- *for every configuration $(q, \boldsymbol{\gamma})$ with $\boldsymbol{\gamma} \in \Gamma^*$ and $q \in Q$ and every $a \in \Sigma$, there is at most one $q' \in Q$ such that $q \xrightarrow{a,\circ\to\boldsymbol{\gamma}'} q \in \delta$.*

*Otherwise, $\mathcal{P}$ is **non-deterministic**.*

Importantly, *not all* context-free languages can be recognized by deterministic pushdown automata. That is, in contrast to finite-state automata, where deterministic machines are just as powerful as non-deterministic ones (at least in the unweighted case—interestingly, some weighted non-deterministic FSAs cannot be determinized), non-deterministic pushdown automata are more expressive than deterministic ones. Specifically, as stated in Theorem 4.2.3, non-deterministic pushdown automata recognize exactly context-free languages, while deterministic pushdown automata only recognize a subset of them (Sipser, 2013).

Weighted Pushdown Automata

Analogously to the finite-state case, and the case of context-free grammars, we now also extend the definition of a pushdown automaton to the weighted case. The formal definition is:

Definition 4.2.38. *A **real-weighted pushdown automaton** (WPDA) is a tuple $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, (q_I, \boldsymbol{\gamma}_I), (q_F, \boldsymbol{\gamma}_F))$, where:*

- *Q is a finite set of states;*
- *Σ is a finite set of input symbols called the input alphabet;*
- *Γ is a finite set of stack symbols called the stack alphabet;*
- *$\delta \subseteq Q \times \Gamma^* \times (\Sigma \cup \{\varepsilon\}) \times Q \times \Gamma^* \times \mathbb{R}$ is a multi-set representing the transition weighting function;*
- *$(q_I, \boldsymbol{\gamma}_I)$ is called the initial configuration and $(q_F, \boldsymbol{\gamma}_F)$ is called the final configuration, where $q_I, q_F \in Q$ and $\boldsymbol{\gamma}_I, \boldsymbol{\gamma}_F \in \Gamma^*$.*

As you can see, the only difference between the weighted and the unweighted case is the transition function, which in the weighted case weights the individual transitions instead of specifying the set of possible target configurations.

As with WFSAs (Definition 4.1.17) and WCFGs (**??**), we now define probabilistic WPDAs. This definition, however, is a bit more subtle. Notice that the transition weighting "function" δ in a WPDA is crucially still a *finite*—there is only a finite number of actions we can ever do. Similarly, when defining a *probabilistic* PDA, we have to limit ourselves to a finite number of configurations over which we define probability distributions over the next possible actions. It turns out that the **stack language**, i.e., the set of strings in Γ^* which can at any point be written on the stack, is *regular*.[32] Moreover, regular languages can be defined in terms of an equivalence relation on the set of strings with a finite number of equivalence classes (Nerode and Sauer, 1957). Therefore, a very natural way to limit ourselves to a finite set of contexts for defining probability distributions in a probabilistic pushdown automaton is in terms of a general equivalence relation. Recall the definition of an equivalence relation.

Definition 4.2.39 (Equivalence relation)**.** *Let $\mathcal{A}$ be a set. A relation $R \subseteq \mathcal{A} \times \mathcal{A}$ is **equivalent** (an equivalent relation) if it satisfies the following three properties:*

1. ***Reflexivity**: $(a, a) \in R$ for all $a \in \mathcal{A}$.*
2. ***Symmetry**: $(a, b) \in R \implies (b, a) \in R$ for all $a, b \in \mathcal{A}$.*
3. ***Transitivity**: $(a, b), (b, c) \in R \implies (a, c) \in R$ for all $a, b, c \in \mathcal{A}$.*

Definition 4.2.40 (Partition)**.** *Let $\mathcal{A}$ be a set. The family of sets $\{\mathcal{C}_1, \ldots, \mathcal{C}_N\}$ with $\mathcal{C}_n \subseteq \mathcal{A}$ is a **partition** of $\mathcal{A}$ if*

1. *$\mathcal{C}_n \neq \varnothing$ for all n,*
2. *$\mathcal{C}_n \cap \mathcal{C}_m = \varnothing$ for $n \neq m$, and*
3. *$\bigcup_{n=1}^{N} \mathcal{C}_n = \mathcal{A}$.*

[32] See Theorem 5.3 in Autebert et al. (1997).

Definition 4.2.41 (Partition, Equivalence class)**.** *An equivalence relation R on $\mathcal{A}$ partitions $\mathcal{A}$ into its **equivalence classes**—that is, into sets $\mathcal{C}_1, \ldots, \mathcal{C}_N$ where each $a \in \mathcal{A}$ belongs into exactly one $\mathcal{C}_n$.*

We denote the equivalence class of $a \in \mathcal{A}$ as $[a]$.

We finally define a probabilistic pushdown automaton given an equivalence relation as follows.

Definition 4.2.42. *Let R be an equivalence relation over Γ^* with a finite number of equivalence classes and $\{[\boldsymbol{\gamma}] \mid \boldsymbol{\gamma} \in \Gamma^*\}$ the set of its equivalence classes. A WPDA $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, (q_I, \boldsymbol{\gamma}_I), (q_F, \boldsymbol{\gamma}_F))$ is **probabilistic** if for any configuration $(q, \boldsymbol{\gamma})$ it holds that*

$$\forall\ q \xrightarrow{a, [\boldsymbol{\gamma}] \to [\boldsymbol{\gamma}']/w} r \in \delta :\ w \geqslant 0 \tag{4.127}$$

and

$$\sum_{q \xrightarrow{a, [\boldsymbol{\gamma}] \to [\boldsymbol{\gamma}']/w} r} w = 1. \tag{4.128}$$

Definition 4.2.43. *If $(q_1, \boldsymbol{\gamma}\boldsymbol{\gamma}_1)$ and $(q_2, \boldsymbol{\gamma}\boldsymbol{\gamma}_2)$ are configurations, and τ is a transition $q_1 \xrightarrow{a, \boldsymbol{\gamma}_1 \to \boldsymbol{\gamma}_2/w} q_2$ with $w \neq 0$, we write $(q_1, \boldsymbol{\gamma}\boldsymbol{\gamma}_1) \Rightarrow_\tau (q_2, \boldsymbol{\gamma}\boldsymbol{\gamma}_2)$.*

Definition 4.2.44. *If $\delta(p, \boldsymbol{\gamma}_1, a, q, \boldsymbol{\gamma}_2) = w$, then we usually write*

$$\delta(p, \boldsymbol{\gamma}_1 \xrightarrow{a} q, \boldsymbol{\gamma}_2) = w \tag{4.129}$$

or that δ has transition $(q \xrightarrow{a, \boldsymbol{\gamma}_1 \to \boldsymbol{\gamma}_2/w} p)$. We sometimes let τ stand for a transition, and we define $\delta(\tau) = w$.

And again, just like we combined the weights of individual transitions into the weights of paths in WFSAs, and we combined the weights of production rules into the weights of the trees in WCFGs, we now multiplicatively combine the weights of individual transitions in a run to define the weight of a run in a WPDA:

Definition 4.2.45. *The **weight** $\mathrm{w}(\boldsymbol{\pi})$ of a run*

$$\boldsymbol{\pi} = (q_0, \boldsymbol{\gamma}_0), \tau_1, (q_1, \boldsymbol{\gamma}_1), \ldots, \tau_N, (q_N, \boldsymbol{\gamma}_N)$$

is the multiplication of the transition weights, i.e.,

$$\mathrm{w}(\boldsymbol{\pi}) \stackrel{\text{def}}{=} \prod_{n=1}^{N} \delta(\tau_n) \tag{4.130}$$

Analogously to a stringsum in WFSAs, we define the stringsum for a string $\boldsymbol{y}$ in a WPDA $\mathcal{P}$ as the sum over the weights of all runs scanning $\boldsymbol{y}$.

Definition 4.2.46. *Let $\mathcal{P}$ be a WPDA and $\boldsymbol{y} \in \Sigma^*$ a string. The **stringsum** for $\boldsymbol{y}$ in $\mathcal{P}$ is defined as*

$$\mathcal{P}(\boldsymbol{y}) \stackrel{\text{def}}{=} \sum_{\boldsymbol{\pi} \in \Pi(\mathcal{P}, \boldsymbol{y})} \mathrm{w}(\boldsymbol{\pi}) \tag{4.131}$$

Definition 4.2.47. *We say that the PDA $\mathcal{P}$ **recognizes** the string $\boldsymbol{y}$ with the weight $\mathcal{P}(\boldsymbol{y})$.*

With this, we can define the weighted language defined by a WPDA.

Definition 4.2.48. *Let $\mathcal{P}$ be a WPDA. The **(weighted) language** $L(\mathcal{P})$ of $\mathcal{P}$ is defined as*

$$L(\mathcal{P}) \stackrel{\text{def}}{=} \{(\boldsymbol{y}, \mathcal{P}(\boldsymbol{y})) \mid \boldsymbol{y} \in \Sigma^*\} \tag{4.132}$$

Finally, we also define the WPDA allsum and normalizable WPDAs.

Definition 4.2.49 (WPDA Allsum)**.** *The **allsum** of a WPDA $\mathcal{P}$ is defined as*

$$Z(\mathcal{P}) \stackrel{\text{def}}{=} \sum_{\boldsymbol{\pi} \in \Pi(\mathcal{P})} \mathrm{w}(\boldsymbol{\pi}) \tag{4.133}$$

Definition 4.2.50. *A WPDA $\mathcal{P}$ is **normalizable** if $Z(\mathcal{P})$ is finite, i.e., if $Z(\mathcal{P}) < \infty$.*

Relationship to Context-free Grammars

We motivated the introduction of pushdown automata as a means of recognizing context-free languages. However, this correspondence is not obvious from the definition! Indeed, the equivalence of the expressive power of context-free grammars and pushdown automata is a classic result in formal language theory, and it is summarised by the theorem below:

Theorem 4.2.3. *A language is context-free if and only if some pushdown automaton recognizes it.*

Proof. See Theorem 2.20 in Sipser (2013). ■

This result extends to the probabilistic case.

Theorem 4.2.4. *A language is generated by a probabilistic context-free grammar if and only if some probabilistic pushdown automaton recognizes it.*

Proof. See Theorems 3 and 7 in Abney et al. (1999). ■

Lastly, analogously to how Theorem 4.2.2 showed that weighted context-free grammars are equally expressive as probabilistic context-free grammars, the following theorem asserts the same about pushdown automata:

Theorem 4.2.5. *Any globally normalized weighted pushdown automaton can be locally normalized. More precisely, this means the following. Let $\mathcal{P}$ be a weighted pushdown automaton. Then, there exists a probabilistic pushdown automaton $\mathcal{P}_p$ such that*

$$\mathcal{P}_p(\boldsymbol{y}) = \frac{\mathcal{P}(\boldsymbol{y})}{Z(\mathcal{P})} \tag{4.134}$$

for all $\boldsymbol{y} \in \Sigma^*$.

Proof. The proof is not straightforward: it can be shown that one cannot simply convert an arbitrary weighted pushdown automaton into a locally-normalized one directly. Rather, the construction of the latter goes through their *context-free grammars*: given a WPDA $\mathcal{P}$, one first constructs the WCFG equivalent to $\mathcal{P}$, and then converts that to a locally normalized one (cf. Theorem 4.2.2), i.e., a PCFG. Then, this PCFG can be converted to a structurally quite different probabilistic pushdown automaton $\mathcal{P}_p$, which nevertheless results in the language we require (Eq. (4.134)).

This construction is described in more detail in Abney et al. (1999) and Butoi et al. (2022). ■

4.2.8 Pushdown Language Models

We can now define pushdown language models, the title of this section.

Definition 4.2.51. *A **pushdown language model** is a language model whose weighted language equals the language of some weighted pushdown automaton, i.e., if there exists a weighted pushdown automaton $\mathcal{P}$ such that $L(\mathcal{P}) = L(p_{LM})$.*

Similarly, pushdown automata also induce language models.

Definition 4.2.52. *Let $\mathcal{P}$ be a weighted pushdown automaton. We define the **language model induced by** $\mathcal{P}$ as the probability distribution induced by the probability mass function*

$$p_{LM\mathcal{P}}(\boldsymbol{y}) \stackrel{\text{def}}{=} \frac{\mathcal{P}(\boldsymbol{y})}{Z(\mathcal{P})}, \tag{4.135}$$

for any $\boldsymbol{y} \in \Sigma^*$.

You might wonder why we specifically define pushdown language models and models induced by them if WPDAs are equivalent to WCFGs (cf. §4.2.7). In that sense, a language model is context-free if and only if it is a pushdown language model. However, this holds only for single-stack pushdown automata which we have discussed so far. We make this explicit distinction of pushdown language models with an eye to the next section, in which we introduce *multi-stack* WPDAs. Those are, as it turns out, much more powerful (expressive) than context-free grammars. We will, however, reuse this definition of a pushdown language model for those more powerful machines.

4.2.9 Multi-stack Pushdown Automata

We now consider an extension of (weighted) pushdown automata, namely, machines that employ *multiple* stacks. While this might not seem like an important distinction, we will see shortly that this augmentation results in a big difference in the expressiveness of the framework!

Definition 4.2.53. *A **two-stack pushdown automaton** (2-PDA) is a tuple* $\mathcal{P} = (\Sigma, Q, \Gamma_1, \Gamma_2, \delta, (q_I, \boldsymbol{\gamma}_{I1}, \boldsymbol{\gamma}_{I2}), (q_F, \boldsymbol{\gamma}_{F1}, \boldsymbol{\gamma}_{F2}))$*, where:*

- Σ *is a finite set of input symbols called the input alphabet;*
- Q *is a finite set of states;*
- Γ_1 *and* Γ_2 *are finite sets of stack symbols called the stack alphabets;*
- $\delta \subseteq Q \times \Gamma_1^* \times \Gamma_2^* \times (\Sigma \cup \{\varepsilon\}) \times Q \times \Gamma_1^* \times \Gamma_2^*$ *is a multiset representing the transition function;*
- $(q_I, \boldsymbol{\gamma}_{I1}, \boldsymbol{\gamma}_{I2})$ *is called the initial configuration and* $(q_F, \boldsymbol{\gamma}_{F1}, \boldsymbol{\gamma}_{F2})$ *is called the final configuration, where* $q_I, q_F \in Q$*,* $\boldsymbol{\gamma}_{I1}, \boldsymbol{\gamma}_{F1} \in \Gamma_1^*$*, and* $\boldsymbol{\gamma}_{I2}, \boldsymbol{\gamma}_{F2} \in \Gamma_2^*$*.*

Note that we could more generally define a k-stack PDA by including k stacks in the definition, but the restriction to two stacks will be sufficient for our needs, as we will see in the next subsection. The transition function now depends on the values stored in *both* of the stacks. The definitions of the configuration and run of a two-stack PDA are analogous to the single-stack variant, with the addition of the two stacks. We again extend this definition to the weighted and the probabilistic case.

Definition 4.2.54. *A **two-stack real-weighted pushdown automaton** (2-WPDA) is a tuple* $\mathcal{P} = (\Sigma, Q, \Gamma_1, \Gamma_2, \delta, (q_I, \boldsymbol{\gamma}_{I1}, \boldsymbol{\gamma}_{I2}), (q_F, \boldsymbol{\gamma}_{F1}, \boldsymbol{\gamma}_{F2}))$*, where:*

- Σ *is a finite set of input symbols called the input alphabet;*
- Q *is a finite set of states;*
- Γ_1 *and* Γ_2 *are finite sets of stack symbols called the stack alphabets;*
- $\delta \subseteq Q \times \Gamma_1^* \times \Gamma_2^* \times (\Sigma \cup \{\varepsilon\}) \times Q \times \Gamma_1^* \times \Gamma_2^* \times \mathbb{R}$ *is a multiset representing the transition weighting function;*
- $(q_I, \boldsymbol{\gamma}_{I1}, \boldsymbol{\gamma}_{I2})$ *is called the initial configuration and* $(q_F, \boldsymbol{\gamma}_{F1}, \boldsymbol{\gamma}_{F2})$ *is called the final configuration, where* $q_I, q_F \in Q$*,* $\boldsymbol{\gamma}_{I1}, \boldsymbol{\gamma}_{F1} \in \Gamma_1^*$*, and* $\boldsymbol{\gamma}_{I2}, \boldsymbol{\gamma}_{F2} \in \Gamma_2^*$*.*

And lastly, we define probabilistic two-stack PDAs:

Definition 4.2.55. *Let* R_1 *and* R_2 *be equivalence relations over* Γ_1^* *and* Γ_2^**, respectively, both with a finite number of equivalence classes. A 2-WPDA* $\mathcal{P} = (Q, \Sigma, \Gamma_1, \Gamma_2, \delta, (q_I, \boldsymbol{\gamma}_{I1}, \boldsymbol{\gamma}_{I2}), (q_F, \boldsymbol{\gamma}_{F1}, \boldsymbol{\gamma}_{F2}))$ *is **probabilistic** if for any configuration* $(q, \boldsymbol{\gamma}_1, [\boldsymbol{\gamma}_2])$ *it holds that*

$$\forall\ q \xrightarrow{a, [\boldsymbol{\gamma}_1] \to [\boldsymbol{\gamma}_1'], [\boldsymbol{\gamma}_2] \to [\boldsymbol{\gamma}_2']/w} r \in \delta :\ w \geqslant 0 \tag{4.136}$$

and

$$\sum_{q \xrightarrow{a,[\gamma_1]\to[\gamma_1'],[\gamma_2]\to[\gamma_2']/w} r} w = 1. \tag{4.137}$$

Turing Completeness of Multi-stack Pushdown Automata

Besides modeling more complex languages than finite-state language models from §4.1, (multi-stack) pushdown automata will also serve an important part in analyzing some modern language models that we introduce later. Namely, we will show that Recurrent neural networks (cf. §5.1.2) can *simulate* any two-stack PDA. This will be useful when reasoning about the computational expressiveness of recurrent neural networks because of a fundamental result in the theory of computation, namely, that two-stack PDAs are *Turing complete*:

Theorem 4.2.6. *Any 2-stack pushdown automaton is Turing complete.*

Proof. The equivalence is quite intuitive: the two stacks (which are infinite in one direction) of the 2-PDA can simulate the tape of a Turing machine by popping symbols from one stack and pushing symbols onto the other one simultaneously. The head of the Turing machine then effectively reads the entries at the top of one of the two stacks. For a formal proof, see Theorem 8.13 in **?**. ■

This is also the reason why we only have to consider two-stack PDAs—they can compute everything that can be computed, meaning that additional stacks do not increase their expressiveness!

