Natural Language Processing (252-3005-00L)

Author: Ryan Cotterell

Contents

Chapter 1 Backpropagation

This chapter offers a formal treatment of backpropagation, also known as reverse-mode automatic differentiation (Griewank and Walther 2008). Modern NLP relies heavily on statistical models whose parameters are almost always estimated using gradient-based methods—more specifically, the class of optimization algorithms based on gradient descent. Backpropagation is useful because it allows us to efficiently compute the gradients needed in such algorithms for almost any model. In particular, we will see that gradient computation has the same complexity as function evaluation!

Many tasks in NLP require building a model that maps inputs $\boldsymbol{x}$ to outputs $\boldsymbol{y}$. We want to choose a model that performs this mapping "well," where we quantify "well" using some objective or loss function. More formally, let us consider a parametric model $\mathbf{f}(\boldsymbol{x}; \boldsymbol{\theta})$ that takes $\boldsymbol{x}$ as input and is parameterized by $\boldsymbol{\theta}$. Given a dataset $\mathcal{D}$ with inputs and outputs $(\boldsymbol{x}, \boldsymbol{y})$, a typical objective function for NLP models is of the form

$$\operatorname*{argmin}_{\theta} L(\boldsymbol{\theta}) = \operatorname*{argmin}_{\theta} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{D}} \ell(\mathbf{f}(\boldsymbol{x}; \boldsymbol{\theta}), \boldsymbol{y}), \tag{1.1}$$

where L is the loss function, a sum of the losses ℓ per input pair from our dataset. The gradient descent update rule is then

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \eta_t \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t}, \tag{1.2}$$

where η_t is the step size and $\nabla_{\boldsymbol{\theta}}$ is the gradient operator. The efficient computation of gradients is therefore critical for model estimation in NLP. Luckily, backpropagation provides us with the means for just this.

1.1 Calculus Background

This short section overviews the basic background in differential calculation necessary for this chapter. The primary focus is on definitions, not intuitions. The scalar partial derivative, that is, where we differentiate a scalar function $f(x) \in \mathbb{R}$ with respect to the scalar variable $x \in \mathbb{R}$ is defined as:

$$f'(x) \stackrel{\text{def}}{=} \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}. \tag{1.3}$$

This limit exists if f is continuous and smooth at x, which is generally true for the class of functions we consider. In this case, we say that f is differentiable at x. When dealing with multiple variables, i.e., if f is a function of $\boldsymbol{x} = \langle x_1, \cdots, x_n \rangle$, it is better to use Leibniz's notation for $f'(x)$ which is written as $\frac{\partial f(x)}{\partial x}$. We can now cleanly extend the scalar to the multivariate case. Given a function $f : \mathbb{R}^n \to \mathbb{R}$, the **gradient** of f with respect to $\boldsymbol{x}$ is:

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \stackrel{\text{def}}{=} \left(\frac{\partial f(\boldsymbol{x})}{\partial x_1}, \ldots, \frac{\partial f(\boldsymbol{x})}{\partial x_n} \right) \in \mathbb{R}^n \tag{1.4}$$

We can further extend this definition to functions $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ that output a vector. The resulting matrix $\nabla_{\boldsymbol{x}} \mathbf{f}(\boldsymbol{x}) \stackrel{\text{def}}{=} [\frac{\partial f_1(\boldsymbol{x})}{\partial \boldsymbol{x}}, \ldots, \frac{\partial f_m(\boldsymbol{x})}{\partial \boldsymbol{x}}]^\top$ is called the **Jacobian** of $\mathbf{f}$, where the (i, j)-th element reflects the amount by which $\mathbf{f}_i(\boldsymbol{x})$ will change if x_j is changed by a small amount.

In our basic calculus courses, we typically learn to compute partial derivatives by hand via symbolic differentiation, coming up with a closed form solution for the derivative using the chain-rule and other differentiation rules. However, this is highly inefficient for differentiating functions composed of perhaps billions of variables. This is where backpropagation, which uses the chain-rule in combination with dynamic programming techniques, comes in handy. The chain-rule states that the rate of change of y relative to z and that

of z relative to x allows to compute the rate of change of y relative to x:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z}\frac{\partial z}{\partial x} \tag{1.5}$$

Example 1 *Let $f(x,y,z) = (x-y)^2 z$. The gradient is easily computed symbolically as $\nabla f(x,y,z) = (2(x-y)z, -2(x-y)z, (x-y)^2)$. To compute the partial derivative with respect to x, we can use the chain rule with $a = (x-y)$ and compute $\frac{\partial a^2 z}{\partial a} = 2az$ and then compute $\frac{\partial a}{\partial x} = 1$ to multiply and obtain $2(x-y)z$. To compute the partial derivative with respect to y, we use the same intermediate variable a.*

In the above example, we make use of the redundancy in computations to efficiently compute gradients for complex functions. Backpropagation is a way to programmatically exploit the use of such redundancies.

1.2 Computation Graphs

A computation graph, which we will denote as $\mathcal{G}$ throughout this section, is a graphical (in the sense of graph theory) representation of a function. Here, we focus on acyclic computation graphs:

Definition 1.1

*A **computation graph** $\mathcal{G}$ is a directed acyclic graph (E, V) where each vertex $v \in V$ is labeled with a differentiable function f_v and each vertex with an outgoing edge to v is an argument of f_v. Furthermore, let $\boldsymbol{\alpha} : V \to V^*$ be an ordering of v's incoming edges. We write $\alpha_k(v)$ to refer to vertex v's k^{th} incoming edge.*

♣

We will also need to introduce some utility functions to make it easier to talk about graphs. We define the **incoming** and **outgoing** vertices functions as follows:

$$\mathsf{in}(v) \stackrel{\text{def}}{=} \left\{ v' \mid (v' \to v) \in E \right\} \tag{1.6}$$

$$\mathsf{out}(v) \stackrel{\text{def}}{=} \left\{ v' \mid (v \to v') \in E \right\} \tag{1.7}$$

In words, Equation (1.6) and Equation (1.7) return the set of incoming and outgoing vertices, respectively, for a given vertex v. We define the set of **input vertices** in a computation graph as

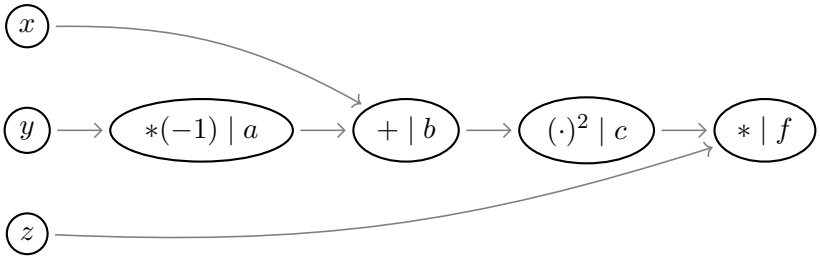
$$\iota(\mathcal{G}) \stackrel{\text{def}}{=} \left\{ v \mid |\mathsf{in}(v)| = 0 \right\} \tag{1.8}$$

In words, the set of vertices with no incoming edges. Likewise, we define the set of **output vertices** in a computation graph as

$$\omega(\mathcal{G}) \stackrel{\text{def}}{=} \left\{ v \mid |\mathsf{out}(v)| = 0 \right\} \tag{1.9}$$

In words, the set of vertices with no outgoing edges. We further define the functions topo and revtopo that take a graph and return the vertices in topologically sorted, respectively reversed, order. Note that there are often many valid topological orders for a given (acyclic) graph and a topological sort can be computed in time linear in the number of edges (Cormen et al. 2009).

Example 2 *We consider the simple function $f(x,y,z) = (x-y)^2 z$ of Example 1 as a computation graph. Each node that is not an input node specifies an operation and we assign an intermediate variable name:*



Now we will turn to evaluating a computation graph. We define the notion of forward-propagation.[1]

Definition 1.2

*Applying **forward-propagation** to a computation graph $\mathcal{G}$ executes the following recursion*

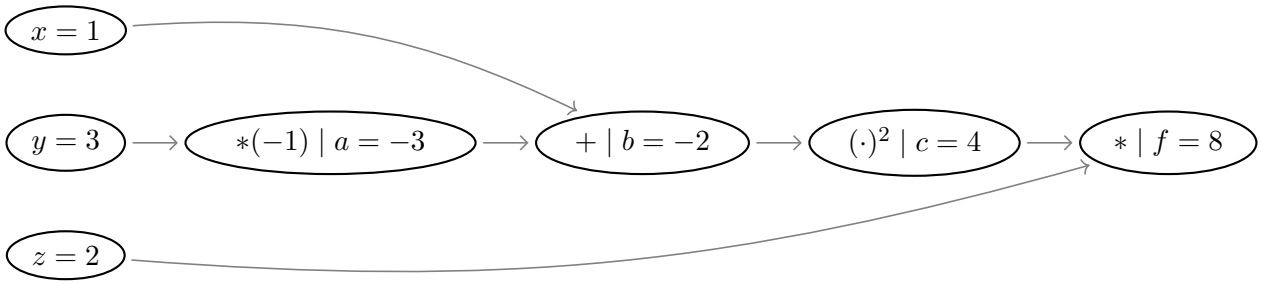
$$z(v) = f_v\Big(z(\alpha_1(v)), \ldots, z(\alpha_K(v))\Big) \tag{1.10}$$

where we have taken $|\boldsymbol{\alpha}(v)| = K$. In the case of input vertices, we have $|\boldsymbol{\alpha}(v)| = 0$ and $f(\cdot)$ is a constant function.