Since unweighted pushdown automata are simply special cases of weighted PDAs, which are equivalent to probabilistic PDAs, we can therefore also conclude:

Corollary 4.2.1. *Any weighted 2-stack pushdown automaton is Turing complete.*

Corollary 4.2.2. *Any probabilistic 2-stack pushdown automaton is Turing complete.*

A straightforward consequence of the Turing completeness of two-stack PPDAs is that their *tightness* is undecidable.

Theorem 4.2.7 (Tightness of 2-PPDA is undecidable)**.** *The tightness of a probabilistic two-stack pushdown automaton is undecidable.*

Proof. We start with a simple observation: a pushdown automaton $\mathcal{P}$ is tight if and only if it *halts* on inputs with measure 1 (given the probability measure on $\Sigma^* \cup \Sigma^\infty$ defined in §2.5.4), as this, by the definition of the language accepted by the WPDA (cf. Definition 4.2.47), corresponds to its language only containing *finite* strings with probability 1.

Let $\mathcal{T}$ then be a Turing machine and $\mathcal{P}$ be a 2-PPDA which simulates it. Then, $\mathcal{P}$ is tight if and only if it halts with probability 1 (again, based on the probability measure from above). This is equivalent to the problem of $\mathcal{T}$ halting

with probability 1—this, however, is a variant of the **halting problem**, which is one of the fundamental undecidable problems. We have therefore reduced the problem of determining the tightness of 2-PPDAs to the halting problem, implying that the former is undecidable. ■

You might wonder what this means for the (weighted) languages recognized by multiple-stack (weighted) automata. Turing machines can recognize *recursively enumerable languages*. This means that weighted multi-stack pushdown automata model *distributions* over recursively enumerable languages. To see why this might be useful, let us finish the discussion of context-free languages with an example of a language model that is *not* context-free:

Example 4.2.13. *Let* $\Sigma = \{a\}$ *and* $p_{LM}(a^n) = e^{-\lambda}\frac{\lambda^n}{n!}$ *for* $n \in \mathbb{N}_{\geqslant 0}$*, i.e.,* $L(p_{LM}) \ni \boldsymbol{y} = a^n \sim \text{Poisson}(\lambda)$ *for some* $\lambda > 0$*. This language is not context-free: the proof, however, is not trivial. We direct the reader to Icard (2020) for one.*

4.3 Exercises

Exercise 4.1

Prove the following lemma.

Lemma 4.3.1. *Let* $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ *and* $q \in Q$. *Then*

$$Z(\mathcal{A}, q) = \sum_{q \xrightarrow{a/w} q' \in \delta_{\mathcal{A}_L}} \mathcal{W}\left(q \xrightarrow{a/\cdot} q'\right) Z(\mathcal{A}, q') + \rho(q) \tag{4.138}$$

Exercise 4.2

Show that the expression for the log-likelihood of the n-gram model can be rewritten as

$$\ell\ell(\mathcal{D}) = \sum_{m=1}^{M} \sum_{t=1}^{|\boldsymbol{y}^{(m)}|} \log \theta_{y_n|\boldsymbol{y}<n} = \sum_{\substack{\boldsymbol{y} \\ |\boldsymbol{y}|=n}} C(\boldsymbol{y})\, \theta_{y_n|\boldsymbol{y}<n} \tag{4.139}$$

with the quantities as defined in Proposition 4.1.3. This is a common trick. It is also known as the **token to type switch** because we switch from counting over the individual tokens to counting over their identities (types)

Exercise 4.3

Let $C(\boldsymbol{y})$ be the string occurrence count for $\boldsymbol{y} \in \Sigma^*$ occurrence count as defined in Proposition 4.1.3. Show (or simply convince yourself) that, in a given training corpus $\mathcal{D}$

$$\sum_{y' \in \Sigma} C\left(y_1 \dots y_{n-1} y'\right) = C\left(y_1 \dots y_{n-1}\right) \tag{4.140}$$

Chapter 5

Neural Network Language Models

Chapter 4 introduced two classical language modeling frameworks: finite-state language models and context-free language models. While those served as a useful introduction to the world of language modeling, most of today's state-of-the-art language models go beyond the modeling assumptions of these two frameworks. This chapter dives into the diverse world of modern language modeling architectures, which are based on neural networks. We define multiple two of the most common architectures—recurrent neural networks and transformers—and some of their variants. The focus is again on rigorous formalization and theoretical understanding—we analyze the introduced models in terms of the theoretical foundations so far (e.g., expressiveness and tightness)—but, due to their practical applicability, we also study some practical aspects of the models.

We begin with recurrent neural networks.

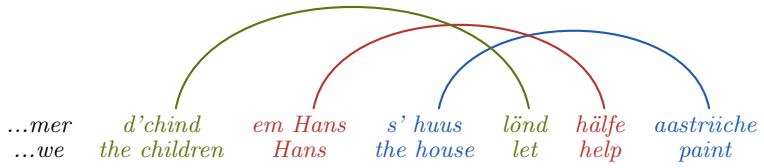
5.1 Recurrent Neural Language Models

The first neural language modeling architecture we consider is one based on recurrent neural networks. Recurrent neural networks capture the idea of the *sequential* processing of strings relatively naturally while also making decisions based on an *infinite* context. Before delving into the technical details of recurrent neural networks (RNNs), however, we first motivate the introduction of modeling contexts of unbounded length. Then, we formally define recurrent neural networks and devote a large portion of the section to their theoretical properties. The most important of those will be the Turing completeness of this architecture, as it has numerous consequences on the solvability of many of the tasks we might be interested in, such as finding the most probable string in the language model represented by a recurrent neural network and determining whether an RNN is tight.

5.1.1 Human Language is Not Context-free

Recall that we motivated the introduction of context-free languages by observing that finite memory is insufficient to model all formal phenomena of human language, e.g., infinite recursion (cf. Example 4.2.1). Context-free languages described by context-free grammars and pushdown automata were able to capture those. However, human language is more expressive than that—it includes linguistic phenomena that cannot be described by context-free grammars. A typical example is called **cross-serial dependencies**, which are common in *Swiss German*.

Example 5.1.1 (Cross-Serial Dependencies in Swiss German, Shieber, 1985)**.** *Swiss German is a textbook example of a language with grammatical cross-serial dependencies, i.e., dependencies in which the arcs representing them, cross. In the example sentence below, the words connected with arcs are objects and verbs belonging to the same predicates (verb phrases). Because of that, they have to agree on the form—they depend on one another. As we show next, context-free languages cannot capture such dependencies.*



Why are cross-serial dependencies non-context-free? Before reasoning about the phenomenon of cross-serial dependencies, we revisit Example 4.2.1 with a somewhat more formal approach. The arbitrarily deep nesting can, for example, be abstractly represented with the expression

$$xA^nB^ny \tag{5.1}$$

with[1]

$$
\begin{aligned}
x &= \text{“The cat”} \\
A &= \text{“the dog”} \\
B &= \text{“barked at”} \\
y &= \text{“likes to cuddle”}.
\end{aligned}
$$

From this abstract perspective, center embeddings are very similar to the D(1) language (Example 4.2.5), in that every noun phrase “the dog” has to be paired with a verb phrase “barked at”, which cannot be represented by any regular language.

In a similar fashion, Example 5.1.1 can abstractly be represented with the expression

$$x A^m B^n C^m y D^n z \tag{5.2}$$

with

$$
\begin{aligned}
x &= \text{“...mer”} \\
A &= \text{“d’chind”} \\
B &= \text{“em Hans”} \\
y &= \text{“s’ huus”} \\
C &= \text{“lönd”} \\
D &= \text{“hälfe”} \\
z &= \text{“aastriiche”}.
\end{aligned}
$$

Admittedly, this is a relatively uncommon formulation even with $n = m = 1$. It should be taken with a grain of salt, as the title of the original publication discussing this phenomenon, *Evidence against context-freeness* (Shieber, 1985), also suggests. However, theoretically, the number of repetitions of “d’chind” and “lönd”, as well as “em Hans” and “hälfe”, can be increased arbitrarily. Repeating the former would correspond to having many groups of children. The last of the groups would let Hans help paint the house, whereas each of the previous groups would let the group after them either let Hans paint the house or recurse onto another group of children. Similarly, repeating “em Hans” and “hälfe” would correspond to a number of Hanses, each either helping another Hans or helping paint the house. Then, using the pumping lemma for *context-free* languages, it can be shown that the expressions of the form in Eq. (5.2) cannot be recognized by any context-free grammar. We refer the readers to Hopcroft et al. (2006, Example 7.20) for detailed proof.

Example 5.1.1 means that to model a human language formally, we need more expressive formalisms than context-free grammars or pushdown automata as described in the previous sections.[2] However, instead of defining a more

[1] In this case, we of course only consider an arbitrarily long sequence of barking dogs.

[2] On the other hand, note that we would ideally also like to upper-bound the expressive power of the formal models, as this introduces useful inductive biases for learning and sparks insights into how humans process language. This means that we would not simply like to jump to Turing-complete models in such an exploration of language models.

expressive formalism motivated by formal language theory (like we did with context-free grammars and center embeddings), we now introduce recurrent neural networks, which, as we will see, under certain assumptions, have the capacity to model all computable languages (i.e., they are Turing complete). Moreover, they can also model *infinite* lengths of the context $\boldsymbol{y}_{<t}$ in a very flexible way. In the next section, we define them formally.

5.1.2 Recurrent Neural Networks

As discussed, natural languages are beyond the descriptive power of regular and context-free languages. Now, we turn to a class of models that is theoretically capable of recognizing all computable languages: **recurrent neural networks** (RNNs).[3]

An Informal Introduction to Recurrent Neural Networks

Human language is inherently sequential: we produce and consume both spoken as well as written language as a stream of units.[4] This structure is reflected in some of the algorithms for processing language we have seen so far. For example, finite-state automata (cf. §4.1) process the input string one symbol at a time and build the representations of the string seen so far in the current state of the automaton. Pushdown automata function similarly, but additionally keep the stack as part of the configuration.

Recurrent neural networks are neural networks that capture the same idea of iterative processing of the input but do so in a more flexible way. A recurrent neural network sequentially processes a sequence of inputs and, while doing so, produces a sequence of **hidden states**, which we will denote as $\mathbf{h}$, based on a transition function in form of a **recurrent dynamics map**, which acts similarly to a (deterministic) transition function in a finite-state machine: given the current hidden state and an input symbol, it (deterministically) determines the next hidden state. The hidden states play, as we will see, an analogous role to the states of a finite-state automaton or the configuration of a pushdown automaton. The current hidden state of a recurrent neural network at time t determines, together with the input at time t, through the dynamics map, the hidden state at time $t + 1$—indeed, very similar to how finite-state automata process strings and transition between their states. Again, the hidden state can be thought of as a compact (constant-size) summary of the input $\boldsymbol{y}_{\leqslant t}$ seen so far and should ideally characterize $\boldsymbol{y}_{\leqslant t}$ as well as possible (in the sense of retaining all information required for continuing the string). Remember from §4.2.1 that the finite number of states of a finite-state automaton presented a serious limitation to its ability to model human language. As we will see, the main difference between (weighted) finite-state automata and RNNs is that

[3] In this subsection, we focus on the applications of recurrent neural networks to language modeling. However, recurrent neural networks have been widely used to process sequential data and time series, thanks to their power of taking in arbitrary-length inputs.

[4] We will discuss in much more detail what "units" correspond to in **??**.

the latter can work with *infinite* state spaces, for example, $\mathbb{R}^D$ in the abstract formulation, or $\mathbb{Q}^D$ in a digital computing system, such as a computer. This, together with the flexibility of the transition function between hidden states, will allow RNNs to represent more complex languages than those recognized by finite-state automata or context-free grammars. In fact, the large state space and the flexible transition functions endow RNNs, under some assumptions, with the possibility to model infinitely-long-term dependencies on the input string, distinguishing them from the Markovian n-gram models.

You might wonder why we refer to the current state of an RNN as *hidden* states instead of only as *states*, as with finite-state automata. Indeed, when analyzing recurrent neural networks in terms of their expressivity and connections to classical models of computation, we will regard the hidden states as completely analogous to states in a finite-state or pushdown automaton. The hidden part comes from the fact that the hidden states $\mathbf{h}$ are usually not what we are interested in when modeling language with an RNN. Rather, $\mathbf{h}$ is simply seen as a component in a system that produces individual *conditional probabilities* over the next symbol, as in sequence models (cf. Definition 2.5.2)—these conditional probabilities are the actual "visible" parts, while the "internal" states are, therefore, referred to as hidden.

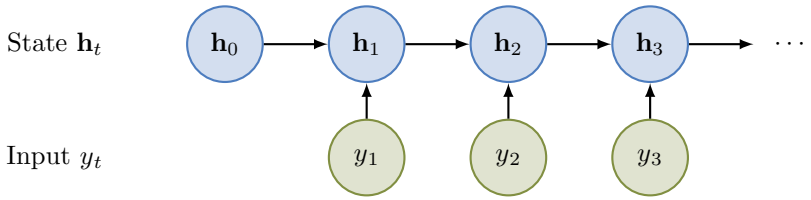
RNNs are abstractly represented in different ways in the literature. Often, they are represented as a sequence of hidden states and the input symbols consumed to arrive at those states—this is shown in Fig. 5.1a. They can also be presented more similarly to automata, with (a possibly infinite) labeled graph, where the transition labels again correspond to the symbols used to enter the individual states. This is presented in Fig. 5.1b. Lastly, due to the infinite state space, one can also think of an RNN as a system that keeps the most current hidden state in memory and updates it as new symbols are consumed—this is shown in Fig. 5.1c.

A Formal Definition of Recurrent Neural Networks

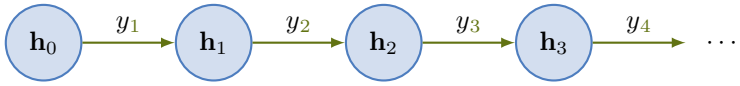
Having introduced RNNs and their motivations informally, we now move to their formal definition. Our definition and treatment of recurrent neural networks might differ slightly from what you might normally encounter in the literature. Namely, we define RNNs below as abstract systems transitioning between possibly infinitely-many states. Our definition will allow for an intuitive connection to classical language models such as finite-state and pushdown language models as well as for tractable theoretical analysis in some special cases. Specifically, when analyzing RNNs theoretically, we will make use of their connections to automata we saw in Chapter 4.

In an abstract sense, recurrent neural networks can be defined as a system transitioning between possibly infinitely-many states, which we will assume to be vectors in a vector space. Specifically, we will distinguish between *real* and *rational* recurrent neural networks.

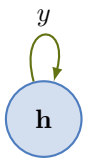
Definition 5.1.1 (Real-valued Recurrent Neural Network)**.** *Let Σ be an alphabet.*



(a) An abstract depiction of how an RNN processes one symbol in a string. The hidden state $\mathbf{h}_t$ summarizes the inputs $y_1 y_2 \dots y_t$.



(b) An abstract depiction of an RNN as an automaton. The transitions between the possibly infinitely-many hidden states are determined by the dynamics map.



(c) An abstract depiction of an RNN as a system updating the hidden state $\mathbf{h}$ depending on the input y.

Figure 5.1: Different possible depictions of an abstract RNN model. The way that the hidden states are updated based on the input symbol y_t is abstracted away.

A (deterministic) ***real-valued recurrent neural network*** $\mathcal{R}$ *is a four-tuple* $(\Sigma, D, \mathbf{f}, \mathbf{h}_0)$ *where*

- Σ *is the alphabet of input symbols;*
- D *is the dimension of* $\mathcal{R}$*;*
- $\mathbf{f}\colon \mathbb{R}^D \times \Sigma \to \mathbb{R}^D$ *is the dynamics map, i.e., a function defining the transitions between subsequent states;*
- $\mathbf{h}_0 \in \mathbb{R}^D$ *is the* ***initial state****.*

We analogously define **rational-valued recurrent neural networks** as recurrent neural networks with the hidden state space $\mathbb{Q}^D$ instead of $\mathbb{R}^D$. You might wonder why we make the distinction. Soon, when we take on theoretical analysis of RNNs, it will become important over which state spaces the models are defined. RNNs implemented in a computer using floating-point numbers, of course, cannot have irrational-valued weights—in this sense, all implemented recurrent neural networks are rational. However, defining the models over the real numbers crucially allows us to perform operations from calculus for which some sort of continuity and smoothness is required, for example, differentiation for gradient-based learning (cf. §3.2.3).

Example 5.1.2. *An example of a rational-valued RNN is the series*

$$h_t = \frac{1}{2} h_{t-1} + \frac{1}{h_{t-1}} \tag{5.3}$$

which we considered in Example 3.1.1. In this case

- $\Sigma = \{a\}$
- $D = 1$
- $\mathbf{f}\colon (x, a) \mapsto \frac{1}{2}x + \frac{1}{x}$
- $\mathbf{h}_0 = 2$

Example 5.1.3. *The tuple* $\mathcal{R} = (\Sigma, D, \mathbf{f}, \mathbf{h}_0)$ *where*

- $\Sigma = \{a, b\}$
- $D = 2$
- $\mathbf{f}\colon (\mathbf{x}, y) \mapsto \begin{cases} \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} \mathbf{x} & \textbf{\textit{if}}\ y = a \\ \begin{pmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{pmatrix} \mathbf{x} & \textbf{\textit{otherwise}} \end{cases}$
- $\mathbf{h}_0 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$

is an example of a real-valued RNN which rotates the current hidden state by the angle ϕ if the input symbol is a and rotates it by ψ if the symbol is b.

Example 5.1.4. *Another example of an RNN would be the tuple*

- $\Sigma = \text{GOOD} \cup \text{BAD} = \{\textit{“great”}, \textit{“nice”}, \textit{“good”}\} \cup \{\textit{“awful”}, \textit{“bad”}, \textit{“abysmal”}\}$
- $D = 2$
- $\mathbf{f}\colon (\mathbf{h}, a) \mapsto \begin{cases} \mathbf{h} + \begin{pmatrix} 1 \\ 0 \end{pmatrix} & \textbf{\textit{if}}\ a \in \text{GOOD} \\ \mathbf{h} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} & \textbf{\textit{otherwise}} \end{cases}$
- $\mathbf{h}_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$

which counts the number of occurrences of positive and negative words.

To define *language models* using recurrent neural networks, we will use them as the encoder functions enc in our general language modeling framework (cf. §3.1). To connect Definition 5.1.1 with the general LM framework, we define the RNN encoding function.

Definition 5.1.2 (Recurrent Neural Encoding Function)**.** *Let $\mathcal{R} = (\Sigma, D, \mathbf{f}, \mathbf{h}_0)$ be a recurrent neural network. A **recurrent neural encoding function** $\mathsf{enc}_{\mathcal{R}}$ is a representation function (cf. Definition 3.1.5) that recursively encodes strings of arbitrary lengths using its dynamics map $\mathbf{f}$:*

$$\mathsf{enc}_{\mathcal{R}}\left(\boldsymbol{y}_{<t+1}\right) \stackrel{\text{def}}{=} \mathbf{f}(\mathsf{enc}_{\mathcal{R}}(\boldsymbol{y}_{<t}), y_t) \in \mathbb{R}^D \tag{5.4}$$

and

$$\mathsf{enc}_{\mathcal{R}}(\boldsymbol{y}_{<1}) \stackrel{\text{def}}{=} \mathbf{h}_0 \in \mathbb{R}^D \tag{5.5}$$

Intuitively, an RNN $\mathcal{R}$ takes an input string $\boldsymbol{y}$ and encodes it with the encoding function $\mathsf{enc}_{\mathcal{R}}$ by sequentially applying its dynamics map $\mathbf{f}$. The representations of individual prefixes (cf. Definition 2.3.6) of the input string are called hidden states.

Definition 5.1.3 (Hidden State)**.** *Let $\mathcal{R} = (\Sigma, D, \mathbf{f}, \mathbf{h}_0)$ be an RNN. The hidden state $\mathbf{h}_t \in \mathbb{R}^D$ describes state of $\mathcal{R}$ after reading y_t. It is recursively computed according to the dynamics map $\mathbf{f}$ as follows:*

$$\mathbf{h}_t \stackrel{\text{def}}{=} \mathsf{enc}_{\mathcal{R}}(\boldsymbol{y}_{<t+1}) = \mathbf{f}(\mathbf{h}_{t-1}, y_t) \tag{5.6}$$

Example 5.1.5. *The hidden states of the RNN from Example 5.1.2 are the individual values h_t, which, as t increases, approach $\sqrt{2}$.*

Recurrent Neural Sequence Models

A recurrent neural network based on Definition 5.1.1 on its own does not yet define a sequence model, but simply a *context encoding function* $\mathsf{enc}_{\mathcal{R}} : \overline{\Sigma}^* \to \mathbb{R}^D$. To define a sequence model based on an RNN, we simply plug in the RNN encoding function Definition 5.1.2 into the General language modeling framework from §3.1.

Definition 5.1.4 (Recurrent neural sequence model)**.** *Let $\mathcal{R} = (\Sigma, D, \mathbf{f}, \mathbf{h}_0)$ be a recurrent neural network and $\mathbf{E} \in \mathbb{R}^{|\overline{\Sigma}| \times D}$ an symbol representation matrix. A D-dimensional **recurrent neural sequence model** over an alphabet Σ is a tuple $(\Sigma, D, \mathbf{f}, \mathbf{E}, \mathbf{h}_0)$ defining the sequence model of the form*

$$p_{SM}(y_t \mid \boldsymbol{y}_{<t}) \stackrel{\text{def}}{=} \text{softmax}(\mathbf{E}\, \mathsf{enc}_{\mathcal{R}}(\boldsymbol{y}_{<t}))_{y_t} = \text{softmax}(\mathbf{E}\, \mathbf{h}_{t-1})_{y_t}. \tag{5.7}$$

From this perspective, we see that RNNs are simply a special case of our general language modeling framework with parameterized representations of tokens $y \in \overline{\Sigma}$ and the history $\boldsymbol{y} \in \Sigma^*$ (cf. §3.1)—an RNN simply defines how the encoding function enc is specified. The three figures from Fig. 5.1 are presented again with this probabilistic perspective in Fig. 5.2.

A Few More Definitions

In the following, we will often use the so-called one-hot encodings of symbols for concise notation. We define them here.

Definition 5.1.5 (One-hot encoding)**.** *Let Σ be an alphabet and $n : \Sigma \to \{1, \ldots, |\Sigma|\}$ a bijection (i.e., an ordering of the alphabet, assigning an index to each symbol in Σ). A **one-hot encoding** $\mathbf{o}$. is a representation function of the symbols in Σ which assigns the symbol $y \in \Sigma$ the $n(y)^{th}$ basis vector:*

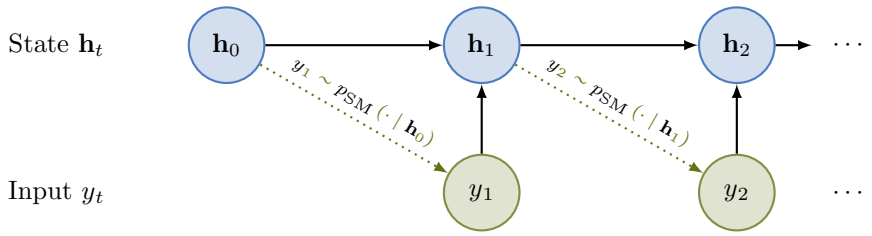
$$\mathbf{o}_y \stackrel{\text{def}}{=} \mathbf{d}_{n(y)}, \tag{5.8}$$

where here $\mathbf{d}_n$ is the n^{th} canonical basis vector, i.e., a vector of zeros with a 1 at position n.

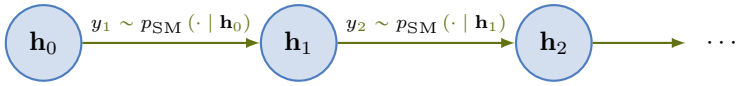
Example 5.1.6 (One-hot encoding)**.** *Let $\Sigma = \{\text{"large"}, \text{"language"}, \text{"models"}\}$ and $n = \{\text{"large"}: 1, \text{"language"}: 2, \text{"models"}: 3\}$. The one-hot encoding of the vocabulary is:*

$$\mathbf{o}_{\text{"large"}} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \mathbf{o}_{\text{"language"}} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \mathbf{o}_{\text{"models"}} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{5.9}$$

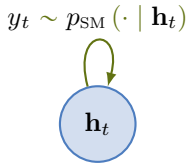
Many specific variants of recurrent neural networks define the dynamics map $\mathbf{f}$ in a specific way: the output of the function is some element-wise (non-linear) transformation of some "inner" function $\mathbf{g}$. The dynamics map of such an RNN is then the composition of $\mathbf{g}$ and the non-linearity.



(a) An abstract depiction of how an RNN generates a string one symbol at a time. The hidden state $\mathbf{h}_t$ summarizes the string $y_1 y_2 \dots y_t$ generated so far. The dotted lines denote the sampling steps.



(b) An abstract depiction of a generative RNN as an automaton.



(c) An abstract depiction of an RNN as a system updating the hidden state $\mathbf{h}_t$ depending on the generated symbol y_t.

Figure 5.2: Different possible depictions of an abstract RNN model *generating* symbols. The way that the hidden states are updated based on the input symbol y_t is abstracted away.

Definition 5.1.6 (Activation function)**.** *Let* $\mathcal{R} = (\Sigma, D, \mathbf{f}, \mathbf{E}, \mathbf{h}_0)$ *be an RNN. If the hidden states* $\mathbf{h}_t$ *of the RNN are computed as*

$$\mathbf{h}_t = \sigma(\mathbf{g}(\mathbf{h}_{t-1}, y)) \tag{5.10}$$

for some function $\mathbf{g}\colon \mathbb{R}^D \times \Sigma \to \mathbb{R}^D$ *and some function* $\sigma\colon \mathbb{R} \to \mathbb{R}$ *which is computed* element-wise *(that is,* $\sigma(\mathbf{x})_d = \sigma(x_d)$ *for all* $d = 1, \ldots, D$ *and* $\mathbf{x} \in \mathbb{R}^D$*), we call* σ *an* ***activation function****.*

This finishes our formal definition of recurrent neural networks. We next consider some of their theoretical properties, starting with tightness.

5.1.3 General Results on Tightness

We now discuss a general result on the tightness of recurrent neural sequence models, as defined in Definition 5.1.4. The analysis is straightforward and is a translation of the generic results on tightness (cf. §3.1.5) to the case of the norm of the hidden states of an RNN, $\mathbf{h}_t$ as the encodings of the prefixes $\boldsymbol{y}_{\leqslant t}$.

Theorem 5.1.1 (Tightness of Recurrent Neural Sequence Model)**.** *A recurrent neural sequence model is tight if for all time steps* t *it holds that*

$$s \left\|\mathbf{h}_t\right\|_2 \leqslant \log t, \tag{5.11}$$

where $s \stackrel{\text{def}}{=} \max_{y \in \Sigma} \left\|\mathbf{e}(y) - \mathbf{e}(\text{EOS})\right\|_2$.

Proof. This is simply a restatement of Theorem 3.1.6 for the case when enc takes the form of a general RNN encoding function, $\mathsf{enc}_{\mathcal{R}}$. ■

Corollary 5.1.1 (RNNs with bounded dynamics maps are tight)**.** *A recurrent neural sequence model* $\mathcal{R} = (\Sigma, D, \mathbf{f}, \mathbf{h}_0)$ *with a bounded dynamics map* $\mathbf{f}$*, i.e, with a dynamics map* $\mathbf{f}$ *such that*

$$\left|\mathbf{f}(\mathbf{x})_d\right| \leqslant M \tag{5.12}$$

for some $M \in \mathbb{R}$*, for all* $d = 1, \ldots, D$ *and all* $\mathbf{x} \in \mathbb{R}^D$*, is tight.*

Proof. If the dynamics map is bounded, the norm of the hidden state, $\left\|\mathbf{h}_t\right\|_2$, is bounded as well. This means that the left-hand-side of Eq. (5.11) is *constant* with respect to t and the condition holds trivially. ■

A special case of Corollary 5.1.1 is RNNs with bounded *activation* functions (cf. Definition 5.1.6). Those are tight if the activation function itself is bounded. This implies that all standard sigmoid and tanh activated recurrent neural networks are tight. However, the same does not hold for RNNs with unbounded activation functions, which have lately been more popular (one of the reasons for this is the vanishing gradient problem, which we discuss in **??**).

Example 5.1.7 (RNNs with unbounded activation functions may not be tight)**.** *A very popular unbounded activation function is the so-called* ***rectified liner unit*** *(*ReLU*), defined as*

$$\mathrm{ReLU}\left(x\right) \stackrel{\text{def}}{=} \max\left(0, x\right). \tag{5.13}$$

This function is clearly unbounded.

Now suppose we had the following RNN over the simple alphabet $\Sigma = \{a\}$*.*

$$\mathbf{h}_t = \mathbf{h}_{t-1} + \left(1\right), \tag{5.14}$$

initial state

$$\mathbf{h}_0 = \left(0\right) \tag{5.15}$$

and the output matrix

$$\mathbf{E} = \begin{pmatrix} -1 \\ 1 \end{pmatrix} \tag{5.16}$$

where the top row of $\mathbf{E}$ *computes the logit of the* EOS *symbol and the bottom one the one of* a*. It is easy to see that*

$$\mathbf{h}_t = \left(t\right). \tag{5.17}$$

This already does not look promising for tightness—the norm of the hidden state, which is, in this case, $\left\|\left(t\right)\right\| = t$*, and is, therefore, increasing at a much higher rate than* $\mathcal{O}\left(\log t\right)$ *required by Theorem 5.1.1. We encounter a similar hint against tightness if we compute the conditional probabilities of the* EOS *symbol and the symbol* a*.*

$$p_{SM}\left(\text{EOS} \mid \boldsymbol{y}_{<t}\right) = \text{softmax}\left(\begin{pmatrix} -1 \\ 1 \end{pmatrix}\left(t-1\right)\right)_{\text{EOS}} = \frac{\exp\left[-t+1\right]}{\exp\left[-t+1\right] + \exp\left[t-1\right]} \tag{5.18}$$

$$p_{SM}\left(\text{EOS} \mid \boldsymbol{y}_{<t}\right) = \text{softmax}\left(\begin{pmatrix} -1 \\ 1 \end{pmatrix}\left(t-1\right)\right)_{a} = \frac{\exp\left[t-1\right]}{\exp\left[-t+1\right] + \exp\left[t-1\right]} \tag{5.19}$$

The probability of ending the string at time step t *is, therefore*

$$p_{SM}\left(\text{EOS} \mid \boldsymbol{y}_{<t}\right) = \frac{1}{1 + \exp\left[2\left(t-1\right)\right]}. \tag{5.20}$$

Intuitively, this means that the probability of ending the string (generating EOS*) diminishes rapidly with* t*—in this case much faster than any diverging sum required by Theorem 2.5.3. All signs, thus, point towards the RNN from Eq. (5.14) not being tight. Indeed, for this specific case, one can show using some algebraic manipulations that*

$$\sum_{\boldsymbol{y} \in \Sigma^*} p_{LN}\left(\boldsymbol{y}\right) = \sum_{n \in \mathbb{N}_{\geqslant 0}} p_{LN}\left(a^n\right) < 0.15 \tag{5.21}$$

where p_{LN} *is the locally normalized model induced by the RNN. This means that the RNN from Eq. (5.14) assigns less than* 0.15 *probability to finite strings—all other probability mass leaks to infinite sequences.*

Example 5.1.8 (RNNs with unbounded activation functions can still be tight)**.** *Example 5.1.7 showed that RNNs with unbounded activation functions can indeed result in non-tight sequence models. However, this is not necessarily the case, as this simple modification of the RNN from Example 5.1.7 shows. The only aspect of the RNN that we modify is the output matrix* $\mathbf{E}$*, which we change by flipping its rows:*

$$\mathbf{E} = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \tag{5.22}$$

Now the probability of ending the string at time step t *is*

$$p_{SM}\left(\text{EOS} \mid \boldsymbol{y}_{<t}\right) = \frac{\exp\left[t-1\right]}{\exp\left[-t+1\right] + \exp\left[t-1\right]} = \frac{1}{\exp\left[-2\left(t-1\right)\right]+1}. \tag{5.23}$$

Compared to Eq. (5.20), the probability of EOS *in Eq. (5.23) does not diminish. Indeed, since* $\frac{1}{\exp[-2(t-1)]+1} > \frac{1}{2}$ *for all* t*, the sum*

$$\sum_{t=0}^{\infty} p_{SM}\left(\text{EOS} \mid \boldsymbol{y}_{<t}\right) \tag{5.24}$$

diverges, which, according to Proposition 2.5.6 implies that the sequence model is tight.

5.1.4 Elman and Jordan Networks

The characterization of dynamics maps we gave in Definition 5.1.4 allows for $\mathbf{f}$ to be an arbitrary mapping from the previous state and the current input symbol to the new state. In this section, we introduce two seminal and particularly simple parameterizations of this map—the simplest recurrent neural sequence models. We term them Elman sequence models and Jordan sequence models, as each is inspired by architectures proposed by Elman (1990) and Jordan (1986), respectively. The definitions we present here are slightly different than those found in the original works—most notably, both Elman and Jordan networks were originally defined for *transduction* (mapping an input string to an output string, as with translation) rather than language modeling.

Put simply, these two models *restrict* the form of the dynamics map $\mathbf{f}$ in the definition of an RNN (cf. Definition 5.1.1). They define particularly simple relationships between the subsequent hidden states, which are composed of affine transformations of the previous hidden state and the representation of the current input symbol passed through a non-linear activation function (cf. Definition 5.1.6). The affine transformations are performed by different matrices and bias vectors—the parameters of the model (cf. Assumption 3.2.2)—each transforming a separate part of the input to the dynamics map.

Definition 5.1.7 (Elman Sequence Model (Elman, 1990))**.** *An* ***Elman sequence model*** $\mathcal{R} = (\Sigma, D, \mathbf{U}, \mathbf{V}, \mathbf{E}, \mathbf{b}_h, \mathbf{h}_0)$ *is a* D*-dimensional recurrent neural sequence model over an alphabet* Σ *with the following dynamics map*

$$\mathbf{h}_t = \sigma\left(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{V}\mathbf{e}'(y_t) + \mathbf{b}_h\right). \tag{5.25}$$

Here, $\mathbf{e}'(\cdot)\colon \Sigma \to \mathbb{R}^R$ *is the input symbol* ***embedding function*** *which represents each symbol* $y \in \Sigma$ *as a* R*-dimensional vector and* σ *is an element-wise non-linearity.* $\mathbf{b}_h \in \mathbb{R}^D$*,* $\mathbf{U} \in \mathbb{R}^{D \times D}$*, and* $\mathbf{V} \in \mathbb{R}^{D \times R}$*.*

Due to its simplicity, the Elman RNN is also known as the *vanilla RNN* variant, emphasizing it is one of the most fundamental variants of the framework.