♣

For the remainder of the chapter, we will assume a bijection between the set of vertices V in a computation graph $\mathcal{G}$ and the first $N \stackrel{\text{def}}{=} |V|$ natural numbers $\{1, \ldots, N\}$. This will allow us to avoid writing the cumbersome $z(v)$ in favor of the sleeker z_n. Now, the first question to ask of a newly encountered algorithm, e.g. the one specified in Equation (1.10), is what is its runtime and space complexity. The runtime complexity is $\mathcal{O}(|E|)$, i.e., linear in the number of edges of the computation graph $\mathcal{G}$. To see this, simply observe that each edge used in the evaluation of z for exactly one v. The space complexity, on the other hand, is $\mathcal{O}(|V|)$, i.e., linear in the number of vertices of the computation graph.

Example 3 *We consider the function $f(x, y, z) = (x - y)^2 z$ of Example 1 again and forward-propagate at $(x, y, z) = (1, 3, 2)$. In the computation graph, this means that we assign values to the variables:*



Now, we will begin to discuss the main point of concern of this chapter: how we can compute partial derivatives on a computation graph. We turn to a simple idea by Bauer 1974, who extended the definition of a partial derivative.

Proposition 1.1 (Bauer's Formula)

*Let $\mathcal{G}$ be a computation graph. For any two nodes i and j in $\mathcal{G}$, let $\mathcal{P}(j, i)$ be the set of all directed paths starting at j and ending at i. (The elements of the set $\mathcal{P}(j, i)$ are referred to as **Bauer paths**.) The partial derivative of z_i with respect to z_j is then equal to*

$$\frac{\partial z_i}{\partial z_j} = \sum_{\boldsymbol{\pi} \in \mathcal{P}(j,i)} \prod_{(k,l) \in \boldsymbol{\pi}} \frac{\partial z_l}{\partial z_k} \tag{1.11}$$

[1] Historical note: Forward-propagation as a term was invented *after* the coining of backpropogation.

*This result is referred to as **Bauer's formula**.*

Proof We prove Bauer's formula by induction backwards in topological order from an arbitrary node z_i.

- **Base Cases:** We have the base case $i = j$, i.e., $z_i = z_j$:

$$\frac{\partial z_i}{\partial z_i} = 1 \tag{1.12}$$

- **Inductive Hypothesis:** Bauer's formula holds for all $j \leq i$:

$$\frac{\partial z_i}{\partial z_j} = \sum_{\boldsymbol{\pi} \in \mathcal{P}(j,i)} \prod_{(k,l) \in \boldsymbol{\pi}} \frac{\partial z_l}{\partial z_k} \tag{1.13}$$

- **Inductive Step:** For any $m < j$, i.e., z_m comes before z_j in topological order, and $j \in \mathsf{out}(m)$:

$$\begin{aligned}
\frac{\partial z_i}{\partial z_m} &= \sum_{j \in \mathsf{out}(m)} \frac{\partial z_j}{\partial z_m} \frac{\partial z_i}{\partial z_j} && \text{(multivariate chain rule)} \\
&= \sum_{j \in \mathsf{out}(m)} \frac{\partial z_j}{\partial z_m} \left(\sum_{\boldsymbol{\pi} \in \mathcal{P}(j,i)} \prod_{(k,l) \in \boldsymbol{\pi}} \frac{\partial z_l}{\partial z_k} \right) && \text{(inductive hypothesis)} \\
&= \sum_{j \in \mathsf{out}(m)} \left(\sum_{\boldsymbol{\pi} \in \mathcal{P}(j,i)} \frac{\partial z_j}{\partial z_m} \prod_{(k,l) \in \boldsymbol{\pi}} \frac{\partial z_l}{\partial z_k} \right) && \text{(distributive property)} \\
&= \sum_{\boldsymbol{\pi} \in \mathcal{P}(m,i)} \prod_{(k,l) \in \boldsymbol{\pi}} \frac{\partial z_l}{\partial z_k} && \text{(concatenate paths)}
\end{aligned}$$

This completes the proof.

1.3 Derivation and Analysis of Backpropagation

The discussion in §1.2 focused on the properties of computation graphs. This section turns to algorithms that can be performed on computation graphs—specifically, the derivation of the backpropation algorithm. Bauer's formula, i.e., Equation (1.11), gives us a starting point from which we are able to develop an efficient algorithm.

Algorithm 1.1

def *BauerBackprop*($\mathcal{G}$)***:***
  $\mathcal{D} = \{\}$ ▷ *Set of partial derivatives* $\frac{\partial z_i}{\partial z_j}$
  for $i \in \omega(\mathcal{G})$***:***
    for $j \in \iota(\mathcal{G})$***:***
      $\frac{\partial z_i}{\partial z_j} \leftarrow \sum_{\boldsymbol{\pi} \in \mathcal{P}(j,i)} \prod_{(k \to l) \in \pi} \frac{\partial z_l}{\partial z_k}$ ▷ *Runs in* $\mathcal{O}|\mathcal{P}(j,i)|$ *time*
    end for
  end for
  return $\mathcal{D}$

The above algorithm may run in exponential time. Why? The issue is that $\mathcal{O}(|\mathcal{P}(j,i)|)$ may grow to be quite large. For example, in a fully connected graph (e.g., a computation graph representing a deep, fully connected neural network), the number of paths connecting j and i will be exponential in the path length. This renders Algorithm 1.1 infeasible in practice. Luckily, there is a simple trick to developing a faster algorithm

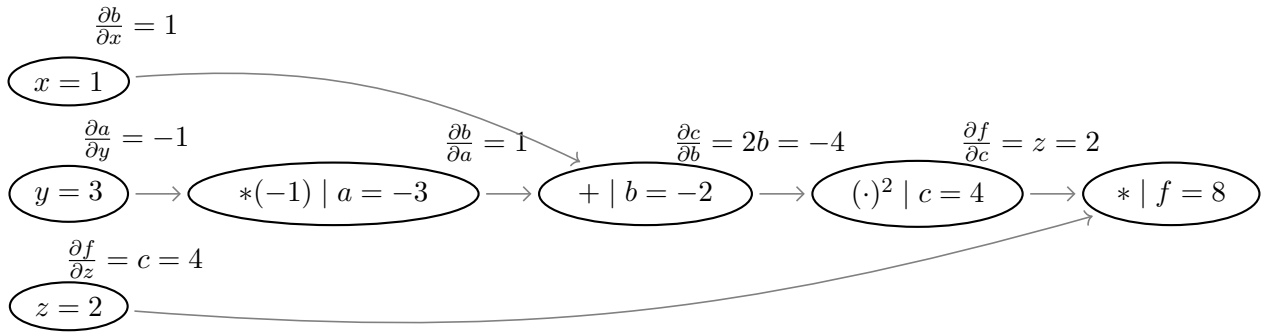
that is revealed in the proof of Proposition 1.1. The insight is that a lot of the work done in the computation of the $\sum_{\boldsymbol{\pi} \in \mathcal{P}(j,i)} \prod_{(k \to l) \in \pi} \frac{\partial z_l}{\partial z_k}$ is redundant. Performing a simple **memoization** of the already computed partial derivatives $\frac{\partial z_l}{\partial z_k}$ yields an efficient algorithm.

Algorithm 1.2

def *BACKPROP*($\mathcal{G}$)*:*
  $\mathcal{D} = \{\}$
  for $i \in \omega(\mathcal{G})$*:*
    for $j \in$ *revtopo*$(\mathcal{G} \setminus \{i\})$*:* ▷ *Assumes* $\mathcal{G}$ *is acyclic*
      $\frac{\partial z_i}{\partial z_j} \leftarrow \sum_{k \in out(j)} \frac{\partial z_i}{\partial z_k} \frac{\partial z_k}{\partial z_j}$ ▷ *Every edge* $i \to j$ *gets touched once;* $\frac{\partial z_i}{\partial z_j}$ *memorized*
    end for
  end for
  return $\mathcal{D}$

Equation (1.11) is easily recognized as a dynamic program. In fact, it has the same structure as the simple dynamic programs used to compute the shortest path in acyclic graphs. The algorithm above uses memorization of previously computed partial derivatives, $\frac{\partial z_i}{\partial z_j}$, that are guaranteed to be computed due to the reversed topological order. The algorithm has a runtime of $\mathcal{O}(|E|)$ since we touch every edge exactly once and requires $\mathcal{O}(|V|)$ space for the partial derivatives at each node. This runtime bound is, in a sense, the magic of backpropogation. It tells us that we can compute the gradient of a function in time that is linear in the size of the computation graph. Griewank and Walther 2008 refer to this property as the **cheap gradient principle**: Gradients are cheap in the sense that they are as fast to compute as the original function itself. To use the gradient for optimization, we need to evaluate the computation graph at current parameter and input values (forward propagation) and use backpropagation to compute the corresponding partial derivatives. An example is given below.

Example 4 *Using the forward-propagated Example 3, we execute the backpropagation step on the same graph.*



The resulting gradients are $\frac{\partial f}{\partial x} = 2bz = 2(x-y)z = -8$, $\frac{\partial f}{\partial y} = -2zb = -2(x-y)z = 8$, $\frac{\partial f}{\partial z} = c = (x-y)^2 = 4$.

Chapter 2 Log-linear Modeling

For our discussion on probabilistic models of language—which constitute a large part of the content of this class—we will start with a brief review of the fundamentals of probability theory. Later in this chapter, we will introduce log-linear models (Section §2.2), a popular class of models in NLP and machine learning. These models strike a balance between accurately representing reality while still being analytically tractable. We demonstrate this in Section §2.3 by showing how to fit log-linear models with simple methods like gradient descent and Bayesian estimation. Finally, we talk about an important component of log-linear models—the softmax—and how it relates the class of log-linear models to the exponential family.