On the symbol representations. Notice that in Eq. (5.25), the input symbols y_t are first transformed into their vector representations $\mathbf{e}'(y_t)$ and then additionally linearly transformed using the matrix $\mathbf{V}$. This results in an overparametrized network—since the symbols are already embedded using the representation function $\mathbf{e}'()$, the matrix $\mathbf{V}$ is theoretically superfluous and could be replaced by the identity matrix. However, the matrix $\mathbf{V}$ could still be useful if the representations $\mathbf{e}'(y_t)$ are fixed—in this case, the matrix can be used by the RNN to transform the representations during training to fit the training data better. This is especially useful if the symbol representations $\mathbf{e}'(y_t)$ already represent the input symbols in a compact representation space in which the parameters can be shared across different symbols. Alternatively, we could represent the symbols using their one-hot encodings, i.e., $\mathbf{e}'(y_t) = \mathbf{o}_{y_t}$, in which case the columns of the matrix $\mathbf{V}$ would correspond to the symbol representations (analogously to the representation matrix $\mathbf{E}$ from Eq. (3.46)). However, notice that in this case, the representations on the symbols do not share any parameters, and each column of the matrix is therefore an unconstrained vector. Such matrix-lookup-based *input* symbol representations from Eq. (5.25) are sometimes **tied**, i.e., $\mathbf{e}'(\cdot) = \mathbf{e}(\cdot)$, with the *output* symbol representations from the embedding matrix $\mathbf{E}$ in the definition of the sequence model induced by an RNN (cf. Definition 3.1.11 and Eq. (5.7)).

However, embedding tying is non-essential to representation-based LMs. The input symbol embedding function can always be chosen independently with the output symbol embedding function Definition 3.1.6.

The Jordan network is somewhat different in that it feeds the *output* logits computed through the output matrix $\mathbf{E}$ into the computation of the next state, and not directly the hidden state.

Definition 5.1.8 (Jordan Sequence Model (Jordan, 1986))**.** *A* ***Jordan sequence model*** *is a* D*-dimensional recurrent neural sequence model over an alphabet* Σ *with the following dynamics map*

$$\mathbf{h}_t = \sigma\big(\mathbf{U}\mathbf{r}_{t-1} + \mathbf{V}\mathbf{e}'(y_t) + \mathbf{b}_h\big) \tag{5.26}$$

$$\mathbf{r}_t = \sigma_o(\mathbf{E}\mathbf{h}_t) \tag{5.27}$$

Again, $\mathbf{e}'(\cdot)\colon \Sigma \to \mathbb{R}^R$ *is the input symbol embedding function which represents each symbol* $y \in \Sigma$ *as a* R*-dimensional vector while* σ *and* σ_o *are element-wise non-linearities.* $\mathbf{b}_h \in \mathbb{R}^D$*,* $\mathbf{U} \in \mathbb{R}^{D \times D}$*, and* $\mathbf{V} \in \mathbb{R}^{D \times R}$*.*

Notice that the hidden state $\mathbf{h}_t$ in **??** is not computed based on the previous hidden state $\mathbf{h}_{t-1}$, but rather on the transformed outputs $\mathbf{r}_{t-1}$—this is analogous

to feeding back in the logits computed in Eq. (5.7) into the computation of $\mathbf{h}$ rather than the previous hidden state. The sequence model induced by a Jordan network is then directly induced by the logits $\mathbf{r}_t$ (i.e., the conditional probabilities are computed by putting v_t through the softmax.

In both architectures, the activation function σ can be any suitable element-wise function. The canonical choices for it have been the sigmoid and tanh functions, however, a more common choice nowadays is the ReLU function or any of its more modern variants.[5]

Since we will refer to the individual matrices defining the dynamics maps in Elman and Jordan networks quite a lot in the next subsections, we give them specific names. The matrix $\mathbf{U}$, which linearly transforms the previous hidden state (or the output) is the **recurrence matrix**. The matrix $\mathbf{V}$, which linearly transforms the representations of the input symbol, is called the **input matrix**. Lastly, the matrix which linearly transforms the hidden state before computing the output values $\mathbf{r}_t$ with an activation function is called the **output matrix**. $\mathbf{b}_h$ is the hidden **bias vector**.

Tightness of Elman and Jordan Recurrent Neural Networks As a simple corollary of Corollary 5.1.1, we can characterize the tightness of Elman and Jordan recurrent neural networks as follows.

Corollary 5.1.2 (Tightness of simple RNNs)**.** *Elman and Jordan RNNs with a bounded activation function σ are tight.*

5.1.5 Expressiveness of Recurrent Neural Networks

Recurrent neural networks are one of the fundamental and most successful neural language model architectures. In this section, we study some theoretical explanations behind their successes as well as some of their theoretical limitations. We do that through the lens of formal language theory—we will study the classes of (weighted) formal languages (such as the regular languages and all computable languages) they can express. Inspecting complex models such as recurrent neural networks through the lens of formal language theory allows us to apply the well-studied theoretical results and understanding from the field to the recently more successful neural models. While studying neural language models, we will revisit various aspects of the classical language models introduced in Chapter 4—indeed, this was also our main motivation for studying those closely.

Specifically, we will focus mainly on the *Elman* recurrent neural networks due to their simplicity. However, note that most of the results presented in the section generalize to other architectures as well. We begin by investigating Elman RNNs in a practical setting, that is, under relatively strict and realistic assumptions, such as fixed-point arithmetic. We show that Elman RNNs under such a regime are in fact equivalent to weighted finite-state automata. Next, in the second subsection, we show that under more permissive assumptions of

[5] See Goodfellow et al. (2016, §6.3.1) for an overview of modern activation functions used in neural networks.

infinite precision and unbounded computation time, the Elman RNNs are Turing complete.

RNNs and Weighted Regular Languages

Analyzing complex systems with intricate temporal dependencies can be tricky. In fact, most, if not all, theoretical frameworks for analyzing models such as RNNs rely on various assumptions about their components to make the analysis feasible. One fruitful manner to analyze the expressivity of recurrent neural networks is by making simplifying assumptions on the non-linear activation functions, since those are what often make analysis difficult. A common simplification is the use of the Heaviside activation function.

Definition 5.1.9. *The **Heaviside** function is defined as*

$$H(x) = \begin{cases} 1 & \textbf{\textit{if}}\ x > 0 \\ 0 & \textbf{\textit{otherwise}} \end{cases} \tag{5.28}$$

In words, the Heaviside function maps every real value either 0 or 1, depending on whether it is greater than or less than zero. In the following, we will refer to the set $\{0, 1\}$ as $\mathbb{B} \stackrel{\text{def}}{=} \{0, 1\}$.

Definition 5.1.10 (Heaviside Elman Network)**.** *A **Heaviside Elman network** is an Elman network with Heaviside function H as the non-linearity.*

Elman network parameters. Importantly, note that the *parameters* of the network do not have to be elements of $\mathbb{B}$—we assume those can take arbitrary real (or rational) values. Indeed, networks constrained to parameters $\theta \in \mathbb{B}$ would only be able to recognize unweighted languages. Furthermore, for this section, we expand our definition of a real- or rational-weighted RNN to be able to contain *weights* ∞ and $-\infty$. While those are not real (or rational) numbers, we will see they become useful when we want to explicitly exclude specific sequences from the support of the model, i.e., when we want to assign probability 0 to them.

The central result of this section is captured in the following theorem.

Theorem 5.1.2 (Equivalence of Heaviside Elman RNNs and WFSAs)**.** *Heaviside Elman RNNs are equivalent to deterministic weighted finite-state automata with positive weights.*

We will prove this theorem by showing that an RNN with Heaviside activations is at most regular, and then showing how such an RNN can in fact simulate any deterministic WFSA. We show each direction as its own lemma.

Lemma 5.1.1. *The distribution represented by a recurrent neural network with a Heaviside non-linearity H is regular.*

Proof. Let $\mathcal{R} = (\Sigma, D, \mathbf{f}, \mathbf{E}, \mathbf{h}_0)$ be a recurrent neural network sequence model defining the sequence model p_{SM} as defined in Definition 5.1.1. We construct

a deterministic probabilistic FSA $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ representing the same weighted language as $\mathcal{R}$. First, we note that by the definition of the Heaviside function (Eq. (5.28)) every $\mathbf{h}_t \in \mathbb{B}^D$—this implies that the RNN defines a hidden state space with at most 2^D different hidden states. We construct a set of 2^D states Q, each of which is associated with a binary vector (i.e., one configuration of the hidden state). We will denote the hidden state associated with $q \in Q$ with $h(q)$, i.e., $Q \stackrel{\text{def}}{=} \left\{\mathcal{Q}(\mathbf{x}) \mid \mathbf{x} \in \mathbb{B}^D\right\}$. We will denote the inverse of $h(\cdot)$ by $h^{-1}(\cdot)$. Now, for every state $q \in Q$, construct a transition $q \xrightarrow{y/w} q'$ where $q' = \mathbf{f}((h(q), y)$ with the weight $w = p_{\text{SM}}(y \mid h(q)) = \text{softmax}(\mathbf{E}\, h(q))_y$. We define the final function ρ with $\rho(q) \stackrel{\text{def}}{=} p_{\text{SM}}(\text{EOS} \mid h(q))$.

It is easy to see that $\mathcal{A}$ defined this way is deterministic. We now prove that the weights assigned to strings $\boldsymbol{y} \in \Sigma^*$ by $\mathcal{A}$ and $\mathcal{R}$ are the same. Let $\boldsymbol{y} \in \Sigma^*$ with $|\boldsymbol{y}| = T$ and $\boldsymbol{\pi} = \left(h^{-1}(\mathbf{h}_0) \xrightarrow{y_1/w_1} q_1, \ldots, q_T \xrightarrow{y_T/w_T} q_{T+1}\right)$ the $\boldsymbol{y}$-labeled path starting in $h^{-1}(\mathbf{h}_0)$ (such a path exists, since we the defined automaton is *complete*—all possible transitions are defined for all states).

$$\begin{aligned}\mathcal{A}(\boldsymbol{y}) &= \lambda\left(h^{-1}(\mathbf{h}_0)\right) \cdot \left[\prod_{t=1}^{T} w_t\right] \cdot \rho(q_{T+1}) \\ &= 1 \cdot \prod_{t=1}^{T} p_{\text{SM}}(y_t \mid h(q_t)) \cdot p_{\text{SM}}(\text{EOS} \mid h(q_{T+1})) = p_{\text{LN}}(\boldsymbol{y})\end{aligned}$$

which is exactly the weight assigned to $\boldsymbol{y}$ by $\mathcal{R}$. Note that all paths not starting in $h^{-1}(\mathbf{h}_0)$ have weight 0 due to the definition of the initial function. ■

Let us look at an example of the construction above.

Example 5.1.9. *Let $\mathcal{R} = (\Sigma, D, \mathbf{f}, \mathbf{E}, \mathbf{h}_0)$ be a Heaviside RNN sequence model with the parameters*

$$\Sigma = \{a, b\} \tag{5.29}$$

$$D = 2 \tag{5.30}$$

$$\mathbf{f}(\mathbf{h}_t, y) = H\left(\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}\mathbf{h}_{t-1} + \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}\mathbf{o}_y\right) \tag{5.31}$$

$$\mathbf{E} = \begin{pmatrix}1 & 0\\ 0 & 1\\ 1 & -\infty\end{pmatrix} \tag{5.32}$$

$$\mathbf{h}_0 = \begin{pmatrix}0\\ 0\end{pmatrix} \tag{5.33}$$

and $n(a) = 1$, $n(b) = 2$, and $n(\text{EOS}) = 3$. The automaton corresponding to this RNN contains the states q_{ij} corresponding to the hidden states $\mathbf{h} = \begin{pmatrix}i\\ j\end{pmatrix}$. It is shown in Fig. 5.3; as we can see, the automaton indeed has an exponential number of useful states in the dimensionality of the hidden state, meaning that the RNN is a very compact way of representing it.

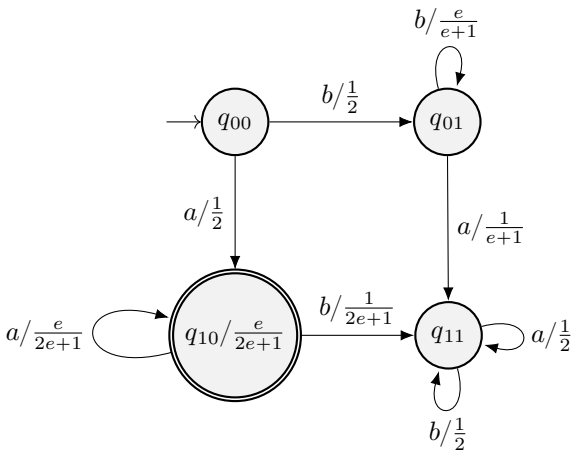


Figure 5.3: The WFSA corresponding to the RNN defined in Eq. (5.29).

Implications for recurrent neural language models. A similar argument regarding a finite number of states could be made for any RNN whose activation functions map onto a finite set. This is for example the case with any implementation of an RNN on a computed with finite-precision arithmetic—in that case, *all* deployed recurrent sequence models are regular, albeit very large ones in the sense of encoding possibly very large weighted finite-state automata. However, there are a few important caveats with this: firstly, notice that, although finite, the number of states represented by an RNN is *exponential* in the size of the hidden state. Even for moderate hidden state dimensionalities, this can be very large (hidden states can easily be of size 100–1000). In other words, one can view RNNs as very compact representations of large deterministic (weighted) finite-state automata whose transition functions are represented by the hidden state update function. Furthermore, since the topology of this implicit WFSA is completely determined by the update function of the RNN, it can be *learned* very flexibly yet efficiently based on the training data—this is made possible by the *sharing* of parameters across the entire graph of the WFSA instead of explicitly parametrizing every possible transition, as, for example, in §4.1.3, or hard-coding the allowed transitions as in §4.1.5. This means that the WFSA is not only represented, but also *parametrized* very efficiently by an RNN. Nevertheless, there is an important detail that we have somewhat neglected so far: this is the requirement that the simulated WFSA be *deterministic*. It turns out that, in contrast to unweighted finite-state automata, *not all* nondeterministic real-weighted FSAs can be determinized—in fact, it can be shown that almost all non-deterministic real-weighted FSAs cannot be determinized (Allauzen and Mohri, 2003). Furthermore, even if they *can* be determinized, the number of states of the determinized machine can be *exponential* in the size of the non-deterministic one (Buchsbaum et al., 2000). This implies two things. Firstly, in general, allowing nondeterminism, weighted finite-state automata are

more expressive than RNNs, which are inherently deterministic. Secondly, the compact nature of the RNN representation of WFSAs can be "undone" using determinization: that is, given an RNN, there might exist a roughly equally compact representation of a deterministic WFSA with a nondeterministic WFSA. We will explore the second point in more detail below as we dive into representing a *given* WFSA using an Elman RNN—as we will see, despite the intuition we tried to build here, an arbitrary WFSA with $|Q|$ states cannot be represented using an RNN with a hidden state of size $\log |Q|$.

To show the other direction of Theorem 5.1.2, we now give a variant of a classic theorem originally due to Minsky (1986) but with a *probabilistic* twist, allowing us to model *weighted* languages with Elman RNNs.

Lemma 5.1.2 (Elman RNNs can encode PFSAs)**.** *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a tight probabilistic deterministic finite-state automaton. Then, there exists a Heaviside-activated Elman network with a hidden state of size $\mathbf{h} = |\Sigma||Q|$ that encodes the same distribution as $\mathcal{A}$.*

Proof. We give proof by construction: given a WFSA $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$, we construct an Elman RNN $\mathcal{R} = (\Sigma, D, \mathbf{U}, \mathbf{V}, \mathbf{E}, \mathbf{b}_h, \mathbf{h}_0)$ accepting the same weighted language as $\mathcal{A}$: $L(\mathcal{A}) = L(\mathcal{R})$ by defining the elements of the tuple $(\Sigma, D, \mathbf{U}, \mathbf{V}, \mathbf{E}, \mathbf{b}_h, \mathbf{h}_0)$. From Theorem 4.1.1, we know we can assume $\mathcal{A}$ is locally normalized without loss of generality. Since the hidden state is obtained after passing the affine combination of the previous hidden state and the representation of the new input symbol through the Heaviside activation function, each hidden state lives in $\mathbb{B}^{|\Sigma||Q|}$. Let $n \colon Q \times \overline{\Sigma} \to [|Q||\Sigma|]$ be a bijection, i.e., an ordering of the set $\overline{\Sigma}$: it assigns each $y \in \overline{\Sigma}$ an integer which can then be used to index into matrices and vectors as we will see below. Furthermore, let $m \colon \overline{\Sigma} \to [|\Sigma| + 1]$ similarly be an ordering of the alphabet $\overline{\Sigma}$. Similarly to the proof of Lemma 5.1.1, we denote with $h()$ and $h^{-1}()$ the mappings from $Q \times \overline{\Sigma}$ to the hidden state space of the RNN and its inverse. The alphabet of the RNN of course matches the one of the WFSA. Furthermore, it turns out we do not need the bias vector, so we simply set $\mathbf{b}_h = \mathbf{0}$.

The hidden states of the RNN live in $\mathbb{B}^{|Q||\Sigma|}$. A hidden state $\mathbf{h}_t$ encodes the state p the simulated $\mathcal{A}$ is in at time t *and* the transition symbol b with which $\mathcal{A}$ "arrived" at p as a *one-hot encoding* of the pair (p, b). Formally,

$$h(p, y) = \mathbf{o}_{n(p,y)}. \tag{5.34}$$

This also means that $D = |Q||\Sigma|$. There is a small caveat here: how do we set the incoming symbol for the *initial* state of the automaton (the first time it is entered)? A straightforward solution would be to augment the alphabet of the RNN with the BOS symbol (cf. §2.4), which we define to be the label of the incoming arc denoting the initial state (this would be the only transition labeled with BOS). However, as we show later, the symbol used to arrive into p does not have an effect on the *subsequent* transitions—it is only needed to determine the *target* of the current transition. Therefore, we can simply represent the initial state $\mathbf{h}_0$ of $\mathcal{R}$ with the one-hot encoding of *any* pair (q_I, a), where q_I is the initial state of the WFSA and $a \in \Sigma$.

For example, for the fragment of a WFSA in Fig. 5.4, the hidden state encoding the current state p and the incoming arc b is of the form presented in Eq. (5.35).

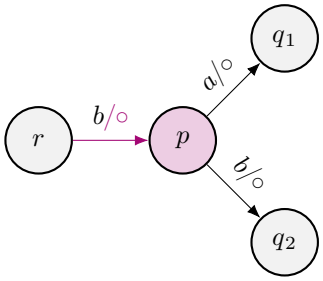


Figure 5.4: An example of a fragment of a WFSA.

$$\mathbf{h}_t = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \leftarrow n\,(p, b) \tag{5.35}$$

The recurrence matrix $\mathbf{U}$ lives in $\mathbb{R}^{|\Sigma||Q| \times |\Sigma||Q|}$. More precisely, it will only contain two values: 1 and -1. The main idea of the construction is for each column $\mathbf{U}_{:,n(q,y)}$ of the matrix to represent the "out-neighborhood" of the state $q \in Q$ in the sense that the column contains 1's at the indices corresponding to the state-symbol pairs (p, y) such that $\mathcal{A}$ transitions from p to q after reading in the symbol y. That is, for $p, q \in Q$ and $a, b \in \Sigma$, we define

$$U_{n(q,b),n(p,a)} \stackrel{\text{def}}{=} \begin{cases} 1 & \mid \text{ if } p \xrightarrow{b/\circ} q \in \delta \\ -1 & \mid \text{ otherwise} \end{cases}, \tag{5.36}$$

Because a is free, each row has $|\Sigma|$ entries that are 1; the rest are -1. This also means that each column in $\mathbf{U}$ is *repeated* $|\Sigma|$-times: once for each possible $a \in \Sigma$—this is precisely why, after entering the correct next state, the symbol used to enter it does not matter anymore and, in the case of the initial state, any "incoming" symbol can be chosen.

For example, for the fragment of a WFSA in Fig. 5.4, the recurrence matrix would take the form

$$\mathbf{U} = \begin{array}{c} n\,(p, b) \\ \downarrow \\ \begin{pmatrix} & -1 & \\ & \vdots & \\ & 1 & \\ \cdots & \vdots & \cdots \\ & 1 & \\ & \vdots & \\ & -1 & \end{pmatrix} \end{array} \begin{array}{l} \\ \\ \\ \leftarrow n\,(q_1, a) \\ \\ \leftarrow n\,(q_2, b) \\ \\ \\ \end{array} \tag{5.37}$$

and the matrix-vector product $\mathbf{U}\mathbf{h}_t$ with $\mathbf{h}_t$ from before results in

$$\mathbf{U}\mathbf{h}_t = \begin{pmatrix} -1 \\ \vdots \\ 1 \\ \vdots \\ 1 \\ \vdots \\ -1 \end{pmatrix} \begin{matrix} \\ \\ \leftarrow n\,(q_1, a) \\ \\ \leftarrow n\,(q_2, b) \\ \\ \\ \end{matrix} \tag{5.38}$$

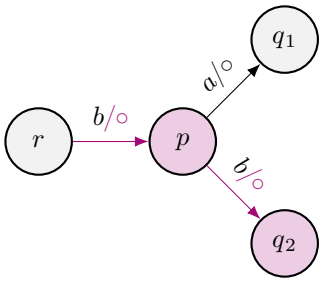
The input matrix $\mathbf{V}$ lives in $\mathbb{R}^{|\Sigma||Q|\times|\Sigma|}$, again containing only 1s and -1s, and encodes the information about which states can be reached by which symbols (from *any* state in $\mathcal{A}$). Its entries with a 1 correspond to the state-symbol pairs (q, b) (corresponding to the rows) such that q (corresponding to the column) is reachable with b from *some* other state:

$$V_{n(q,b),m(b)} \overset{\text{def}}{=} \begin{cases} 1 \mid & \text{if } \circ \xrightarrow{b/\circ} q \in \delta \\ -1 \mid & \text{otherwise} \end{cases} . \tag{5.39}$$

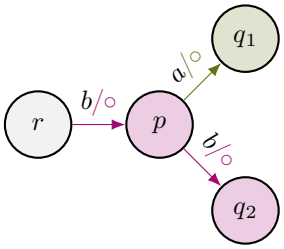
For example, for the fragment of a WFSA in Fig. 5.5a, the input matrix would take the form

$$\mathbf{V} = \begin{matrix} \\ \\ \\ \\ \\ \\ \\ \end{matrix} \begin{matrix} m\,(b) \\ \downarrow \\ -1 \\ \vdots \\ 1 \\ \vdots \\ 1 \\ \vdots \\ -1 \end{matrix} \begin{matrix} \\ \\ \\ \\ \\ \\ \\ \end{matrix} \begin{pmatrix} & & \\ & & \\ & & \\ \cdots & & \cdots \\ & & \\ & & \\ & & \end{pmatrix} \begin{matrix} \\ \\ \\ \leftarrow n\,(p, b) \\ \\ \leftarrow n\,(q_2, b) \\ \\ \end{matrix} \tag{5.40}$$

and the matrix-vector product $\mathbf{V}\mathbf{e}_{m(a)}$ and $\mathbf{V}\mathbf{e}_{m(b)}$ would take the form (see



(a) An example of a fragment of a WFSA.



(b) An example of a fragment of a WFSA.

also Fig. 5.5b)

$$\mathbf{Ve}_{m(a)} = \begin{pmatrix} -1 \\ \vdots \\ 1 \\ \vdots \\ -1 \end{pmatrix} \leftarrow n(q_1, a) \qquad \mathbf{Ve}_{m(b)} = \begin{pmatrix} -1 \\ \vdots \\ 1 \\ \vdots \\ 1 \\ \vdots \\ -1 \end{pmatrix} \begin{matrix} \\ \\ \leftarrow n(p, b) \\ \\ \leftarrow n(q_2, b) \\ \\ \end{matrix} \tag{5.41}$$

To put these components together, consider that, at each step of the computation, $\mathcal{R}$ computes $\mathbf{h}_{t+1} = H(\mathbf{U}\mathbf{h}_t + \mathbf{V}\mathbf{e}_a)$ where $y_{t+1} = a$. The input to the non-linearity is computed as follows:

$$\mathbf{U}\mathbf{h}_t + \mathbf{V}\mathbf{e}_{m(a)} = \begin{pmatrix} -1 \\ \vdots \\ 1 \\ \vdots \\ 1 \\ \vdots \\ -1 \end{pmatrix} \begin{matrix} \\ \\ \leftarrow n(q_1, a) \\ \\ \leftarrow n(q_2, b) \\ \\ \end{matrix} + \begin{pmatrix} -1 \\ \vdots \\ 1 \\ \vdots \\ -1 \end{pmatrix} \leftarrow n(q_1, a) \tag{5.42}$$

More formally, since $\mathcal{A}$ is by assumption deterministic, the only entry in which $\mathbf{U}\mathcal{Q}(\mathbf{h}_t) + \mathbf{V}\mathbf{o}_y$ is positive (specifically, 2), is $n(q, y)$, where $p \xrightarrow{y/w} q \in \delta$.

Therefore, by the definition of the Heaviside non-linearity (Definition 5.1.9) $H(\mathbf{U}\mathbf{h}_t + \mathbf{V}\mathbf{e}_a)$ contains only one non-zero element—at position $n(q_1, a)$. This shows that the new hidden state encodes the new state of $\mathcal{A}$ and the label of the transition used to enter it, showing that the constructed RNN $\mathcal{R}$ indeed captures the (unweighted) transition function of $\mathcal{A}$.

To generalize this construction to the weighted case, we include the transition weights, by adding an encoding matrix $\mathbf{E} \in \mathbb{R}^{(|\Sigma|+1)\times|\Sigma||Q|}$. It embeds the symbols $\overline{y} \in \overline{\Sigma}$ and thus enables the computation of the softmax in the definition of an RNN sequence model (cf. Definition 5.1.4).[6] More formally, for $y \in \Sigma$, we define

$$\mathbf{E}_{m(b)n(p,a)} \stackrel{\text{def}}{=} \begin{cases} \log w & \mid \textbf{if } p \xrightarrow{a/w} \circ \in \delta \\ -\infty & \mid \textbf{otherwise} \end{cases} \tag{5.43}$$

and for EOS, we define

$$\mathbf{E}_{m(\text{EOS})n(p,a)} \stackrel{\text{def}}{=} \begin{cases} \log \rho(q) & \mid \textbf{if } \rho(q) > 0 \\ -\infty & \mid \textbf{otherwise} \end{cases} \tag{5.44}$$

Note that the $-\infty$ entries are only needed whenever the original WFSA assigns 0 probability to some transitions. In many implementations using softmax-activated probabilities, this would not be required.

For example, for the fragment of a WFSA in Fig. 5.6, the output matrix would take the form

$$\mathbf{E} = \begin{pmatrix} & & n(p,b) & & \\ & & \downarrow & & \\ & & -\infty & & \\ & & \vdots & & \\ & & \log w_1 & & \leftarrow m(a) \\ \cdots & & \vdots & \cdots & \\ & & \log w_2 & & \leftarrow m(b) \\ & & \vdots & & \\ & & -\infty & & \end{pmatrix} \tag{5.45}$$

This means that, if $\mathbf{h}_t$ encodes the state-symbol pair (p, b), the vector $\mathbf{E}\mathbf{h}_t$ will copy the selected column in $\mathbf{E}$ which contains the output weight for all out symbols a of p, i.e., the entry $\mathbf{E}\mathbf{h}_{m(a)}$ contains the weight on the arc $p \xrightarrow{a/w} \circ$. Over the course of an entire input string $\boldsymbol{y}$, these probabilities are simply multiplied as the RNN transitions between different hidden states corresponding to the transitions in the original WFSA $\mathcal{A}$.

[6] The factor $(|\Sigma| + 1)$ comes from the EOS-augmented alphabet.

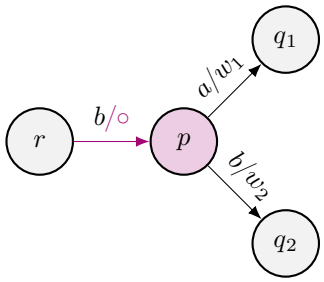


Figure 5.6: An example of a fragment of a WFSA.

For example, for the fragment of a WFSA in Fig. 5.6, the matrix-vector product $\mathbf{E}\mathbf{h}_t$ would take the form

$$\mathbf{E}\mathbf{h}_t = \begin{pmatrix} -\infty \\ \vdots \\ \log w_1 \\ \vdots \\ \log w_2 \\ \vdots \\ -\infty \end{pmatrix} \begin{matrix} \\ \\ \leftarrow m(a) \\ \\ \leftarrow m(b) \\ \\ \\ \end{matrix} \tag{5.46}$$

Formally, let q be the state arrived at by $\mathcal{A}$ after reading in the string $\boldsymbol{y} \in \Sigma^*$. The RNN $\mathcal{R}$ computes the probability $p_{\text{SM}}(y \mid \boldsymbol{y})$ as

$$p_{\text{SM}}(y \mid \boldsymbol{y}) = \text{softmax}(\mathbf{E}\mathcal{Q}(q))_y \tag{5.47}$$

$$= \begin{cases} \dfrac{\exp[\log w]}{\displaystyle\sum_{q \xrightarrow{y'/w'} q'} \exp[\log w'] + \exp[\log \rho(q)]} & \textbf{if } \circ \xrightarrow{y/w} \circ \in \delta \\ 0 & \textbf{otherwise} \end{cases} \tag{5.48}$$

$$= \begin{cases} \dfrac{w}{\displaystyle\sum_{q \xrightarrow{y'/w'} q'} w' \rho(q)} = w & \textbf{if } \circ \xrightarrow{y/w} \circ \in \delta \\ 0 & \textbf{otherwise} \end{cases} \tag{5.49}$$

The last step used the fact that $\mathcal{A}$ is locally normalized, meaning that the denominator is 1. Similarly, it holds that

$$p_{\text{SM}}(\text{EOS} \mid \boldsymbol{y}) = \rho(q), \tag{5.50}$$

where q is the state arrived at by $\mathcal{A}$ after reading in $\boldsymbol{y}$.

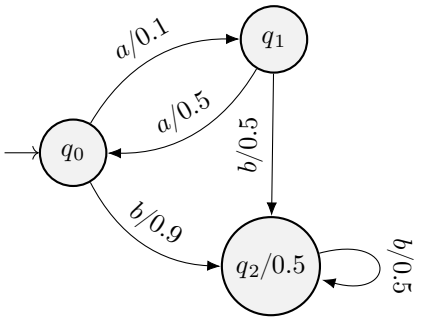


Figure 5.7: The WFSA $\mathcal{A}$.

To show that the RNN $\mathcal{R}$ indeed encodes the same weighted language as $\mathcal{A}$, we consider its locally normalized model. Clearly, the $-\infty$ logits in the conditionals $p_{\text{SM}}(y \mid \boldsymbol{y})$ zero out the probability of any string not in the support of $\mathcal{A}$. Suppose now that $\boldsymbol{y} \in \Sigma^*$ such that $\mathcal{A}(\boldsymbol{y})$. Assume that $\mathcal{A}$ arrives at the state q_t after reading in $\boldsymbol{y}_{\leqslant t}$ and let $\boldsymbol{\pi} = \left(q_I \xrightarrow{y_1/w_1} q_1, \ldots, q_T \xrightarrow{y_T/w_T} q_{T+1}\right)$ be the the $\boldsymbol{y}$-labeled path where q_I is the sole initial state of $\mathcal{A}$. The RNN $\mathcal{R}$ computes the probability $p_{\text{LN}}(\boldsymbol{y})$ with $T = |\boldsymbol{y}|$ as

$$p_{\text{LN}}(\boldsymbol{y}) = p_{\text{SM}}(\text{EOS} \mid \boldsymbol{y}) \prod_{t=1}^{T} p_{\text{SM}}(y \mid \boldsymbol{y}) = \rho(q_T) \prod_{t=1}^{T} w_t = \mathcal{A}(\boldsymbol{y})$$

where we took into account that $\mathcal{A}$ is deterministic and therefore only has one initial state with the initial weight 1. This finishes the proof of the lemma. ■

We now walk through an example of the Minsky construction.

Example 5.1.10. *Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA as shown in Fig. 5.7. Since $\mathcal{A}$ has $|Q| = 3$ states and an alphabet of $|\Sigma| = 2$ symbols, the hidden state of the representing RNN $\mathcal{R}$ will be of dimensionality $3 \cdot 2 = 6$. Assume that the set of state-symbol pairs is ordered as $(q_0, a), (q_0, b), (q_1, a), (q_1, b), (q_2, a), (q_2, b)$. The initial state can be represented (choosing a as the arbitrary "incoming symbol") as*

$$\mathbf{h}_0 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}. \tag{5.51}$$

The recurrent matrix $\mathbf{U}$ *of* $\mathcal{R}$ *is*

$$\mathbf{U} = \begin{pmatrix} -1 & -1 & 1 & 1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}, \tag{5.52}$$

the input matrix $\mathbf{V}$

$$\mathbf{V} = \begin{pmatrix} 1 & -1 \\ -1 & -1 \\ 1 & -1 \\ -1 & -1 \\ -1 & -1 \\ -1 & 1 \end{pmatrix}, \tag{5.53}$$

and the output matrix $\mathbf{E}$ *is*

$$\mathbf{E} = \begin{pmatrix} \log(0.1) & \log(0.1) & \log(0.5) & \log(0.5) & -\infty & -\infty \\ \log(0.9) & \log(0.9) & \log(0.5) & \log(0.5) & \log(0.5) & \log(0.5) \\ -\infty & -\infty & -\infty & -\infty & \log(0.5) & \log(0.5) \end{pmatrix}, \tag{5.54}$$

where the last row corresponds to the symbol EOS*. The target of the* b*-labeled transition from* q_0 *(*$q_0 \xrightarrow{b/0.9} q_2$*) is computed as follows:*

$$\begin{aligned} \mathbf{h}_1 &= H\left(\mathbf{U}\mathbf{h}_0 + \mathbf{V}\mathbf{o}_b\right) \\ &= H\left(\begin{pmatrix} -1 & -1 & 1 & 1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 & -1 \\ -1 & -1 \\ 1 & -1 \\ -1 & -1 \\ -1 & -1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right) \\ &= H\left(\begin{pmatrix} -1 \\ -1 \\ 1 \\ -1 \\ -1 \\ 1 \end{pmatrix} + \begin{pmatrix} -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ 1 \end{pmatrix}\right) = H\left(\begin{pmatrix} -2 \\ -2 \\ 0 \\ -2 \\ -2 \\ 2 \end{pmatrix}\right) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}, \end{aligned}$$

which corresponds exactly the configuration in which $\mathcal{A}$ *is in state* q_2 *which it arrived to by reading in the symbol* b*.*

The probability of the string $\boldsymbol{y} = b$ *under the locally-normalized model induced*

by $\mathcal{R}$ can be computed as

$$
\begin{aligned}
p_{LN}(\boldsymbol{y}) &= p_{LN}(b) = p_{SM}(b \mid \text{BOS})\, p_{SM}(\text{EOS} \mid b) = p_{SM}(b \mid \mathbf{h}_0)\, p_{SM}(\text{EOS} \mid \mathbf{h}_1) \\
&= \operatorname{softmax}(\mathbf{E}\mathbf{h}_0)_b \operatorname{softmax}(\mathbf{E}\mathbf{h}_1)_{\text{EOS}} \\
&= \operatorname{softmax}\left(\begin{pmatrix} \log(0.1) & \cdots & -\infty \\ \log(0.9) & \cdots & \log(0.5) \\ -\infty & \cdots & \log(0.5) \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}\right)_b \cdot \\
&\quad \operatorname{softmax}\left(\begin{pmatrix} \log(0.1) & \cdots & -\infty \\ \log(0.9) & \cdots & \log(0.5) \\ -\infty & \cdots & \log(0.5) \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}\right)_{\text{EOS}} \\
&= \operatorname{softmax}\begin{pmatrix} \log(0.1) \\ \log(0.9) \\ -\infty \end{pmatrix}_b \cdot \operatorname{softmax}\begin{pmatrix} -\infty \\ \log(0.5) \\ \log(0.5) \end{pmatrix}_{\text{EOS}} = 0.9 \cdot 0.5 = 0.45.
\end{aligned}
$$

Constructing a smaller RNN. Lemma 5.1.2 gives a relatively simple construction of an RNN recognizing a weighted regular language. However, the resulting RNN is relatively large, with a hidden state of size linear in the number of states of the (deterministic) WFSA recognizing the language, with the additional multiplicative factor in the size of the alphabet. Note that constructions resulting in smaller RNNs exist, at least for the unweighted case. For example, for an arbitrary WFSA $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$, Dewdney (1977); Alon et al. (1991) present a construction of an RNN with a hidden state of size $\mathcal{O}\left(|Q|^{\frac{3}{4}}\right)$ simulating $\mathcal{A}$, whereas Indyk (1995) provides a construction of an RNN with a hidden state of size $\mathcal{O}\left(|Q|^{\frac{1}{2}}\right)$. The latter is also provably a *lower bound* on the number of neurons required to represent an arbitrary unweighted FSA with a Heaviside-activated recurrent neural network (Indyk, 1995). It is not yet clear if this can be generalized to the weighted case or if Minsky's construction is indeed optimal in this setting. This is quite interesting since one would expect that an RNN with a hidden state of size D can represent up to 2^D individual states (configurations of the D-dimensional vector). However, the form of the transition function with the linear transformation followed by a Heaviside activation limits the number of transition functions that can be represented using D dimensions, resulting in the required exponential increase in the size of the hidden state.