2.1 Probability Theory

Probability concerns the study of uncertainty. In machine learning and statistics, there are two predominant interpretations of probability, namely the frequentist interpretation and the Bayesian interpretation:

- **The frequentist interpretation** of probability is as the expected behavior of an experiment when the experiment is repeated an arbitrary number of times, i.e., the probability of an event is defined as the relative frequency of the event in the limit when one has infinite data. To give a more precise, but still informal definition, let us consider an experiment with a fixed set of possible outcomes Ω. After executing the experiment n times, we measure the number n_A of times that a particular event of interest $A \subseteq \Omega$ occurs. The *probability of* A, notated as $p(A)$, is then given as

$$p(A) = \lim_{n \to \infty} \frac{n_A}{n}. \tag{2.1}$$

  For example, in the experiment of rolling a die, after sufficiently many rolls, one can expect any of the six possible outcomes to appear one sixth of the time.
- **The Bayesian interpretation** uses probability to specify the degree of uncertainty that the user has about an event. Such a definition allows us to speak about probabilities in experiments that cannot be repeated. For example, Bayesianists can state that the probability that the sun will freeze in the next decade is zero, even when they do not have the ability to observe thousands of suns within the next decade. More precisely, this interpretation states that the *probability of* $\omega \in \Omega$ is a value between 0 and 1 that quantifies how much the experimenter believes that ω will be the outcome in a single execution of the experiment. Here, 1 denotes full certainty that ω will be the outcome and 0 denotes full certainty that ω will not be the outcome.

The choice of which interpretation to employ is largely a matter of opinion, but is also constrained by the problem itself and the available computational resources. Estimates based on a frequentist definition are often easier to compute, but at the price of potentially being overconfident. Estimates based on a Bayesian definition are preferred when you do not have enough data to be fully certain about your estimates; however, this approach often involves difficult computations.

Independent from which definition we choose, we can still agree on a single framework for modeling experimental outcomes and their associated probabilities. We assume the reader already understands the notions of σ-algebras and probability measures, but they can be easily found in any standard textbook.

Modern probability is defined in terms of three concepts: **sample space**, **event space** and **probability measure**, which are introduced in a set of axioms proposed by Kolmogorov.

Definition 2.1
*The **sample space** Ω is the set of all possible outcomes of the experiment.* ♣

Definition 2.2
*The **event space** $\mathcal{E}$ is the set of potential results of the experiment. $\mathcal{E}$ is a σ-algebra of subsets of Ω.* ♣

Definition 2.3
*The **probability measure** is a number that measures the probability or degree of belief that an event $e \in \mathcal{E}$ will occur, which is denoted as $p(e)$ where $p : \mathcal{E} \to [0, 1]$.* ♣

The concepts of sample space, event space and probability measure together define a probability space:

Definition 2.4
*A **probability space** is a triple $(\Omega, \mathcal{E}, p)$, where*

- *Ω is the **sample space**,*
- *$\mathcal{E}$ is the **event space**,*
- *p is a **probability measure**,*

such that

- *$p(\Omega) = 1$ and*
- *for any disjoint collection $\{A_i\}_{i\in\mathbb{N}}$ of subsets of Ω, we have $p\left(\bigcup_{i\in\mathbb{N}} A_i\right) = \sum_{i\in\mathbb{N}} p(A_i)$.* ♣

2.1.1 Random variables

Given a probability space $(\Omega, \mathcal{E}, p)$, we may want to transform an outcome space Ω into an event space $\mathcal{E}$. We often avoid explicitly referring to the sample space, but instead refer to probabilities on properties of interest. With this purpose, we often use **random variables**.

Definition 2.5
*Let $(\Omega, \mathcal{E}, p)$ be a probability space. A **random variable** is a function $X : \Omega \to \mathcal{E}$, where $\mathcal{E}$ is a measurable space and X is a measurable function. This function takes an element of Ω, and return a value in $\mathcal{E}$.* ♣

Here we do not go into the details of what "measurable" means. In practice, we usually let $\mathcal{E}$ be $\mathbb{R}^d$ or a finite set, equipped with natural measures from measure theory. To preserve the structure of the event space $\mathcal{E}$, X must be a measurable function. For our purposes, you can assume that a random variable is just a function mapping Ω to some space without worrying that the function is measurable or that $\mathcal{E}$ has an adequate measure.

Let us give an example to better illustrate the concepts we introduce in this section. For example, in the case of tossing two coins and *counting the number of heads*, a random variable X maps to the following possible outcomes: $X(hh) = 2, X(ht) = 1, X(th) = 1, X(tt) = 0$, where we have a sample space $\Omega = \{hh, tt, ht, th\}$, and an event space $\mathcal{E} : \{0, 1, 2\}$, where h and t represents "heads" and "tails," respectively.

We can think of another scenario to further explain why we need random variables in our analysis. Let us define a new probability space where sample space Ω is the set of all people in the world. In that case, the event space $\mathcal{E}$ is the power set of Ω. Let us define a uniform distribution $p(x) = \frac{1}{|\Omega|}, \forall x \in \Omega$. We are interested in whether a person speaks Swiss German and/or Romansh. With this purpose, we define a Swiss German random variable $S(x) =$ **true** if x speaks Swiss German, and **false** otherwise. Similarly, let us define a random variable

for Romansh, $R(x) =$ **true** if x speaks Romansh and **false** otherwise. We have a single source of randomness (over Ω), but we are interested in answering some questions such as the following:

- $p(x$ speaks Swiss German and x speaks Romansh$) = p(S =$ **true**, $R =$ **true**$)$
- $p(x$ speaks Swiss German if we know x speaks Romansh$) = p(S =$ **true** $\mid R =$ **true**$)$
- $p(x$ speaks Swiss German if we know x does not speak Romansh$) = p(S =$ **true** $\mid R =$ **false**$)$

However, reasoning about the probabilities above would be very hard without random variables. Random variables are fundamentally about interactions between different properties of the sample space. Without random random variables, it would be also hard to talk about properties such as independence and correlation, which are essentially properties of random variables and not of the probability spaces.

2.1.2 Probability Refresher

We will now provide an overview of some key concepts that underlie probabilistic modeling. First, it is important to note that there are two classes of probability distributions that we might come across in our analyses: discrete and continuous (see Table 2.1 for definitions). We must often employ different techniques when working with each of these distributions.

Type	Point Probability	Interval Probability
Discrete	$p(X = x)$ - Probability Mass Function	not applicable
Continuous	$p(x)$ - Probability Density Function	$p(X \leq x)$ - Cumulative Distribution Function

Table 2.1: Probability distributions

Since our outputs in NLP are often discrete, we focus largely on this class of distributions. For probabilistic analyses, we rely on three fundamental conditions, called the axioms of probability theory. Below, the axioms of probability in the discrete case are presented.

Definition 2.6

Axioms of Probability (in the discrete case): *Let $\mathcal{X}$ be a countable set. A discrete probability distribution must satisfy three axioms:*

- ***Non-negativity:*** $p(X = x) \geq 0$
- ***Sums to 1:*** $\sum_{x \in \mathcal{X}} p(X = x) = 1$
- ***Countable additivity:*** $p(X = x \cup X = y) = p(X = x) + p(X = y)$*, if $X = x$ and $X = y$ are disjoint events, i.e., if $X = x$ and $X = y$ are mutually exclusive.*

♣

Joint probability: We are often interested in two or more random variables at the same time. In such situations, we rely on the concept of *joint probability*, which allows us to compute probabilities of events involving these random variables and understand the relationship between the variables. The **joint distribution** of two random variables X and Y is denoted as $p(X = x, Y = y)$. We often use $p(x, y)$ as a shorthand notation when we would like to refer to $p(X = x, Y = y)$.

Conditional probability: Often, the probability of an event changes if we have extra information about some other event. Conditional probability answers the question "What is the probability of an event, given that we have already observed some other event?", by quantifying how we update our beliefs in the face of new evidence.

Definition 2.7

Conditional probability *The probability of $X = x$ given that (conditioned on) $Y = y$ is computed as: $p(x|y) = \frac{p(x,y)}{p(y)}$, provided that $p(y) \neq 0$.*

♣

Marginalization: Given the distribution of a joint pair of random variables X, Y taking values in $\mathcal{X} \times \mathcal{Y}$, we can recover the probability distribution of any variable by integrating (for the continuous case) or summing (for the discrete case) over all other variables. This process is called marginalization. See Table 2.2 for an example.

Variable	Discrete	Continuous
X	$p(x) = \sum_{y \in \mathcal{Y}} p(x, y)$, for $x \in \mathcal{X}$	$p(x) = \int p(x, y)\mathrm{d}y$, for $x \in \mathcal{X}$
Y	$p(y) = \sum_{x \in \mathcal{X}} p(x, y)$, for $y \in \mathcal{Y}$	$p(y) = \int p(x, y)\mathrm{d}x$, for $y \in \mathcal{Y}$

Table 2.2: Marginalization

Bayes' rule: We are often interested in making inferences of unobserved random variables given that we have observed other random variables. Let us assume that we have some prior information about an unobserved random variable Y, and information about some relationship $p(X = x|Y = y)$ between Y and another random variable X, which we can in fact observe. Bayes' rule can be used to improve our predictions for Y, given the observed values of X.

Definition 2.8

Bayes' rule:

$p(Y = y|X = x) = p(y|x) = \frac{p(x|y)p(y)}{p(x)} = \frac{p(x|y)p(y)}{\int p(x|y')p(y')\mathrm{d}y'}$, *where*

- $p(y|x)$ *is the* ***posterior****.*
- $p(x|y)$ *is the* ***likelihood****.*
- $p(y)$ *is the* ***prior****.*
- $p(x)$*, or* $\int p(x|y')p(y')\mathrm{d}y'$*, is the* ***marginal****.*

♣

When two random variables X and Y are **independent**, the joint distribution $p(x, y)$ factorizes into a product of the marginals: $p(x, y) = p(x)p(y)$. In that we case, we denote this independence by $X \perp Y$.

Definition 2.9

When two random variables are ***independent****, making an observation of the variable X does not give us any information about Y, and vice-versa:*

- $p(y|x) = p(y)$
- $p(x|y) = p(x)$

♣

Expectation: The expectation of a random variable is the expected or average value of the random variable where each value is weighted according to the underlying probability distribution.

Definition 2.10

- *Suppose that X is a* ***discrete random variable*** *with PMF $p(x)$ and $f : \mathbb{R} \to \mathbb{R}$ is an arbitrary function. In this case, $f(X)$ can be considered a random variable, and we define the* ***expectation*** *or* ***expected*** *value of $f(X)$ as $\mathbb{E}[f(x)] = \sum_x f(x)p(x)$.*
- *For a* ***continuous random variable*** *X with PDF $p(x)$, the expected value of $f(X)$ is defined as, $\mathbb{E}[f(x)] = \int f(x)p(x)\mathrm{d}x$.*

♣

Below, we outline the computation of expectation, and list the properties of expectation.

Definition 2.11

Properties of expectation

Rule 1 ***(Linearity):*** *Expected value of a constant is the constant, i.e.* $\mathbb{E}[k] = k$.

Rule 2 ***(Linearity):*** *Expected value of a constant times function = constant times expected value function, i.e.* $\mathbb{E}[kf(x)] = k\mathbb{E}[f(x)]$.

Rule 3 ***(Linearity):*** *Expectation of the sum of functions is the sum of expectation of functions, i.e.* $\mathbb{E}[f(x) + g(x)] = \mathbb{E}[f(x)] + \mathbb{E}[g(x)]$.

Rule 4 *: Expectation of the product of functions is the product of expectations of functions if* x *and* y *are independent (Follows from the distributive property), i.e.* $\mathbb{E}[f(x)g(y)] = \mathbb{E}[f(x)]\mathbb{E}[g(y)]$.

2.2 Log-linear Models

Log-linear models are a very general family of probability distributions that can be defined as

$$p(y|x, \boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \exp(\boldsymbol{\theta} \cdot \mathbf{f}(x, y)) \tag{2.2}$$

where we have the following definitions

- **Inputs:** $x \in \mathcal{X}$
- **Output label:** $y \in \mathcal{Y}$
- **Feature function:** $\mathbf{f} : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^K$
- **Parameters:** $\boldsymbol{\theta} \in \mathbb{R}^K$
- **Partition function:** $Z(\boldsymbol{\theta}) := \sum_{y' \in \mathcal{Y}} \exp(\boldsymbol{\theta} \cdot \mathbf{f}(x, y'))$, which is a normalization constant for the probability distribution. Z stands for "Zustandsumme".

To see why this family of distributions is called *log-linear*, note that if we take the log of §2.2, we end up with a linear function:

$$\log p(y|x, \boldsymbol{\theta}) = \boldsymbol{\theta} \cdot \mathbf{f}(x, y) + \text{const.} \tag{2.3}$$

2.2.1 Feature Engineering

The feature function $\mathbf{f}$ is a crucial part of a log-linear model. The process of designing a suitable feature function is called **feature engineering**. In the earlier years of NLP, feature engineering used to be a big portion of machine learning for NLP. Here, we outline two important steps in feature engineering, namely preprocessing and the design of the feature function:

- **Preprocessing:** In preprocessing, the goal is to convert raw text into a form that is more amenable to feature design. Below are some examples of preprocessing operations:
  - Tokenization
    - "Joao's quadricycle is completely BROKEN!" $\longrightarrow$ [Joao, 's, quadricycle, is, completely, BROKEN, !]
  - Lower casing
    - [Joao, 's, quadricycle, is, completely, BROKEN, !] $\longrightarrow$ [joao, ..., broken, !]
  - Stemming
    - [joao, 's, quadricycle, is, completely, broken, !] $\longrightarrow$ [.., complete, broke, ...]

- Stop word removal
  - [joao, 's, quadricycle, is, complete, broke, !] $\longrightarrow$ [joao, quadricycle, complete, broke, !]
- Reducing vocabulary
  - [joao, quadricycle, complete, broke, !] $\longrightarrow$ [<UNK>, <UNK>, complete, broke, !]

- **Feature function design:** Our next step is to extract **features** from our text. These features are what are ultimately used by the model when scoring a piece of text. Below, we list several potential features:
  - n-grams
    - [(BOS, BOS, UNK), (BOS, UNK, 's), ..., (complete, broke, !)]
  - One-hot encoding
    - [0, 0, 1, 0, ...]
  - Bag-of-words (see the spam classification example from the slides)
    - [0, 1, 0, 2, ...]
  - Word embeddings
    - Convert word into continuous representations
    - Can be contextual (e.g., ELMo, BERT) or not (e.g., Glove, fastText)
  - Bag-of-embeddings
    - Average embeddings for all words in a sequence
  - Domain-specific features: Often, domain knowledge could also help us design better feature functions.
    - "She thought of herself". $\longrightarrow$ Using 3-gram features, (thought, of) $\longrightarrow$ herself/himself should be equally probable (excluding data biases). But the reflexive pronoun must agree with the subject's grammatical gender, i.e. we can amend our feature function to track the grammatical gender of the subject:
    - $p(w_3 = \text{herself} | \text{SubjGen} = \text{FEM}, w_1 = \text{thought}, w_2 = \text{of}) > p(w_3 = \text{himself} | \text{SubjGen} = \text{FEM}, w_1 = \text{thought}, w_2 = \text{of})$

2.3 Estimating the parameters of log-linear models

2.3.1 Maximum likelihood estimation

In order to estimate the parameters $\boldsymbol{\theta}$ of a log-linear model, we can perform **maximum likelihood estimation (MLE)**, i.e., find the parameters that maximize the likelihood of observing our training data. We call the resulting parameter vector the MLE estimate, and denote it as $\boldsymbol{\theta}_{\text{MLE}}$.

More formally, assume that we are given training data $\{(x_n, y_n)\}_{n=1}^N$. The log-likelihood of the training data under a distribution parameterized by $\boldsymbol{\theta}$ is computed as

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{n=1}^{N} \log p(y_n \mid x_n, \boldsymbol{\theta}). \tag{2.4}$$

The MLE estimate $\boldsymbol{\theta}_{\text{MLE}}$ is then defined as

$$\boldsymbol{\theta}_{\text{MLE}} := \arg\max_{\boldsymbol{\theta} \in \Theta} \mathcal{L}(\boldsymbol{\theta}), \tag{2.5}$$

where Θ is restricted to be a compact subset of $\mathbb{R}^K$. To understand this restriction, consider the case of binary classification where your data is *separable*. That is, there is $\boldsymbol{\theta} \in \mathbb{R}^K$, such that $\boldsymbol{\theta} \cdot \mathbf{f}(x_n, y_n) \geq \boldsymbol{\theta} \cdot \mathbf{f}(x_n, 1 - y_n)$, for $n \in \{n : 1 \leq n \leq N, n \in \mathcal{Z}^+\}$. In this setting, for any $\boldsymbol{\theta}^*$, there is another $\boldsymbol{\theta}^{**}$ such that $\mathcal{L}(\boldsymbol{\theta}^{**}) > \mathcal{L}(\boldsymbol{\theta}^*)$, making the $\boldsymbol{\theta}_{\text{MLE}}$ ill-defined.

A standard way to estimate the maximizer of $\mathcal{L}(\boldsymbol{\theta})$ for $\boldsymbol{\theta} \in \Theta$ is to solve

$$\nabla \mathcal{L}(\boldsymbol{\theta}) = 0 \tag{2.6}$$

This solution has an interesting property. We can rewrite the gradient of $\mathcal{L}$ as

$$\nabla \mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \mathbf{f}(x_n, y_n) - \sum_{n=1}^{N} \mathbb{E}_{Y \sim p(\cdot|x_n, \boldsymbol{\theta})}[\mathbf{f}(x_n, Y)]. \tag{2.7}$$

Thus, when $\nabla \mathcal{L}(\boldsymbol{\theta}^*) = 0$, we have

$$\sum_{n \leq N} \mathbf{f}(x_n, y_n) = \sum_{n=1}^{N} \mathbb{E}_{Y \sim p(\cdot|x_n, \boldsymbol{\theta}^*)}[\mathbf{f}(x_n, Y)] \tag{2.8}$$

Therefore, the optimum is where the expected feature counts under our model look like the observed feature counts from our training data.

Unfortunately, there is often no analytical solution to Equation (2.6). Fortunately, computing $\nabla \mathcal{L}$ is easy, and $-\mathcal{L}$ turns out to be a convex function so we can instead employ numerical optimization algorithms.

Definition 2.12

Convexity: *A real-valued function defined on an n-dimensional interval is called **convex** if the line segment between any two points on the graph of the function lies above the graph between the two points.* ♣

✍ **Exercise 2.1** Show that for log-linear models, $-\mathcal{L}$ is a convex function.

It is important to note that any local minimum of a convex function is a global minimum. In such situations where the objective function is convex, we typically use gradient-based methods (such as gradient descent) to find a minimizer since due to the convexity of our objective, these methods will converge to a global optimum.

2.3.2 Gradient descent

Gradient descent can be motivated by the following intuition. Imagine that you are in the mountains and want to get to the lowest point of the region. To complicate things, assume that there is a fog that allows you only to see a neighborhood around you. A good heuristic to find the lowest point is to walk to the lowest point in your visible neighborhood. You repeat this procedure until you reach a point where you cannot go further down. This is exactly what we do during gradient descent.