Discussion and the practical applicability of this result. This section showed that Heaviside-activated RNNs are equivalent to WFSAs. This might come as a bit of a surprise considering that we introduced RNNs with the goal

of overcoming some limitations of exactly those models, e.g., the finite context length. However, note that to arrive at this result, we considerately restricted the form of a recurrent neural network. While on the one hand restriction to the Heaviside activation function means that all the RNNs we considered in this section can be implemented and represented in a computer, the RNN sequence models that we usually deal with are much more complex than this analysis allowed for. Furthermore, note that the RNNs in practice do not learn sparse hidden states of the form considered in the construction in the proof of Lemma 5.1.2—indeed, networks with Heaviside activation functions are not trainable with methods discussed in §3.2 as the gradient on the entire parameter space would be either $\mathbf{0}$ or undefined and in this sense, the trained networks would never have such hidden state dynamics. The dynamics of RNNs in practice result in *dense* hidden states, i.e., states in which many dimensions are non-zero. Nonetheless, keep in mind that theoretically, due to the finite-precision nature of our computers, all models we ever consider will be at most finite-state—the differentiating factor between them will be how appropriately to the task they are able to learn the topology (transitions) of the finite-state automaton they represent and how efficiently they are able to learn it.

Turing Completeness of Recurrent Neural Networks

We now turn to the purely theoretical treatment of the expressive capacity of recurrent neural networks in which we take the liberty of making some in practice unrealistic assumptions. Whereas we saw that RNNs with Heaviside activations in a practical setting lie at the bottom of the weighted Chomsky hierarchy, being able to only recognize regular languages, we will see that if we relax some of the assumptions we made, we RNNs jump directly to the top of the hierarchy: they become Turing complete. We start by introducing the saturated sigmoid, one of the building blocks we will use to show this.

Definition 5.1.11 (Saturated Sigmoid)**.** *The* ***saturated sigmoid*** *is defined as*

$$\sigma(x) = \begin{cases} 0 & \textbf{\textit{if}}\ x \leqslant 0 \\ x & \textbf{\textit{if}}\ 0 < x \leqslant 1 \\ 1 & \textbf{\textit{if}}\ x > 1 \end{cases}. \tag{5.55}$$

Intuitively, the saturated sigmoid clips all negative values to 0, all values larger than 1 to 1, and leaves the elements of $[0, 1]$ intact. The graph of this function is shown in Fig. 5.8.

The central result of this subsection is then summarized in the following theorem.

Theorem 5.1.3 (Saturated Sigmoid Elman RNNSMs are Turing complete)**.** *Elman recurrent neural network sequence models with the saturated sigmoid activation functions are Turing complete.*

By the end of this subsection, we will have proven this result by showing that Saturated Sigmoid Elman RNNSMs can encode two-stack pushdown automata

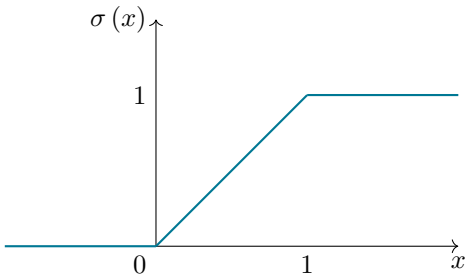


Figure 5.8: The saturated sigmoid.

(cf. Definitions 4.2.53 and 4.2.54). However, before we do that, we will first consider a simpler construction: building on our placement of RNNs onto at least the regular step of the ladder of formal language complexity (cf. Lemma 5.1.2), we will first take one step up the ladder and show that RNNs can simulate a *single-stack* pushdown automaton (cf. Definition 4.2.31). This will help us gain the intuition behind how RNNs can use infinite precision arithmetic to simulate a stack. We will then simply generalize this construction to the two-stack case.

Let us begin by considering the problem of representing a stack—a possibly arbitrarily long sequence of symbols—in a vector of constants size, e.g., the hidden state of a recurrent neural network. For simplicity, but without the loss of generality, assume that we are working with a simple two-letter stack alphabet $\Gamma = \{0, 1\}$. Any stack sequence $\boldsymbol{\gamma}$ will be a member of Γ^*, i.e., a string of 0's and 1's. If we think of the stack *symbols* as *numbers* for a moment, there is a natural correspondence between the possible stack configurations and *numbers* expressed in base 2. By convention, we will represent a string of stack symbols $\boldsymbol{\gamma}$ with numbers after the decimal point, rather than as integers. Assuming infinite precision, we can therefore simply represent each stack configuration as a single number (of course, the stack alphabet does not have to be exactly $\Gamma = \{0, 1\}$—we can always map symbols from any alphabet into their numeric representations in some base large enough to allow for the entire alphabet). Notice that in this case, pushing or popping from the stack can be performed by division and multiplication of the value representing the stack—if we want to push a value $x \in \{0, 1\}$, we can *divide* the current representation (by 2) and append x to the right side of the new representation and if we want to pop any value, we simply have to *multiply* the current representation by 2. This also gives us an idea of how to represent a stack in the hidden state of an RNN: the *entire* stack sequence will simply be represented in a single dimension of the hidden state, and the value stored in the cell will be updated according to the transitions defined by the simulated automaton. Note that, however, the RNN will not only have a single dimension in the hidden state: other dimensions will correspond to some sort of control or configuration values that will be required to control the RNN updates correctly.

In our proofs, we also consider a special case of general pushdown automata,

as defined in Definition 4.2.31: we will use pushdown automata which only consider the *topmost* element of the stack when defining the possible transitions from a configuration and can only push one stack symbol at a time. More formally, this means that in the tuple $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, (q_I, \boldsymbol{\gamma}_I), (q_F, \boldsymbol{\gamma}_F))$, we have that $\delta \subseteq Q \times \Gamma \times (\Sigma \cup \{\varepsilon\}) \times Q \times \Gamma$ rather than the more general $\delta \subseteq Q \times \Gamma^* \times (\Sigma \cup \{\varepsilon\}) \times Q \times \Gamma^*$. Furthermore, we assume that $\boldsymbol{\gamma}_I = \varepsilon$ and $\boldsymbol{\gamma}_F = \varepsilon$, that is, the PDA starts off with an empty stack and has to empty it again to arrive at a final configuration. Note that these restrictions can be done without loss of generality—that is, such pushdown automata are as powerful as the unrestricted versions (Sipser, 2013). With this in mind, we can show that arbitrary precision RNNs are capable of recognizing at least deterministic context-free languages:

Theorem 5.1.4. *Elman recurrent neural networks can recognize deterministic context-free languages.*

Before we continue to the proof of Theorem 5.1.4, let us remark on three simple but important intuitions which will be crucial for understanding the construction of the Elman RNN, both in the single- as well as the two-stack variants of PDAs. Multiple times in the construction, we will be faced with the task of *moving* or copying the value from some dimension i to the dimension j in the vector. The following fact shows how this can be done using simple matrix multiplication with a specific matrix.

Fact 5.1.1 (Copying elements of a vector)**.** *Let $\mathbf{x} \in \mathbb{R}^D$, $i, j \in \{1, \ldots, D\}$, and $\mathbf{M} \in \mathbb{R}^{D \times D}$ such that $\mathbf{M}_{j,:} = \mathbf{e}_i$, where $\mathbf{M}_{j,:}$ denotes the j^{th} row of $\mathbf{M}$ and $\mathbf{e}_i$ denotes the i^{th} basis vector. Then, it holds that $(\mathbf{M}\mathbf{x})_j = x_i$.*

Also, keep in mind that setting the row $\mathbf{M}_{i,:}$ to the zero vector $\mathbf{0}_{\in \mathbb{R}^D}$ sets the entry x_n to 0, i.e., it erases the entry.

Furthermore, we will use the saturated sigmoid function multiple times to *detect* whether a number of dimensions of a vector are set to one at the same time. Given the recurrent dynamics of the Elman RNN (cf. Eq. (5.25)), we can perform this check as follows.

Fact 5.1.2 (Detecting the activation of multiple values in the hidden state)**.** *Let σ be the saturated sigmoid from Definition 5.1.11, $m \in \{1, \ldots, D\}$, $i_1, \ldots, i_m, j \in \{1, \ldots, D\}$, $\mathbf{x} \in \mathbb{R}^D$, $\mathbf{b} \in \mathbb{R}^D$, and $\mathbf{M} \in \mathbb{R}^{D \times D}$ such that*

$$\mathbf{M}_{j,i} = \begin{cases} 1 & \textbf{\textit{if}}\ i \in \{i_1, \ldots, i_m\} \\ 0 & \textbf{\textit{otherwise}} \end{cases}$$

and $b_j = -(m-1)$. Then, it holds that $(\sigma(\mathbf{M}\mathbf{x} + \mathbf{b}))_j = 1$ if and only if $x_{i_k} = 1$ for all $k = 1, \ldots, m$.

Lastly, we will sometimes have to *turn off* certain dimensions of the hidden state if any of the other dimensions are active. Using the dynamics of Elman RNNs and the saturated sigmoid, this can be done as follows.

Fact 5.1.3 (Turning off dimensions in the hidden state)**.** *Let σ be the saturated sigmoid from Definition 5.1.11, $m \in \{1, \ldots, D\}$, $i_1, \ldots, i_m, j \in \{1, \ldots, D\}$, $\mathbf{x} \in \mathbb{R}^D$, $\mathbf{b} \in \mathbb{R}^D$, and $\mathbf{M} \in \mathbb{R}^{D \times D}$ such that*

$$\mathbf{M}_{j,i} = \begin{cases} -1 & \textbf{\textit{if}}\ i \in \{i_1, \ldots, i_m\} \\ 0 & \textbf{\textit{otherwise}} \end{cases}$$

and $b_j = 1$. Then, it holds that $(\sigma(\mathbf{M}\mathbf{x} + \mathbf{b}))_j = 0$ if and only if $x_{i_k} = 1$ for some $k = 1, \ldots, m$.

With these intuitions in mind, we now prove Theorem 5.1.4. Due to the relatively elaborate construction, we limit ourselves to pushdown automata with a two-symbol input alphabet $\Sigma = \{a, b\}$ as well as a two-symbol stack alphabet $\Gamma = \{0, 1\}$. Note, however, that this restriction can be done without the loss of generality, meaning that this is enough to prove the Turing completeness of RNNs in general.[7]

Proof. We show this by constructing, for a given deterministic pushdown automaton recognizing a deterministic context-free language an Elman RNN simulating the steps performed by $\mathcal{P}$. Let $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, (q_I, \boldsymbol{\gamma}_I), (q_F, \boldsymbol{\gamma}_F))$ then be a deterministic pushdown automaton. We now define the parameters of the RNN $\mathcal{R} = (\Sigma, D, \mathbf{U}, \mathbf{V}, \mathbf{E}, \mathbf{b}_h, \mathbf{h}_0)$ such that the updates to $\mathcal{R}$'s hidden state will correspond to the configuration changes in $\mathcal{P}$.

The construction is more involved than the one in Minsky's theorem (cf. Lemma 5.1.2). We, therefore, first intuitively describe the semantics of the different *components* of the hidden state of the RNN. Then, we describe the submatrices of the parameters $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{b}$ that *control* these components of the vector. The hidden state $\mathbf{h}$ of the RNN will altogether have *five* components.

- **Component 1: Data component**: This component, consisting of three cells, will contain the actual numerical representation of the stack, $\texttt{STACK}$, as well as two additional "buffer" cells, $\texttt{BUFF}_1$ and $\texttt{BUFF}_2$, which will be used for intermediate copies of the stack values during the computation of the new state.

- **Component 2: Top of stack component**: This component contains three cells, each corresponding to a flag denoting that *(a)* the stack is empty ($\texttt{STACK}_\varepsilon$), *(b)* the top element of the stack is a 0 ($\texttt{STACK}_0$), or *(c)* the top element of the stack is a 1 ($\texttt{STACK}_1$).

- **Component 3: Configuration component**: This component encodes the current configuration of the stack (Component 2) together with the current input symbol. Note that, while we assume that the input PDA works with the two-symbol alphabet $\Sigma = \{a, b\}$, the sequence model defined

[7] To simulate an arbitrary Turing machine with a machine with the binary alphabet, we simply have to encode each of the finitely-many symbols of the simulated machine using binary encoding.

by the RNN requires an EOS symbol to be able to terminate generation (cf. Eq. (2.41)): $\mathcal{R}$, therefore, defines the conditional probabilities over the set $\overline{\Sigma} = \{a, b, \text{EOS}\}$. With this, there are nine possible configurations (y, γ) for $\gamma \in \{\varepsilon, 0, 1\}$ and $y \in \{a, b, \text{EOS}\}$, meaning that there are nine cells in this configuration, $\texttt{CONF}_{\gamma,y}$, each corresponding to one of these configurations.

- **Component 4: Computation component**: This component contains four cells in which the computation of the next value of the stack is computed. There are four cells $\texttt{OP}_{\texttt{action},\gamma}$ because *all* possible actions (PUSH 0, PUSH 1, POP 0, POP 1, and NO-OP) are performed simultaneously, and only the correct one is copied into the data component (Component 1) in the end.

- **Component 5: Acceptance component**: This component contains a single cell, ACCEPT, signaling whether the RNN accepts the string $\boldsymbol{y}$ after reading in the input $\boldsymbol{y}$ EOS.

Altogether, the hidden state of $\mathcal{R}$ contains $3+3+9+5+1 = 21$ dimensions. The *initial hidden state* $\mathbf{h}_0$ is a vector with a single non-zero component, whose value is 1: the cell $\texttt{STACK}_\varepsilon$ since we assume that the stack of the simulated automaton is empty at the beginning of the execution. We now intuitively describe the dynamics that these components define.

The full update step of the network. The RNN will compute the next hidden state corresponding to the new stack configuration by applying the Elman update rule (cf. Eq. (5.25)) four times to complete four discrete sub-steps of the computation. We first define

$$\mathbf{h}_{t+1}^{(1)} \stackrel{\text{def}}{=} \mathbf{h}_t \tag{5.56}$$

and

$$\mathbf{h}_{t+1}^{(n)} \stackrel{\text{def}}{=} \sigma\left(\mathbf{U}\mathbf{h}_{t+1}^{(n-1)} + \mathbf{V}\mathbf{e}(y_t) + \mathbf{b}_h\right) \tag{5.57}$$

for $n = 2, 3, 4$. Then

$$\mathbf{h}_{t+1} \stackrel{\text{def}}{=} \mathbf{h}_{t+1}^{(4)}. \tag{5.58}$$

Intuitively, each of the four stages of computation of the actual next hidden state "detects" some parts of the pattern contributing to the transition in the pushdown automaton. We describe those patterns next intuitively before talking about the submatrices (or subvectors) of the RNN parameters corresponding to the specific parts that update the individual components of the hidden state.

Data component. The cells of the data component form a queue of three components: the STACK cell forms the head of the queue, followed by $\texttt{BUFF}_1$ and $\texttt{BUFF}_2$. The values in the cells are updated at each execution of Eq. (5.57) by moving the currently stored values into the next cell in the queue. By doing so, the entry in $\texttt{BUFF}_2$ gets discarded. The value of STACK is copied from the cells of the *computation* component by *summing* them. We will see later that at any point of the computation (when it matters), only one of the computation components

will be non-zero, which means that the summation simply corresponds to copying the non-zero computation component. All these operations can be performed by matrix multiplication outlined in Fact 5.1.1.

Encoding the stack sequence. While we outlined a possible encoding of a stack sequence above, the encoding we use in this construction is a bit different. Remember that for a stack sequence $\boldsymbol{\gamma} \in \Gamma^*$ of length N, the *right-most* symbol γ_N denotes the top of the stack. We encode the stack sequence $\boldsymbol{\gamma} \in \Gamma^*$ as follows:

$$\texttt{rep}\left(\gamma_1 \ldots \gamma_N\right) \stackrel{\text{def}}{=} \sum_{n=1}^{N} \texttt{digit}\left(\gamma_n\right) 10^{N-n-1} \tag{5.59}$$

where $\texttt{digit}\left(\gamma\right) \stackrel{\text{def}}{=} \begin{cases} 1 & \textbf{if } \gamma = 0 \\ 3 & \textbf{otherwise} \end{cases}$.

Example 5.1.11. *For example, the stack sequence* $\boldsymbol{\gamma} = 00110111$ *would be represented with* $\texttt{rep}\left(00110111\right) = 0.33313311$. *Notice the "opposite orientation" of the two strings: the top of the stack in* γ *is the right-most symbol, while it is the left-most digit in the numerical representation.*

Note that the digits 1 and 3 in the definition of $\texttt{digit}\left(\cdot\right)$ are chosen somewhat arbitrarily—the encoding could also have been chosen differently. Similarly, a different (non-decimal) base could have been chosen.

Top of stack component. As mentioned, the $\texttt{STACK}_{\varepsilon}$ cell of this component is one of the two cells set to 1 in the initial state of the RNN. The individual cells of this component then get updated according to the top symbol on the stack encoded in the STACK cell by taking into account how the stack is represented by STACK. Specifically, the parameters of the RNN are defined such that $\mathbf{h}_{\texttt{STACK}_{\varepsilon}} = 1$ if previously $\mathbf{h}_{\texttt{STACK}} = 0$, $\mathbf{h}_{\texttt{STACK}_0} = 1$ if previously $\mathbf{h}_{\texttt{STACK}} = 0.1\ldots$, and $\mathbf{h}_{\texttt{STACK}_1} = 1$ if previously $\mathbf{h}_{\texttt{STACK}} = 0.3\ldots$.

Configuration component. The cells of the configuration component combine the pattern captured by the top of the stack component with the input symbol at the current time step to activate only the appropriate cell $\texttt{CONF}_{\gamma,y}$. This can be done by incorporating the information from the top of the stack component with the information about the current input symbol from $\mathbf{V}\mathbf{o}_{y_t}$. More precisely, the parameters of $\mathcal{R}$ are set such that $\mathbf{h}_{\texttt{CONF}_{\gamma,y}} = 1$ if at the previous one of the four sub-steps of the computation of the next hidden state, $\mathbf{h}_{\texttt{STACK}_{\gamma}} = 1$ *and* the input symbol is y.

Computation component. The computation component contains the cells in which the results of all the possible actions on the stack are executed. The parameters of the computation component are set such that, given that the

previous stack configuration is $\mathbf{h}_{\texttt{STACK}} = x_1 x_2 \dots x_N$ and the input symbol is y, cells of the computation component are set as

$$\mathbf{h}_{\texttt{OP}_{\texttt{POP},\gamma}} = x_2 \dots x_N$$
$$\mathbf{h}_{\texttt{OP}_{\texttt{PUSH},\gamma}} = \texttt{digit}\,(y) x_1 \dots x_N.$$

Acceptance component. The cell in the acceptance component is activated if and only if the current input symbol is EOS (denoting the end of the string whose recognition should be determined) and the stack is empty, i.e., the $\texttt{STACK}_\varepsilon$ cell is activated.

More precisely, the dynamics described here are implemented by the four steps of the hidden state update as follows (where $\mathbf{h}_t^{(1)} = \mathbf{h}_t$).

- In *phase 1*, the configuration of the stack is determined by setting the top of the stack component in $\mathbf{h}_t^{(2)}$.
- In *phase 2*, the configuration of the stack and the input symbol are combined by setting the configuration component in $\mathbf{h}_t^{(3)}$.
- In *phase 3* all possible operations on the stack are performed in the computation component, and, at the same time, the results of all invalid operations (only one operation is valid at each time step due to the deterministic nature of $\mathcal{P}$) are zeroed-out in $\mathbf{h}_t^{(4)}$. This is done by setting the entries of the recurrence matrix $\mathbf{U}$ such that only the valid action is not zeroed out.
- In *phase 4* the result of the executed operations (only one of which is non-zero) is copied over to the STACK cell in the hidden state in $\mathbf{h}_{t+1}$.

Having defined the intuition behind the dynamics of the hidden state updates, we now formally define how the parameters of the RNN are set to enable them. Whenever an entry of a matrix or vector is not set explicitly, it is assumed that it is 0 (that is, we only explicitly set the non-zero values). Again, we define them for each component in turn.

The data component. The values of the parameters in the data component are set as follows.

$$U_{\texttt{BUFF}_1,\texttt{STACK}} = 1 \tag{5.60}$$
$$U_{\texttt{BUFF}_2,\texttt{BUFF}_1} = 1 \tag{5.61}$$
$$U_{\texttt{STACK},\texttt{OP}_{\texttt{PUSH},0}} = U_{\texttt{STACK},\texttt{OP}_{\texttt{PUSH},1}} = U_{\texttt{STACK},\texttt{OP}_{\texttt{POP},0}} = U_{\texttt{STACK},\texttt{OP}_{\texttt{POP},1}} = 1 \tag{5.62}$$

The first two elements correspond to moving the values to the next element in the data component queue, while the entries in the last row correspond to *summing up* the values from the computation component to move them into the stack cell after the computation has been completed. Note that, of course, the elements of the computation component are *always* summed up and written

in the STACK cell, no matter what the values there are. However, the division of the computation of the next hidden state into phases ensures that *when it matters*, i.e., after the third phase, there is only a single computation component that is non-zero, and that one is copied into the STACK component in the fourth computation sub-step. All other parameters (in $\mathbf{V}$ and $\mathbf{b}_h$) are 0.

The top of the stack component. The parameters setting the top of the stack component are set as follows:

$$U_{\mathsf{STACK}_\varepsilon,\mathsf{STACK}} = -10 \tag{5.63}$$
$$U_{\mathsf{STACK}_0,\mathsf{STACK}} = -10 \tag{5.64}$$
$$U_{\mathsf{STACK}_1,\mathsf{STACK}} = 10 \tag{5.65}$$
$$b_{\mathsf{STACK}_\varepsilon} = 1 \tag{5.66}$$
$$b_{\mathsf{STACK}_0} = 3 \tag{5.67}$$
$$b_{\mathsf{STACK}_1} = -2. \tag{5.68}$$

Other parameters ($\mathbf{V}$) are 0. The reasoning behind these parameters is the following. The cell STACK contains the numeric encoding of the stack content. We distinguish three cases.

- If the stack is *empty*, $\mathbf{h}_{\mathsf{STACK}} = 0$. Therefore, using the parameters above, the value of the cell STACK_1 after the sub-step update will be 0, while the cells $\mathsf{STACK}_\varepsilon$ and STACK_0 will be 1 due to the positive bias term. This might not be what you would expect—it might seem like, in this case, this step erroneously signals both an empty stack and a stack whose top component is 0. This, however, is corrected for in the configuration component, as we discuss below.

- If the top of the stack is the symbol 0, $\mathbf{h}_{\mathsf{STACK}} = 0.1\ldots$. This means that $10 \cdot \mathbf{h}_{\mathsf{STACK}} \leqslant 1$ and, therefore, after the update rule application, $\mathbf{h}_{\mathsf{STACK}_1} = 0$. It is easy to see that the setting of the parameters also implies $\mathbf{h}_{\mathsf{STACK}_\varepsilon} = 0$. However, since $-10 \cdot \mathbf{h}_{\mathsf{STACK}} \geqslant -2$, we have that $\mathbf{h}_{\mathsf{STACK}_0} = 1$.

- Lastly, if the top of the stack is the symbol 1, $\mathbf{h}_{\mathsf{STACK}} = 0.3\ldots$. Therefore, $10 \cdot \mathbf{h}_{\mathsf{STACK}} \geqslant 3$, meaning that after the update rule application, $\mathbf{h}_{\mathsf{STACK}_1} = 1$. Again, it is easy to see that the setting of the parameters also implies $\mathbf{h}_{\mathsf{STACK}_\varepsilon} = 0$. On the other hand, since $-10 \cdot \mathbf{h}_{\mathsf{STACK}} \leqslant -3$, it also holds that $\mathbf{h}_{\mathsf{STACK}_0} = 0$.

The configuration component. The configuration component is composed of the most cells of any component:

$$U_{\mathsf{CONF}_{\gamma,y},\mathsf{STACK}_\gamma} = 1 \textbf{ for } \gamma \in \{\varepsilon, 0, 1\}, y \in \{\text{EOS}, a, b\} \tag{5.69}$$
$$U_{\mathsf{CONF}_{\gamma,0},\mathsf{STACK}_\varepsilon} = -1 \textbf{ for } y \in \{\text{EOS}, a, b\} \tag{5.70}$$
$$V_{\mathsf{CONF}_{\gamma,y},m(y)} = 1 \textbf{ for } \gamma \in \{\varepsilon, 0, 1\}, y \in \{\text{EOS}, a, b\} \tag{5.71}$$
$$b_{\mathsf{CONF}_{\gamma,y}} = -1 \textbf{ for } \gamma \in \{\varepsilon, 0, 1\}, y \in \{\text{EOS}, a, b\} \tag{5.72}$$

Here, the first, third, and fourth terms together ensure that the cell $\texttt{CONF}_{\gamma,y}$ is activated if the current input symbol is a ($V_{\texttt{CONF}_{\gamma,y},m(y)}$) and the top of the stack is γ ($U_{\texttt{CONF}_{\gamma,y},\texttt{STACK}_\gamma}$). $b_{\texttt{CONF}_{\gamma,y}}$ ensures that both conditions have to be met. The second term, $U_{\texttt{CONF}_{0,y},\texttt{STACK}_\varepsilon}$, on the other hand, takes care of an edge case: as shown above, $b_{\texttt{STACK}_0} = 0$, which means that $\texttt{STACK}_0$ is, by default, set to 1. The negative weight $U_{\texttt{CONF}_{0,y},\texttt{STACK}_\varepsilon} = -1$ ensures that, if the stack is indeed empty, the effect of this default value is "canceled out", i.e., the configuration cell is not activated by mistake.

The computation component. This is the most complicated component. The computation components are manipulated with the following parameters:

$$U_{\texttt{PUSH0},\texttt{BUFF}_2} = U_{\texttt{PUSH1},\texttt{BUFF}_2} = \frac{1}{10} \tag{5.73}$$

$$U_{\texttt{POP0},\texttt{BUFF}_2} = U_{\texttt{POP1},\texttt{BUFF}_2} = 10 \tag{5.74}$$

$$U_{\texttt{NO-OP},\texttt{BUFF}_2} = 1 \tag{5.75}$$

$$U_{\texttt{A},\texttt{CONF}_{\gamma,y}} = -10 \textbf{ for } \texttt{A} \in \{\texttt{OP}_{\texttt{PUSH},0}, \texttt{OP}_{\texttt{PUSH},1}, \texttt{OP}_{\texttt{POP},0}, \texttt{OP}_{\texttt{POP},1}, \texttt{OP}_{\texttt{NO-OP}}\} \quad \gamma \in \{0,1\}, y \in \{a,b\} \tag{5.76}$$

$$U_{\texttt{OP}_{\texttt{A},\gamma'},\texttt{CONF}_{\gamma,y}} = 0 \textbf{ for } q \xrightarrow{y,\gamma\rightarrow\gamma'} q \in \delta, \quad \texttt{A} \in \{\texttt{OP}_{\texttt{PUSH},0}, \texttt{OP}_{\texttt{PUSH},1}, \texttt{OP}_{\texttt{POP},0}, \texttt{OP}_{\texttt{POP},1}, \texttt{OP}_{\texttt{NO-OP}}\} \tag{5.77}$$

$$U_{\texttt{OP}_{\texttt{NO-OP}},\texttt{CONF}_{\gamma,\text{EOS}}} = 0 \textbf{ for } \gamma \in \{\varepsilon, 0, 1\} \tag{5.78}$$

$$b_{\texttt{OP}_{\texttt{PUSH},0}} = \frac{1}{10} \tag{5.79}$$

$$b_{\texttt{OP}_{\texttt{PUSH},1}} = \frac{3}{10} \tag{5.80}$$

$$b_{\texttt{OP}_{\texttt{POP},0}} = -1 \tag{5.81}$$

$$b_{\texttt{OP}_{\texttt{POP},1}} = -3 \tag{5.82}$$

$$b_{\texttt{OP}_{\texttt{NO-OP}}} = 0 \tag{5.83}$$

The first three parameters above concern *copying* the value of the stack encoded by the previous hidden state into the computation component and preparing it for modification. They work together with the corresponding entries in the bias vector $\mathbf{b}_h$. For example, a value can be *pushed* onto the stack by dividing the value of the stack encoding by 10 and adding either 0.1 or 0.3, depending on whether 0 or 1 is being pushed. This is encoded by the first setting above and $b_{\texttt{OP}_{\texttt{PUSH},0}}$ and $b_{\texttt{OP}_{\texttt{PUSH},1}}$. Similarly, a value can be popped from the stack by multiplying the stack encoding with 10 and then subtracting the appropriate value according to the bias entry. The NO-OP action is implemented simply by copying the values of the stack into its cell. The remaining three parameter settings above ensure that, after executing all possible stack actions, *only the appropriate computation* is kept and all others are zeroed out. The fourth row above ensures that "by default", all computation cells are reset to 0 after every update. However, the next row "removes" the negative weights (sets them to 0)

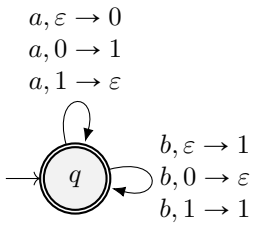


Figure 5.9: The single-stack pushdown automaton $\mathcal{P}$.

for the changes in the configuration which correspond to the *valid transitions*, or valid actions, in the pushdown automaton. That is, setting those values of the matrix $\mathbf{U}$ to 0 disables "erasing" the entry $\mathtt{OP}_{\mathtt{A},\gamma'}$ in the hidden state by the configuration $\mathtt{CONF}_{\gamma,y}$ *if* the transition from the configuration with the top of the stack γ to γ' with the action $\mathtt{A}$ upon reading y is encoded by the original automaton. The last remaining row simply ensures that reading in the EOS symbol results in the $\mathtt{NO\text{-}OP}$ action being executed (EOS actions are not encoded by the original pushdown automaton).

The acceptance component. Lastly, the acceptance component is controlled by the following parameters:

$$U_{\mathtt{ACCEPT},\mathtt{A}} = -10 \text{ for } \mathtt{A} \in \{\mathtt{OP}_{\mathtt{PUSH},0}, \mathtt{OP}_{\mathtt{PUSH},1}, \mathtt{OP}_{\mathtt{POP},0}, \mathtt{OP}_{\mathtt{POP},1}, \mathtt{OP}_{\mathtt{NO\text{-}OP}}\} \tag{5.84}$$

$$U_{\mathtt{ACCEPT},\mathtt{CONF}_{\gamma,y}} = -10 \text{ for } \gamma \in \{\varepsilon, 0, 1\}, y \in \{\text{EOS}, a, b\} \tag{5.85}$$

$$b_{\mathtt{ACCEPT}} = 1 \tag{5.86}$$

The entry $b_{\mathtt{ACCEPT}}$ ensures that, by default, the value of $\mathtt{ACCEPT}$ is set to 1. However, the other parameters ensure that, as soon as any part of the configuration is not compatible with the acceptance state (the read symbol is not EOS or the stack is not empty), the acceptance bit is turned off.

A full proof of the theorem would now require us to show formally that the update rule Eq. (5.58) results in the correct transitions in the PDA. We, however, leave the proof with the intuitive reasoning behind the setting of the parameters and leave this as an exercise for the reader. The proof is also demonstrated in the `python` implementation of the constructions here: https://github.com/rycolab/rnn-turing-completeness. ■

The construction described in the proof of Theorem 5.1.4 is demonstrated in the following example.

Example 5.1.12. *Let $\mathcal{P}$ be a single-stack PDA presented in Fig. 5.9. We now simulate the recognition of the string $\boldsymbol{y} = ab$, which is accepted by $\mathcal{P}$. The initial state $\mathbf{h}_0$ has a single non-zero cell, $\mathtt{STACK}_\varepsilon$. The four phases of the processing of the first input symbol a are shown in Tab. 5.1. The four phases of the processing of the second input symbol b are shown in Tab. 5.2.*

	Initial state	Phase 1	Phase 2	Phase 3	Phase 4
STACK	0	0	$\frac{2}{5}$	$\frac{2}{5}$	$\frac{1}{10}$
BUFF_1	0	0	0	$\frac{2}{5}$	$\frac{2}{5}$
BUFF_2	0	0	0	0	$\frac{2}{5}$
$\mathsf{STACK}_\varepsilon$	1	1	1	0	0
STACK_0	0	1	1	0	0
STACK_1	0	0	0	1	1
$\mathsf{CONF}_{\mathrm{EOS},a}$	0	0	1	1	0
$\mathsf{CONF}_{\mathrm{EOS},b}$	0	0	0	0	0
$\mathsf{CONF}_{0,a}$	0	0	0	0	0
$\mathsf{CONF}_{0,b}$	0	0	0	0	0
$\mathsf{CONF}_{1,a}$	0	0	0	0	1
$\mathsf{CONF}_{1,b}$	0	0	0	0	0
$\mathsf{CONF}_{\varepsilon,\mathrm{EOS}}$	0	0	0	0	0
$\mathsf{CONF}_{0,\mathrm{EOS}}$	0	0	0	0	0
$\mathsf{CONF}_{1,\mathrm{EOS}}$	0	0	0	0	0
$\mathsf{OP}_{\mathsf{PUSH},0}$	0	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$
$\mathsf{OP}_{\mathsf{PUSH},1}$	0	$\frac{3}{10}$	$\frac{3}{10}$	0	0
$\mathsf{OP}_{\mathsf{POP},0}$	0	0	0	0	0
$\mathsf{OP}_{\mathsf{POP},1}$	0	0	0	0	0
$\mathsf{OP}_{\mathsf{NO\text{-}OP}}$	0	0	0	0	0
ACCEPT	0	1	0	0	0

Table 5.1: The simulation of the processing of the first symbol a by the RNN simulating the PDA in Fig. 5.9. After the fourth phase, the stack cell contains the encoding of the stack as 0.1.