We can instantiate this procedure by letting the negative log likelihood $-\mathcal{L}$ define the "landscape." We then start at an arbitrary location $\boldsymbol{\theta}_0 \in \Theta$. Afterwards, we compute a sequence of locations $\boldsymbol{\theta}_0, \boldsymbol{\theta}_1, \ldots$ where

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \text{update magnitude} \times \text{update direction}. \tag{2.9}$$

The update added to $\boldsymbol{\theta}_t$ to obtain $\boldsymbol{\theta}_{t+1}$ is similar to moving to the lowest point in the visible neighborhood. Hence, it is natural to choose the update direction to be $\nabla \mathcal{L}(\boldsymbol{\theta}_t)$[1] and the update magnitude η_t to be a moderate positive value, as we only make a small move. To ensure convergence, η_t is often chosen to decrease with t. The update magnitude is commonly called the *learning rate*. We can formally define our update rule as

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta_t \nabla \mathcal{L}(\boldsymbol{\theta}_t). \tag{2.10}$$

[1]Recall that the direction of steepest descent of a differentiable function f at point x is $-\nabla f(x)$. Thus, the direction of steepest descent in our case is $-\nabla - \mathcal{L}(\boldsymbol{\theta}_t) = \nabla \mathcal{L}(\boldsymbol{\theta}_t)$.

2.4 The Softmax

This class is about building probabilistic models of natural language. Often these models must produce a discrete probability distribution over K classes, i.e., a vector in the $(K-1)$ simplex.

Definition 2.13

The $K-$dimensional simplex $\mathbb{R}^K_{\geq 0}$ is a region of space such that the sum of the components is one. This means that any point in a $(K-1)-$simplex defines a unique categorical distribution, where K is the number of categories.

♣

The softmax is the default way of building probabilistic models when we want such distribution. For example, in the context of our current section, we can think of the log-linear model as a dot product followed by a softmax function. Models parameterized by neural-networks also typically use the softmax. Let us first visit the definition of the softmax function:

Definition 2.14

*The **softmax** is a map from a vector in $\mathcal{R}^K$ to a point on the simplex Δ^{K-1}. The softmax at position y of a vector $\mathbf{h} \in \mathbb{R}^K$ is defined as*

$$\text{softmax}(\mathbf{h}, y, T) = \frac{\exp(h_y/T)}{\sum_{y' \in \mathcal{Y}} \exp(h_{y'}/T)} \tag{2.11}$$

*where **temperature** $T \in \mathbb{R}^+$ is a hyperparameter (often omitted, i.e., $T = 1$), which is also called the annealing parameter.*

♣

Note that in the following, we overload notation such that $\text{softmax}(\mathbf{h}, y)$ implies a temperature of 1 and further, $\text{softmax}(\mathbf{h})$ refers to the entire vector over all K classes.

The temperature parameter in §2.4 is also referred to as the *annealing* parameter. T allows us to smoothly interpolate between the *softmax* and *argmax*. Let us show that the limit of softmax as $T \to 0$ is the argmax function, i.e. a function whose value is the index of the maximum. Taking a two-dimensional vector $\mathbf{h} = [h_1, h_2]^T$ as an example:

$$\lim_{T \to 0} \text{softmax}(\mathbf{h}, 1, T) = \lim_{T \to 0} \frac{\exp\left(\frac{h_1}{T}\right)}{\exp\left(\frac{h_1}{T}\right) + \exp\left(\frac{h_2}{T}\right)} \tag{2.12}$$

$$= \lim_{T \to 0} \frac{\exp\left(\frac{h_1}{T}\right) \exp\left(-\frac{h_1}{T}\right)}{\left(\exp\left(\frac{h_1}{T}\right) + \exp\left(\frac{h_2}{T}\right)\right) \exp\left(-\frac{h_1}{T}\right)} \tag{2.13}$$

$$= \lim_{T \to 0} \frac{1}{1 + \exp\left(\frac{h_2 - h_1}{T}\right)} \tag{2.14}$$

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

$$\lim_{T \to 0} \exp\left(\frac{h_2 - h_1}{T}\right) = \begin{cases} 0, & \text{if } h_1 > h_2 \\ 1, & \text{if } h_1 = h_2 \\ \infty, & \text{o.w.} \end{cases} \tag{2.15}$$

Then the limit of softmax as $T \to 0$ is given as

$$\lim_{T \to 0} \text{softmax}(\mathbf{h}, T) = \begin{cases} [1, 0]^\top, & \text{if } h_1 > h_2 \\ \left[\frac{1}{2}, \frac{1}{2}\right]^\top, & \text{if } h_1 = h_2 \\ [0, 1]^\top, & \text{o.w.} \end{cases} \tag{2.16}$$

2.4.1 Gradient of the Softmax

Recall that we typically work with log-likelihoods; consequently, we are interested specifically in the gradient of the *log*-softmax, as this will be the transformation applied in the context of log-linear modeling. Let $T = 1$. The log-softmax is then defined as:

$$\log \text{softmax}(\mathbf{h}, y) = h_y - \log \sum_{y' \in \mathcal{Y}} \exp\left(h_{y'}\right) \tag{2.17}$$

We can derive the gradient of the log-softmax with the chain rule as follows:

$$\begin{aligned} \frac{\partial \log \text{softmax}(\mathbf{h}, y)}{\partial h_i} &= \frac{\partial}{\partial h_i} \left[h_y - \log \sum_{y' \in \mathcal{Y}} \exp\left(h_{y'}\right) \right] && (2.18) \\ &= \delta_{yi} - \frac{\partial}{\partial h_i} \left[\log \sum_{y' \in \mathcal{Y}} \exp\left(h_{y'}\right) \right] && (2.19) \\ &= \delta_{yi} - \frac{1}{\sum_{y' \in \mathcal{Y}} \exp\left(h_{y'}\right)} \frac{\partial}{\partial h_i} \left[\exp\left(h_i\right)\right] && (2.20) \\ &= \delta_{yi} - \frac{1}{\sum_{y' \in \mathcal{Y}} \exp\left(h_{y'}\right)} \exp\left(h_i\right). && (2.21) \end{aligned}$$

Since a log-linear model is just a softmax-ed dot product, it is easy to (re-)derive its gradient using the above formula. We set $h_y = \boldsymbol{\theta} \cdot \mathbf{f}(x, y)$. Note that $\frac{\partial h_i}{\partial \boldsymbol{\theta}} = \frac{\partial}{\partial \boldsymbol{\theta}} \boldsymbol{\theta} \cdot \mathbf{f}(x, i) = \mathbf{f}(x, i)$. Then,

$$\begin{aligned} \frac{\partial \log \text{softmax}(\mathbf{h}, y)}{\partial \boldsymbol{\theta}} &= \frac{\partial \log \text{softmax}(\mathbf{h}, y)}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \boldsymbol{\theta}} && (2.22) \\ &= \sum_{i=1}^{K} \frac{\partial \log \text{softmax}(\mathbf{h}, y)}{\partial h_i} \frac{\partial h_i}{\partial \boldsymbol{\theta}} && (2.23) \\ &= \mathbf{f}(x, y) - \sum_{i=1}^{K} \text{softmax}(\mathbf{h}, I)\, \mathbf{f}(x, i) && (2.24) \\ &= \mathbf{f}(x, y) - \mathbb{E}_{y' \sim \text{softmax}(\mathbf{h}, \cdot)} \left[\mathbf{f}(x, y')\right]. && (2.25) \end{aligned}$$

2.5 The Exponential Family

The exponential family provides a framework for generalizing the softmax.

Definition 2.15

*The **exponential family** is a family of probability distributions over $x \in \mathcal{X}$, parameterized by some $\boldsymbol{\theta}$, of the form*

$$p(x \mid \boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \mathbf{h}(x) \exp\left(\boldsymbol{\theta} \cdot \boldsymbol{\phi}(x)\right), \tag{2.26}$$

where

- *$Z(\boldsymbol{\theta})$ is the partition function,*
- *$\mathbf{h}(x)$ determines the support of the function (exact zeros in the model),*
- *$\boldsymbol{\theta}$ are the canonical parameters,*
- *$\boldsymbol{\phi}(x)$ are the sufficient statistics. In our context, this is the same as a feature function.*

♣

To avoid any subsequent confusion, let us clarify the difference between "distribution" and "family." The exponential family is a *set* of distributions, where the specific distribution varies with the parameter. A "parametric family of distributions" are often referred to as a "distribution," such as the normal distribution which refers to the family of normal distributions. *The* exponential family outlined in this section refers to the set of all exponential distributions.

One of the main reasons why we are interested in the exponential family is that some of the most commonly used/encountered probability distributions belong to this family, such as the Gaussian, Poisson, Bernoulli/Categorical, Gamma and Beta distributions. This framework makes our results applicable to a large number of distributions: if we prove something about the exponential family, we have proven it for a lot of distributions at once!

The exponential family has a number of interesting properties:

- It is the only family with **finite sufficient statistics**. Given data from a member of the exponential family, we can compress all of it into a finite vector without any loss of information.
- It is the only family with **conjugate priors**.
- The exponential family corresponds to **maximum entropy distributions** subject to certain constraints, namely, that the sufficient statistic matches the training data.

✍ **Exercise 2.2** Show that the Gaussian distribution is in the exponential family. Hint:

$$\boldsymbol{\theta} = \left[\frac{\mu}{\sigma^2}, -\frac{1}{2\sigma^2}\right] \tag{2.27}$$

$$Z(\boldsymbol{\theta}) = \sqrt{2\pi\sigma^2} \exp\left(\frac{\mu^2}{2\sigma^2}\right) \tag{2.28}$$

$$h(x) = 1 \tag{2.29}$$

$$\boldsymbol{\phi}(x) = [x, x^2]. \tag{2.30}$$

✍ **Exercise 2.3** Show that the Poisson distribution

$$p(x \mid \lambda) = \frac{\lambda^x}{x!} e^{-\lambda}. \tag{2.31}$$

is in the exponential family. Hint:

$$\theta = \log \lambda \tag{2.32}$$
$$Z(\theta) = e^{\lambda} \tag{2.33}$$
$$h(x) = \frac{1}{x!} \tag{2.34}$$
$$\phi(x) = x. \tag{2.35}$$

2.5.1 Bayesian Machine Learning

Earlier we spoke about the Bayesian interpretation of probability. Let us now discuss how it relates to machine learning. Let $\mathcal{D}$ be some dataset, and $\boldsymbol{\theta}$ be the parameters of our model. According to Bayes' theorem,

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathcal{D})}. \tag{2.36}$$

where the above distributions can be defined as follows

- A **prior distribution** $p(\boldsymbol{\theta})$ over the parameters $\boldsymbol{\theta}$ that define our model.
- A **likelihood** $p(\mathcal{D} \mid \boldsymbol{\theta})$ which defines the probability of observing the data $\mathcal{D}$ given a specific value for $\boldsymbol{\theta}$.
- A **posterior distribution** $p(\boldsymbol{\theta} \mid \mathcal{D})$ over our parameter $\boldsymbol{\theta}$, conditioned on observing the data $\mathcal{D}$

Using the Bayes' rule, we can construct the posterior distribution of a specific model (defined by parameters $\boldsymbol{\theta}$) using our prior and likelihood distributions. In words, we combine prior beliefs about the nature of the data-generating distribution—embodied by $p(\cdot)$—with observations generated from that distribution in order to estimate a model.

Many mainstream NLP techniques focus on finding parameters that maximize the *likelihood* (MLE), i.e., $p(\mathcal{D} \mid \boldsymbol{\theta})$. In a Bayesian setting, however, we care about the *posterior* distribution over the parameters of our model after we have seen all the data, i.e. $p(\boldsymbol{\theta}|\mathcal{D})$. A Bayesian approach offers some advantages over standard MLE. First, the incorporation of a prior produces a regularizing effect on our estimates, which can prevent overfitting. Second, a Bayesian approach produces a distribution over models rather than a single estimate, as we would attain using MLE. Having such a distribution provides us with the means to quantify how confident we are of in our estimate, given the observed data. In other words, we can quantify the **uncertainty** we have about our parameters. Unfortunately, most of the time, being Bayesian comes at the cost of a lot more work; for example, we might have to deal with intractable integrals.

One exception to this situation is when we use **conjugate priors**. The idea here is that, for certain kinds of **likelihood** distributions, we can pick a **prior** distribution such that the **posterior** distribution takes the same form as the prior, which allows us to avoid computing the (often intractable) integral that we encounter when computing the denominator of Equation (2.36). The **exponential family** is the only family of distribution that has conjugacy, so we can use likelihood distributions in the exponential family for this purpose! Intuitively, our ability to analytically solve for the posterior comes from the fact that the likelihood distribution must be a member of the exponential family, which implies that we must be able to summarize the data into a finite vector (recall finite sufficient statistics). See Table 2.4 for some examples.

Definition 2.16

*A prior distribution p is a **conjugate prior** for a likelihood distribution q if the posterior distribution that these two functions induce has the same form as the prior.*

Likelihood	Domain	Parameters modeled by
Bernoulli	$x \in \{0, 1\}$	beta
categorical	$x \in \{1, 2, \ldots, K\}$	Dirichlet
univariate normal	$x \in \mathbb{R}$	normal inverse gamma
multivariate normal	$\mathbf{x} \in \mathbb{R}^k$	normal inverse Wishart

Table 2.4: Likelihood, their domains, and distributions modeling their parameters

As shown earlier, log-linear models are members of the *exponential family*. One can show that for likelihood distributions that are in the exponential family, there are priors who happen to be conjugate for that likelihood. Choosing a conjugate prior is very convenient as it reduces the problem of Bayesian inference to just updating the parameters of the prior. To see this, consider the problem of estimating the mean μ of a Gaussian with known variance σ^2, using only a set of samples $X = \{x_1, \ldots, x_n\}$ from this distribution. Let us choose as prior another Gaussian $\mu \sim \mathcal{N}\left(m, s^2\right)$, where m and s are real positive constants. It can be shown that the posterior is a Gaussian with mean

$$\frac{1}{s^{-2} + n\sigma^{-2}} \left(ms^{-2} + \sigma^{-2} N\bar{X}\right) \tag{2.37}$$

and variance

$$\frac{1}{s^{-2} + n\sigma^{-2}}. \tag{2.38}$$

Observe then how the process of estimating a mean, following a Bayesian approach, reduces to just calculate the two expressions above from the sample mean and the parameters of the prior. In a similar fashion, the process of estimating the parameter of a log-linear model reduces to just updating the prior parameters.

2.5.2 Maximum Entropy

Let p be a discrete distribution over $\mathcal{X} = \{1, \ldots, C\}$. Let $f : \mathcal{X} \to \mathbb{R}$. Suppose that we know $\mathbb{E}[f(x)] = \sum_{x \in \mathcal{X}} p(x) f(x) = \mathbf{F}$. This would mean that we know the expectation, but not the underlying distribution. In this situation, we would like to find the most suitable distribution that conforms the constraints. However, there are potentially an infinite number of solutions that explain the data. In this case, **maximum entropy principle** shows us which probability distribution to choose for inference. The **maximum entropy principle** states that a probability distribution that maximizes the entropy subject to certain constraints represents the current state of knowledge about a system the best. The crux of the argument is that the information-theoretic **entropy** quantifies the uncertainty of a distribution:

$$H(p) = -\sum_{x \in \mathcal{X}} p(x) \log p(x) \tag{2.39}$$

We should opt for the distribution that maximizes the entropy, given whatever information we have available (e.g. mean, covariance, expected counts etc.)

2.5.2.1 Maximum entropy: discrete case

We require a distribution over $\mathcal{X} = \{1, \ldots C\}$. Suppose we know the value of $\mathbb{E}[\phi(x)] = \mathbf{F}$ under the **true** probability distribution for some $\phi(x) = [\phi_1(x), \phi_2(x), \ldots, \phi_K(x)]$. This could be possible in the case that we have a lot of data, and can therefore estimate $\mathbf{F}$ reliably.

Using the **maximum entropy principle**, we want to find a

$$J(p) = -\sum_{x \in X} p(x) \log p(x) \tag{2.40}$$

subject to the following constraints:

$$\begin{aligned} p(x) \geq 0, \forall x \in \mathcal{X} \quad & \text{(non-negativity)} \\ \sum_{x \in \mathcal{X}} p(x) = 1 \quad & \text{(sum-to-one)} \\ \mathbf{F} = \sum_{x \in \mathcal{X}} p(x)\phi(x) \quad & \text{(user-defined constraint)} \end{aligned} \tag{2.41}$$

We can maximize our constraint objective using Lagrange multipliers.

$$J(p, \lambda) = \underbrace{-\sum_{x \in \mathcal{X}} p(x) \log p(x)}_{\text{entropy(original objective)}} + \underbrace{\lambda_0 (1 - \sum_{x \in \mathcal{X}} p(x))}_{\text{sum-to-one constraint(Lagrange multiplier)}} + \underbrace{\sum_k \lambda_k (F_k - \sum_{x \in \mathcal{X}} p(x)\phi_k(x))}_{\text{user-defined constraint(Lagrange multiplier)}} \tag{2.42}$$

Hence,

$$\frac{\partial}{\partial p(x)} J(p, \lambda) = -1 - \log p(x) - \lambda_0 - \sum_k \lambda_k \phi_k(x). \tag{2.43}$$

Setting this to 0, we obtain

$$\log p(x) = -1 - \lambda_0 - \sum_k \lambda_k \phi_k(x) = \log \left(\frac{1}{Z} \exp \left(-\sum_k \lambda_k \phi_k(x) \right) \right), \tag{2.44}$$

where $Z = \exp(1 + \lambda_0)$. Enforcing the sum-to-one constraint, we find that

$$Z = \sum_x \exp \left(-\sum_k \lambda_k \phi_k(x) \right). \tag{2.45}$$

So the discrete distribution that maximizes the entropy w.r.t. known $\mathbb{E}[\phi(x)]$ is

$$p(x) = \frac{1}{\sum_{x \in \mathcal{X}} \exp(-\boldsymbol{\lambda}^\top \boldsymbol{\phi}(x))} \exp(-\boldsymbol{\lambda}^\top \boldsymbol{\phi}(x)), \tag{2.46}$$

which is in the **exponential family**! Notice that this is agnostic to $\boldsymbol{\phi}(x)$, so if we have

$$\boldsymbol{\phi}(x) = [\mathbb{I}[x = 1], \mathbb{I}[x = 2], \ldots, \mathbb{I}[x = C - 1]]^\top, \tag{2.47}$$

we recover the exponential family form of the **categorical distribution**.

Chapter 3 Multi-layer Perceptrons

Let's consider a two features binary classification problem: given a data point $\mathbf{x}$, we want to predict its label y. We consider two features of $\mathbf{x}$: $f_1(\mathbf{x})$ and $f_2(\mathbf{x})$. As a concrete example, let's consider the problem of sentiment analysis, a problem well presented in Pang and Lee 2008.We want to classify the sentiment of a sentence, y as 'positive' or 'negative;' we could take as features $f_1(\mathbf{x})$: if the sentence contains 'not' and $f_2(\mathbf{x})$ = length of the sentence. We can then define the input space as the set of all possible sentences $\mathcal{X}$ = {*I like NLP, I dislike apples, ...}* and the output space as the set of labels $\mathcal{Y} = \{-1, 1\}$, where -1 is for a negative sentiment and 1 for a positive sentiment. In Fig. 3.1, a possible representation of the data is displayed.