	Initial state	Phase 1	Phase 2	Phase 3	Phase 4
STACK	1/10	1/10	1	2/5	0
BUFF_1	2/5	1/10	1/10	1	2/5
BUFF_2	2/5	2/5	1/10	1/10	1
STACK_ε	0	0	0	0	0
STACK_0	0	1	1	0	0
STACK_1	1	0	0	1	1
$\text{CONF}_{\text{EOS},a}$	0	0	0	0	0
$\text{CONF}_{\text{EOS},b}$	0	0	0	0	0
$\text{CONF}_{0,a}$	0	0	0	0	0
$\text{CONF}_{0,b}$	0	0	1	1	0
$\text{CONF}_{1,a}$	1	0	0	0	0
$\text{CONF}_{1,b}$	0	1	0	0	1
$\text{CONF}_{\varepsilon,\text{EOS}}$	0	0	0	0	0
$\text{CONF}_{0,\text{EOS}}$	0	0	0	0	0
$\text{CONF}_{1,\text{EOS}}$	0	0	0	0	0
$\text{OP}_{\text{PUSH},0}$	1/10	0	0	0	0
$\text{OP}_{\text{PUSH},1}$	0	0	0	0	0
$\text{OP}_{\text{POP},0}$	0	0	0	0	0
$\text{OP}_{\text{POP},1}$	0	1	0	0	0
$\text{OP}_{\text{NO-OP}}$	0	0	2/5	0	0
ACCEPT	0	0	0	0	0

Table 5.2: The simulation of the processing of the second symbol b by the RNN simulating the PDA in Fig. 5.9. After the fourth phase, the stack cell contains the encoding of the empty stack.

Theorem 5.1.4 shows that Elman RNNs are theoretically at least as expressive as deterministic CFGs. We now return to the main result of this subsection: the Turing completeness of RNNs. Luckily, Theorem 5.1.4 gets us most of the way there! Recall that by **??**, *two-stack* PDAs are Turing complete. We make use of this fact by generalizing the construction in the proof of Theorem 5.1.4 to the two-stack case. This will prove that RNNs can in fact simulate any Turing machine, and are, therefore, Turing complete.

Lemma 5.1.3. *Let* $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, (q_I, \boldsymbol{\gamma}_I, \boldsymbol{\sigma}_I), (q_F, \boldsymbol{\gamma}_F, \boldsymbol{\sigma}_F))$ *be a two-stack pushdown automaton. Then, there exists an Elman RNN* $\mathcal{R}$ *simulating* $\mathcal{P}$*, i.e.,* $L(\mathcal{R}) = L(\mathcal{P})$*.*

Proof. Again, given a two-stack PDA $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, (q_I, \boldsymbol{\gamma}_I, \boldsymbol{\sigma}_I), (q_F, \boldsymbol{\gamma}_F, \boldsymbol{\sigma}_F))$ with $\Sigma = \{a, b\}$, $\Gamma_1 = \{0, 1\}$, and $\Gamma_2 = \{0, 1\}$, we construct an Elman RNN $\mathcal{R} = (\Sigma, D, \mathbf{U}, \mathbf{V}, \mathbf{E}, \mathbf{b}_h, \mathbf{h}_0)$ which recognizes the same language as $\mathcal{P}$.

The hidden state of $\mathcal{R}$ will contain the same components as in the proof of Theorem 5.1.4. Moreover, their dynamics will be exactly the same—they will simply be *larger* to account for more possible configurations of the two stacks together. For example, the top of the stack component will now consist of cells $\texttt{STACK}_{\gamma_1\gamma_2}$ for $\gamma_1 \in \Gamma_1$ and $\gamma_2 \in \Gamma_2$ flagging that the top symbol on stack 1 is γ_1 and the top symbol of stack 2 is γ_2. Furthermore, the configuration component will contain cells of the form $\texttt{OP}_{\text{action},\gamma_1\gamma_2}$ with an analogous interpretation. Lastly, all the computation and data component cells would be duplicated (with one sub-component for each of the stacks), whereas the acceptance component stays the same. We now again describe these components and their dynamics intuitively, whenever there is any difference to the single-stack version.

Data component. Instead of having a *single* queue of data cells, $\mathcal{R}$ now has *two* queues, one for each of the stacks. The first queue will be formed by $\texttt{STACK}_1$, $\texttt{BUFF}_{11}$, and STACK21, and the second one by $\texttt{STACK}_2$, $\texttt{BUFF}_{12}$, and STACK22. Each of the queues acts exactly the same as in the single-stack version, and they act independently based on the computations done in the computation cells of the respective stacks. Each of the stacks is also encoded as a numeric sequence in the same way as in the single-stack version.

Top of stack component. Again, $\mathcal{R}$ starts in an initial state in which the cell $\texttt{STACK}_{\varepsilon\varepsilon}$ is 1 and all others are 0. The individual cells of this component then get updated according to the top symbols on the stacks encoded in the $\texttt{STACK}_1$ and $\texttt{STACK}_2$ cells.

Configuration component. The cells of the configuration component combine the pattern captured by the top of *both* stack components with the input symbol at the current time step to activate only the appropriate cell $\texttt{CONF}_{\gamma_1\gamma_2,y}$.

Computation component. Again, the computation component contains the cells in which the results of all the possible actions on both stacks are executed.

They execute the actions on both stacks independently.

Acceptance component. The acceptance component functions identically to the single-stack case.

Using these components, the RNN then transitions between the phases exactly like in the single-stack case. We leave the specifications of the matrix parameter values to the reader. They again follow those presented in the single-stack case but treat the transitions and configurations of both stacks.

■

This concludes our investigation of the formal properties of recurrent neural language models. The sequential nature of the architecture and the relatively simple transition functions in the vanilla RNN architectures made the link to automata from formal language theory relatively straightforward, which allowed relatively strong theoretical insights. However, it was exactly this sequential nature and the issues associated with it of RNNs that eventually led to them being overtaken by another neural architecture, which is now at the core of most if not all, modern state-of-the-art language models: the transformer.[8] We introduce them and discuss their theoretical underpinnings in the next section.

[8]We will not discuss the issues with the training speed and parallelization of RNNs in detail. Some of these issues will be highlighted in the latter parts of the course.

Index

I

J

K

L

M

N

O

P

R

Bibliography

Steven Abney, David McAllester, and Fernando Pereira. 1999. Relating probabilistic grammars and automata. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*, pages 542–549, College Park, Maryland, USA. Association for Computational Linguistics.

Cyril Allauzen and Mehryar Mohri. 2003. Efficient algorithms for testing the twins property. *J. Autom. Lang. Comb.*, 8:117–144.

Noga Alon, A. K. Dewdney, and Teunis J. Ott. 1991. Efficient simulation of finite automata by neural nets. *J. ACM*, 38(2):495–514.

Stéphane Aroca-Ouellette and Frank Rudzicz. 2020. On Losses for Modern Language Models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4970–4981, Online. Association for Computational Linguistics.

Jean-Michel Autebert, Jean Berstel, and Luc Boasson. 1997. Context-free languages and pushdown automata. *Handbook of Formal Languages: Volume 1 Word, Language, Grammar*, pages 111–174.

Enes Avcu, Chihiro Shibata, and Jeffrey Heinz. 2017. Subregular complexity and deep learning. *ArXiv*, abs/1705.05940.

Anton Bakhtin, Yuntian Deng, Sam Gross, Myle Ott, Marc'Aurelio Ranzato, and Arthur Szlam. 2021. Residual energy-based models for text. *Journal of Machine Learning Research*, 22(40):1–41.

Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*, volume 28.

Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828. Cite arxiv:1206.5538.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, Christian Jauvin, Jauvinciro Umontreal Ca, Jaz Kandola, Thomas Hofmann, Tomaso Poggio, and John Shawe-Taylor. 2003. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155.

Julian Besag. 1975. Statistical analysis of non-lattice data. *Journal of the Royal Statistical Society. Series D (The Statistician)*, 24(3):179–195.

Patrick Billingsley. 1995. *Probability and Measure*, 3rd edition. Wiley.

Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

Mathieu Blondel, Andre Martins, and Vlad Niculae. 2019. Learning classifiers with fenchel-young losses: Generalized entropies, margins, and algorithms. In *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics*, volume 89 of *Proceedings of Machine Learning Research*, pages 606–615. PMLR.

L. Boltzmann. 1868. *Studien über das Gleichgewicht der lebendigen Kraft zwischen bewegten materiellen Punkten: vorgelegt in der Sitzung am 8. October 1868*. k. und k. Hof- und Staatsdr.

T.L. Booth and R.A. Thompson. 1973. Applying probability measures to abstract languages. *IEEE Transactions on Computers*, C-22(5):442–450.

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

Alexandra Butoi, Brian DuSell, Tim Vieira, Ryan Cotterell, and David Chiang. 2022. Algorithms for weighted pushdown automata.

Stanley F. Chen and Joshua Goodman. 1996. An empirical study of smoothing techniques for language modeling. In *34th Annual Meeting of the Association for Computational Linguistics*, pages 310–318, Santa Cruz, California, USA. Association for Computational Linguistics.

Yining Chen, Sorcha Gilroy, Andreas Maletti, Jonathan May, and Kevin Knight. 2018. Recurrent neural networks as weighted language recognizers. *NAACL HLT 2018 - 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, 1:2261–2271.

Zhiyi Chi. 1999. Statistical properties of probabilistic context-free grammars. *Computational Linguistics*, 25(1):131–160.

Zhiyi Chi and Stuart Geman. 1998. Estimation of probabilistic context-free grammars. *Computational Linguistics*, 24(2):299–305.

N. Chomsky and M.P. Schützenberger. 1963. The algebraic theory of context-free languages. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, volume 35 of *Studies in Logic and the Foundations of Mathematics*, pages 118–161. Elsevier.

Noam Chomsky. 1959. On certain formal properties of grammars. *Information and Control*, 2(2):137–167.

Noam Chomsky. 1965. *Aspects of the Theory of Syntax*, 50 edition. The MIT Press.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. 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 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

A. K. Dewdney. 1977. Threshold matrices and the state assignment problem for neural nets. pages 227–245.

Jesse Dodge, Gabriel Ilharco, Roy Schwartz, Ali Farhadi, Hannaneh Hajishirzi, and Noah A. Smith. 2020. Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping. *CoRR*, abs/2002.06305.

Li Du, Lucas Torroba Hennigen, Tiago Pimentel, Clara Meister, Jason Eisner, and Ryan Cotterell. 2022. A measure-theoretic characterization of tight language models.

John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12(null):2121–2159.

Rick Durrett. 2019. *Probability: Theory and Examples*, 5$^{\text{th}}$ edition. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge university press.

Jason Eisner. 2016. Inside-outside and forward-backward algorithms are just backprop (tutorial paper). In *Proceedings of the Workshop on Structured Prediction for NLP*, pages 1–17, Austin, TX. Association for Computational Linguistics.

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

W. G. Gibbs. 1902. *Elementary Principles in Statistical Mechanics*. Charles Scribner's Sons.

Glorot, Xavier and Bengio, Yoshua. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy. PMLR.

Chengyue Gong, Di He, Xu Tan, Tao Qin, Liwei Wang, and Tie-Yan Liu. 2018. FRAGE: Frequency-Agnostic word representation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 1341–1352, Red Hook, NY, USA. Curran Associates Inc.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Andreas Griewank and Andrea Walther. 2008. *Principles and Techniques of Algorithmic Differentiation*. SIAM.

Charles M. Grinstead and J. Laurie Snell. 1997. *Introduction to Probability*, 2nd revised edition. American Mathematical Society.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2001. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034.

Elad Hoffer, Itay Hubara, and Daniel Soudry. 2017. Train longer, generalize better: Closing the generalization gap in large batch training of neural networks. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 1729—-1739, Red Hook, NY, USA. Curran Associates Inc.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

Roger A. Horn and Charles R. Johnson. 2012. *Matrix Analysis*, 2nd edition. Cambridge University Press.

Ferenc Huszár. 2015. How (not) to Train your Generative Model: Scheduled Sampling, Likelihood, Adversary? *CoRR*, abs/1511.05101.

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

P. Indyk. 1995. Optimal simulation of automata by neural nets. In *STACS 95*, pages 337–348, Berlin, Heidelberg. Springer Berlin Heidelberg.

Gerhard Jäger and James Rogers. 2012. Formal language theory: Refining the chomsky hierarchy. *Philos Trans R Soc Lond B Biol Sci*, 367(1598):1956–1970.

E. T. Jaynes. 1957. Information theory and statistical mechanics. *Phys. Rev.*, 106:620–630.

Lifeng Jin, Finale Doshi-Velez, Timothy Miller, William Schuler, and Lane Schwartz. 2018. Depth-bounding is effective: Improvements and evaluation of unsupervised PCFG induction. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2721–2731, Brussels, Belgium. Association for Computational Linguistics.

Michael I. Jordan. 1986. Serial order: A parallel distributed processing approach. *Technical report*.

Daniel Jurafsky and James H. Martin. 2009. *Speech and Language Processing (2nd Edition)*. Prentice-Hall, Inc., USA.

Fred Karlsson. 2007. Constraints on multiple center-embedding of clauses. *Journal of Linguistics*, 43(2):365–392.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations*.

Matthieu Labeau and Shay B. Cohen. 2019. Experimenting with power divergences for language modeling. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4104–4114, Hong Kong, China. Association for Computational Linguistics.

Daniel J. Lehmann. 1977. Algebraic structures for transitive closure. *Theoretical Computer Science*, 4(1):59–76.

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.

Chu-Cheng Lin and Arya D. McCarthy. 2022. On the uncomputability of partition functions in energy-based sequence models. In *International Conference on Learning Representations*.

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

Clara Meister, Elizabeth Salesky, and Ryan Cotterell. 2020. Generalized entropy regularization or: There's nothing special about label smoothing. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6870–6886, Online. Association for Computational Linguistics.

George A. Miller and Noam Chomsky. 1963. Finitary models of language users. In D. Luce, editor, *Handbook of Mathematical Psychology*, pages 2–419. John Wiley & Sons.

Thomas Minka. 2005. Divergence measures and message passing. Technical report, Microsoft.

Marvin Lee Minsky. 1986. *Neural Nets and the brain model problem*. Ph.D. thesis, Princeton University.

Mehryar Mohri, Fernando Pereira, and Michael Riley. 2008. *Speech Recognition with Weighted Finite-State Transducers*, pages 559–584. Springer Berlin Heidelberg, Berlin, Heidelberg.

James R. Munkres. 2000. *Topology*, 2nd edition. Prentice Hall, Inc.

Anil Nerode and Burton P. Sauer. 1957. *Fundamental Concepts in the Theory of Systems*. Wright Air Development Center, Air Research and Development Command, United States Air Force.

Frank Nielsen. 2018. What is an information projection? *Notices of the American Mathematical Society*, 65:1.

Gabriel Pereyra, George Tucker, Jan Chorowski, Łukasz Kaiser, and Geoffrey E. Hinton. 2017. Regularizing neural networks by penalizing confident output distributions. In *Proceedings of the International Conference on Learning Representations*.

B.T. Polyak. 1964. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17.

Alethea Power, Yuri Burda, Harrison Edwards, Igor Babuschkin, and Vedant Misra. 2022. Grokking: Generalization beyond overfitting on small algorithmic datasets. *CoRR*, abs/2201.02177.

Halsey L. Royden. 1988. *Real Analysis*, 3rd edition. Prentice-Hall.

Thibault Sellam, Steve Yadlowsky, Ian Tenney, Jason Wei, Naomi Saphra, Alexander D'Amour, Tal Linzen, Jasmijn Bastings, Iulia Raluca Turc, Jacob Eisenstein, Dipanjan Das, and Ellie Pavlick. 2022. The multiBERTs: BERT reproductions for robustness analysis. In *International Conference on Learning Representations*.

Claude E. Shannon. 1948. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423.

Stuart M. Shieber. 1985. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8:333–343.

Michael Sipser. 2013. *Introduction to the Theory of Computation*, third edition. Course Technology, Boston, MA.

Noah A. Smith and Mark Johnson. 2007. Weighted and probabilistic context-free grammars are equally expressive. *Computational Linguistics*, 33(4):477–491.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2015. Rethinking the inception architecture for computer vision. *2016 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826.

Terence Tao. 2011. *An Introduction to Measure Theory*. American Mathematical Society.

Terence Tao. 2016. *Analysis II: Third Edition*. Texts and Readings in Mathematics. Springer Singapore.

Wilson L. Taylor. 1953. "Cloze Procedure": A new tool for measuring readability. *Journalism Quarterly*, 30(4):415–433.

L. Theis, A. van den Oord, and M. Bethge. 2016. A note on the evaluation of generative models. In *4th International Conference on Learning Representations*.

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

George Kingsley Zipf. 1935. *The Psycho-Biology of Language*. Houghton-Mifflin, New York, NY, USA.