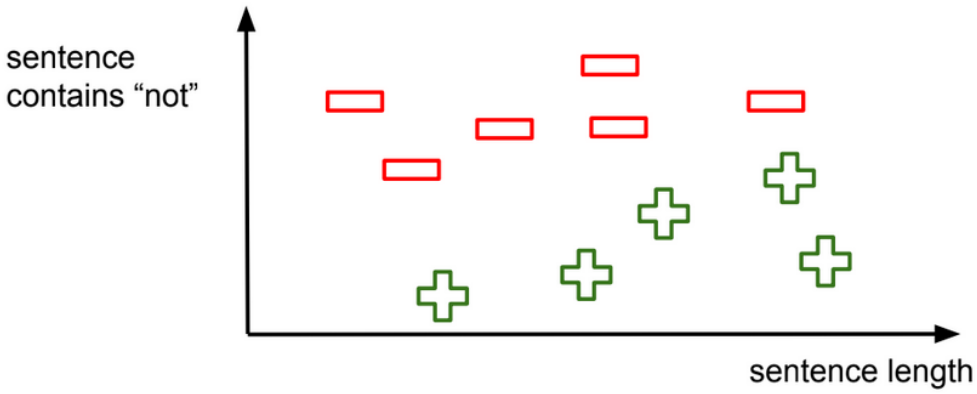


Figure 3.1: Representation of the training data. The + symbols are for the positive class and the - symbols for the negative class.

One simple approach to solving this problem is using a standard log-linear model with: a feature function $\mathbf{f}(\mathbf{x}, y) : \mathcal{X} \times \mathcal{Y} \longrightarrow \mathbb{R}^2$ and a set of learned weights $\boldsymbol{\theta} \in \mathbb{R}^2$. Recall that the log-linear model is then defined as:

$$h_y = \boldsymbol{\theta} \cdot \mathbf{f}(\mathbf{x}, y) \tag{3.1}$$

$$p(y \mid \mathbf{x}) = \text{softmax}(\mathbf{h}, y, T) = \frac{\exp(h_y/T)}{\sum_{y' \in \mathcal{Y}} \exp(h_{y'}/T)} \tag{3.2}$$

We define the sigmoid function σ as $\sigma(x) = \frac{1}{1+\exp(-x)}$ and the function $\mathbf{g}(\mathbf{x}) = \mathbf{f}(\mathbf{x}, y_0) - \mathbf{f}(\mathbf{x}, y_1)$

✍ **Exercise 3.1** Prove that for binary classifcation with two classes y_0 and y_1, the sigmoid is a binary softmax, i.e. that $\text{softmax}(y_0) = \sigma(\boldsymbol{\theta} \cdot \mathbf{g}(x))$.

In our log-linear model for binary classification, we have $p(Y = y_1 \mid \mathbf{x}, \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta} \cdot \mathbf{g}(\mathbf{x}))$. We then choose the **decision boundary** of this model—the boundary determining whether we classify $\mathbf{x}$ as -1 or 1 as $p(Y = y_1 \mid \mathbf{x}, \boldsymbol{\theta}) > 0.5 \Leftrightarrow \boldsymbol{\theta} \cdot \mathbf{g}(\mathbf{x}) > 0$. For a problem with 2 features, this decision boundary is a line. A possible decision boundary is shown in Fig. 3.2. In two dimension, two sets of points are called "linearly separable" if there exists at least one line in the input space that separates the first set from the second set. This is, for example, the case for the points in Fig. 3.2. However, quite often data is not linearly separable. This is for example the case of the training data represented in Fig. 3.3

Can a log-linear model represent this non-linear boundary? Yes, if we change the feature function. The decision boundary is an ellipse, i.e. the equation of the boundary is of the form $\frac{x_1^2}{a^2} + \frac{x_2^2}{b^2} = 1$. With the feature function $g(\mathbf{x}, y) = [x_1, x_1^2, x_2, x_2^2, -1]$ and the weights $\boldsymbol{\theta} = [0, \frac{1}{a^2}, 0, \frac{1}{b^2}, 1]$ we can represent this ellipsoidal decision boundary. Therefore, log-linear models allow to construct non-linear decision boundaries,by introducing non-linear features in the feature function.

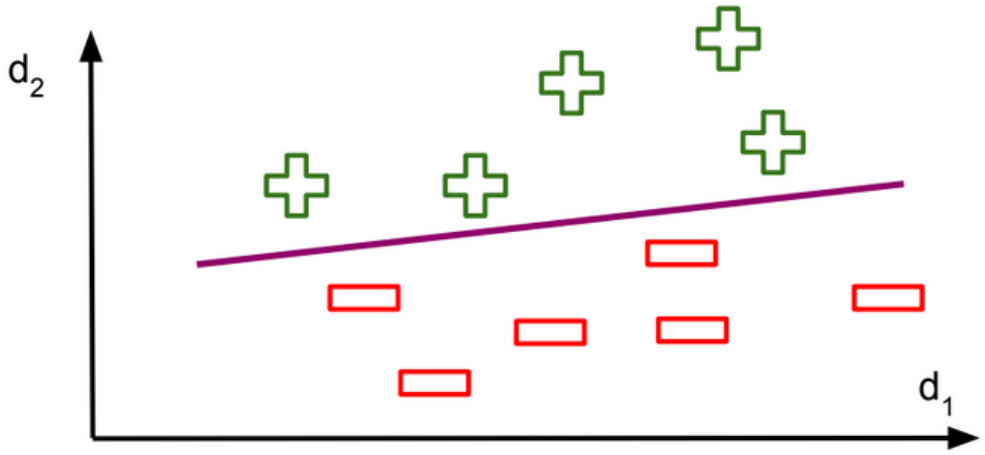


Figure 3.2: Decision boundary represented by a line

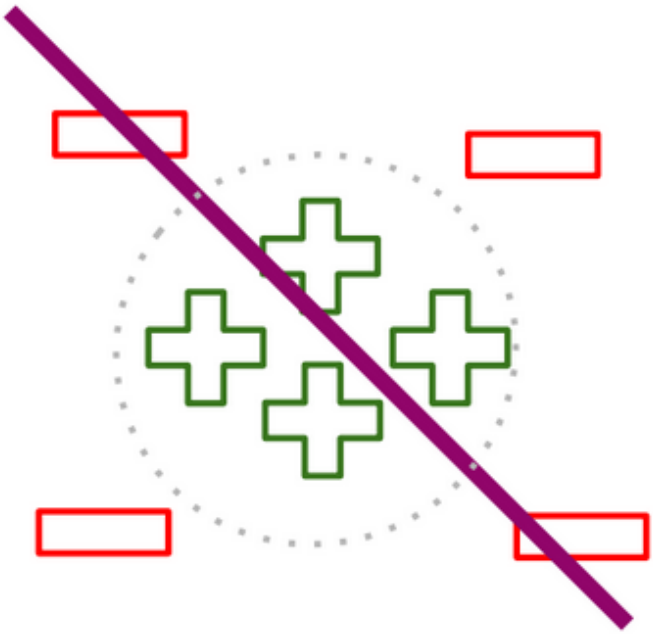


Figure 3.3: Non-linearly separable data

However, there is a limitation with log-linear models: they require that we know which non-linear features we need to extract, which is often not straightforward. To solve this issue, neural networks such as multi-layer perceptrons are useful since they have the ability to *learn* the feature function $\mathbf{f}$ along with the feature weights $\boldsymbol{\theta}$. These multi-layer perceptrons can be used for NLP tasks such as text classification, as proposed by Iyyer et al. 2015

3.1 Multi-layer Perceptrons

Recall that under a log-linear model, we express the probability of an outcome y given input $\mathbf{x}$ as $p(y \mid \mathbf{x}, \boldsymbol{\theta}) = \text{softmax}(\boldsymbol{\theta} \cdot \mathbf{f}(\mathbf{x}, y))$. Under the traditional log-linear modeling scheme, we define $\mathbf{f}$ ourselves. Neural networks allow us to bypass this step, instead jointly learning the feature extraction function and

Sigmoid	**Hyperbolic tangent**	**ReLU**
$\sigma(x) = \frac{1}{1+\exp(-x)}$	$\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$	$\text{ReLU}(x) = \max(x, 0)$

Table 3.1: Common activation functions

corresponding weights at the same time. Under this paradigm, our responsibility shifts from feature engineering to architecture engineering, i.e., creating a model capable of learning e.g., the shape of a complex decision boundary. The multi-layer perceptron (MLP) is one of the simplest neural network architectures, upon which many subsequent models have been built.

3.1.1 Architecture of the MLP

MLPs consist of alternating linear projections and non-linearities. More specifically, each layer of an MLP performs a linear projection of its input, which is then passed through a non-linear **activation function**. This procedure is performed for a set number of rounds N (a.k.a. layers), each with its own linear projection and activation function, in order to attain the final output $h^{(N)}$. We can formulate this process mathematically as follows:

$$\mathbf{h^{(N)}} = \sigma^{(N)}(W^{(N)}...\sigma^{(2)}(W^{(2)}\sigma^{(1)}(W^{(1)}\mathbf{e}(\mathbf{x}))))$$

where the matrix $W^{(i)} \in \mathbb{R}^{d_{i+1} \times d_i}$ is the linear projection at the ith layer, that takes as input a vector of dimension d_i and outputs a vector of dimension d_{i+1}. This output is then passed through activation function $\sigma^{(i)}$ before being passed to the next layer. Finally we can model the probability of an outcome y given input $\mathbf{x}$ as:

$$p(y \mid \mathbf{x}) = \frac{\exp(h_y)}{\sum_{y' \in \mathcal{Y}} \exp(h_{y'})} = \text{softmax}(\mathbf{h}^{(N)}, y)$$

Why did we choose this family of functions? It turns out that MLPs are universal approximators. The universal approximation theorem by Cybenko (Cybenko 1989) states that an MLP with single hidden layer and sigmoidal activation function can represent any function in the unit cube. As the number of layers in an MLP is increased, we gain additional modeling power, which implies that MLPs can represent almost any function that we want.

3.1.2 Activation functions

Activation functions are used to introduce non-linearities. The following functions are given in Table 3.1.

The range of the **sigmoid** function is (0,1), making it a natural choice for modeling probability distributions. As we have previously seen, the sigmoid function is a binary softmax: it can be used in the case of binary classification on the last layer. The bounded range of the sigmoid ensures that the output will not blow-up, when we have a sequence of sigmoid functions. For high input values the derivative of the sigmoid becomes very small, which can lead to the problem of vanishing gradient in deep neural networks: the gradient becomes almost zero.

✍ **Exercise 3.2** Prove that the derivative of the sigmoid is :

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

The **hyperbolic tangent** is a shifted sigmoid with values between -1 and 1. As with the sigmoid, the hyperbolic tangent is subject to vanishing gradient issues.

✍ **Exercise 3.3** Prove that the derivative of the hyperbolic tangent is : $\tanh'(x) = 1 - \tanh^2(x)$.

For all negative values the **ReLU** function is zero, which leads to a sparse computation graph. This is more computationally efficient, since a lot of values are zeros. Moreover the function doesn't saturate for high values, which reduces the risk of having vanishing gradients.

3.1.3 Training an MLP

Just like the log-linear model from chapter 3, we maximize the log-likelihood of the training data with a gradient-based method. Thanks to chapter 2 we have an efficient way of computing the gradient with backpropagation.

Consider a neural network parametrized by parameters $\boldsymbol{\theta}$. We define a differentiable loss function $L : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ which takes as arguments our predictions and the ground truth labels. We optimize the parameters of the network by minimizing this loss function. This optimization can be done by stochastic gradient descent for instance. Let's consider a training datapoint $(\mathbf{x}, y)$. The forward propagation produces a prediction $\hat{y}$. We compute the loss $L(y, \hat{y})$. Then we perform the backpropagation step. Each parameter θ_i get updated :

$$\theta_i \longleftarrow \theta_i - \eta \frac{\partial L(y, \hat{y})}{\partial \theta_i}$$

with η the learning rate.

✍ **Exercise 3.4** Let's consider a neural network with N input nodes, H nodes in the hidden layer with matrix weights $W^{(1)}$ and bias $b^{(1)}$ and a scalar output y with weights $W^{(2)}$. We can formulate the layers of our network as:

$$h_i = ReLU(W_i^{(1)} \cdot \mathbf{x} + b_i^{(1)}) = ReLU(\textstyle\sum_{j=1}^{N} W_{ij}^{(1)} x_j + b_i^{(1)}) \text{ for i} \in \{1, ..., H\} \tag{3.3}$$

$$\hat{y} = W^{(2)} \cdot \mathbf{h} \tag{3.4}$$

with the loss function $l : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$. For each parameter, write the explicit formula of the gradient of the loss with respect to this parameter.

3.2 Embedding Natural Language into Vector Spaces

For the moment, we presented the multi-layer perceptrons, that can extract features and realize a prediction from vector data. It remains an important question : how can we give language to such a model?

Neural networks require vectors of real numbers as inputs. Consequently, we must transform raw text data into real-valued vector representations before using such models. Generally, text data is passed through an entire pipeline before we begin to work with computational models—such pipelines can include tokenization, stemming, and punctuation removal, among other operations. The final part of this pipeline is the encoding of text into vectors, which we focus on in this section.

3.2.1 One-hot encoding

One-hot encoding is perhaps the simplest way to encode a word as a vector. Given a vocabulary of words V, we associate each word $w \in V$ of this vocabulary with an index. A word can then be represented as a vector $\mathbf{e}(w)$ of length |V| with 1 at the position of its index and 0 elsewhere.

3.2.2 Skip-gram model

There are several issues with the one-hot encoding scheme. First, the high dimensionality of the resulting vectors is computationally expensive. Second, this encoding does not have the ability to reflect semantic or syntactic relationships between words in mathematical terms, as e.g., standard distance metrics between words all yield the same value. Another way to encode words that partially addresses these problems is through continuous word embeddings. These representations are typically learned through an auxiliary task. Here we present in detail one model for learning such word embeddings: the **skip-gram** model.

In the skip-gram model, the objective is to predict context (i.e., surrounding) words given a specific word. Note that the inverse of this task is also a valid objective, that is from the context words, we can predict the current word. This model is known as CBOW.

Formally, let us consider a vocabulary of words V for which we wish to create embeddings $e(w) \in \mathbb{R}^N$. We assume our skip-gram model considers the T preceding words and the T words following our input word in the prediction step.

The first step in the algorithm for training a skip-gram model is the processing of each sample in our training data into a (multi-) set $\mathcal{D}$ of pairs of input and context words (w_i, w_t), where $t \in \{-T, -(T-1), \ldots, -1, 1, \ldots, T\}$ indicates the position of w_t with respect to w_i in the training sample. The goal is to optimize $\sum_{(w_i,w_t)\in\mathcal{D}} p(w_t \mid w_i)$. We can model the probability $p(w_t \mid w_i)$ as a log-linear model :

$$p(w_t \mid w_i) = \frac{1}{Z(w_i)} \exp(\mathbf{e}_{wrd}(w_i) \cdot \mathbf{e}_{ctx}(w_t))$$

with $Z(w) = \sum_{w'\in V} \exp(\mathbf{e}_{wrd}(w) \cdot \mathbf{e}_{ctx}(w'))$

We project the one-hot encoding of our input word $\mathbf{e}_{oh}(w_i) \in \mathbb{R}^{|V|}$ with a weight matrix $W \in \mathbb{R}^{|V|\times N}$. This projection is then multiplied by weight matrix W' and the softmax function is applied, resulting in a vector that represents the predicted probability of $w \in V$ appearing in the size $2T$ context window around word w_i. Our goal is to maximize the log-likelihood of our data: $\log p(w_t \mid w_i, W, W')$. The log-likelihood can be written as :

$$\log p(w_t \mid w_i) = \mathbf{e}_{wrd}(w_i) \cdot \mathbf{e}_{ctx}(w_t) - \log Z(w_i) = W_i \cdot (W')_t^\top - \log Z(w_i)$$

. The optimization is typically done using gradient-descent methods.

However $Z(w) = \sum_{w'\in V} \exp(\mathbf{e}_{wrd}(w) \cdot \mathbf{e}_{ctx}(w'))$ can take a long time to compute for large V. We can instead use **negative sampling**: rather than computing the $|V|$-dimensional probability distribution via the softmax, we use the sigmoid function between the ground truth label and the label of a word drawn randomly from V.

Formally, for each pair of words in our training data (w_i, w_t), we randomly sample (with replacement) a set C^- from V. Our objective under the negative sampling approach becomes

$$\arg\max_{W,W'} \sum_{(w_i,w_t,C^-)} \left(\log(p(w_t \mid w_i)) + \sum_{w^-\in C^-} \log(1 - p(w^- \mid w_i)) \right)$$

where we compute $p(w_t \mid w_i)$ using the sigmoid function: $p(w_t \mid w_i) = \sigma(\mathbf{e}_{wrd}(w_i) \cdot \mathbf{e}_{ctx}(w_t))$.

At the end of training, we end up with two matrices W and W', that represent that represent the embeddings of the word as target and context respectively. We usually choose the matrix W for our embedding. It is also possible to concatenate the target and context embeddings.

3.2.3 Combining Embeddings

To encode a full sentence $\mathbf{x}$, we apply a pooling method on its word encodings:

- sum pooling : $\mathbf{e}(\mathbf{x}) = \sum_{w \in \mathbf{x}} \mathbf{e}(w)$
- mean pooling : $\mathbf{e}(\mathbf{x}) = \frac{1}{|x|} \sum_{w \in \mathbf{x}} \mathbf{e}(w)$
- max pooling : $\mathbf{e}(\mathbf{x}) = \max_{w \in \mathbf{x}} \mathbf{e}(w)$

3.2.4 Evaluating Word Embeddings

How can we evaluate the quality of our word embeddings? One useful metric is the cosine similarity.

Definition 3.1

*The **cosine similarity** between two vectors* $\mathbf{x}$ *and* $\mathbf{y}$ *is defined as :*

$$\text{sim}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{||\mathbf{x}||||\mathbf{y}||}$$

♣

The cosine similarity quantifies the similarity between two vectors. Two similar words are supposed to be represented by similar vectors. Therefore, to evaluate a model, we can check that embeddings of similar words have a high cosine similarity.

We can also use word analogies. Indeed, word embeddings also capture relation between words. For example the vector $\mathbf{e}(\text{king}) - \mathbf{e}(\text{man}) + \mathbf{e}(\text{woman})$ is supposed to be close to the vector $\mathbf{e}(\text{queen})$. We can check the cosine similarity between such vectors to evaluate an embedding model.

Chapter 4 Language Modeling

So far we have implicitly assumed that we can efficiently sum over the output classes, as is the case for e.g. sentiment analysis where we predict a label within a small set of discrete values. In many NLP tasks however, we are interested in predicting **structured objects**, such as strings, trees or graphs, where the output space is large. In other words, we are as before interested in the prediction problem

$$p(\mathbf{y} \mid \mathbf{x}) = \frac{\exp\{\text{score}(\mathbf{y}, \mathbf{x})\}}{\sum_{\mathbf{y}' \in \mathcal{Y}} \exp\{\text{score}(\mathbf{y}', \mathbf{x})\}} \tag{4.1}$$

where, as usual, $\text{score}(\cdot, \cdot)$ is a measure of the fit of the label $\mathbf{y}$ to the data $\mathbf{x}$, either parameterized by a neural network or obtained via feature engineering. Only now, $|\mathcal{Y}|$ is exponentially (or even infinitely) large, and naive computation of the normalizer therefore often becomes intractable. We call this task **structured prediction**. In structured prediction, we often require efficient algorithms to be able to iterate over the output space, leading us to turn to tools from theoretical computer science.

4.1 The Language Modeling Task

In this chapter, we consider a first example of such a structured prediction task, namely **language modeling**. In language modeling we fit a probability distribution over all possible instances of natural language sequences drawn from a vocabulary. More formally, consider a vocabulary $\mathcal{V}$. Our task is to model a probability distribution over $\mathcal{Y} = \{\textsc{bos} \circ \mathbf{v} \circ \textsc{eos} \mid \mathbf{v} \in \mathcal{V}^*\}$. Two special tokens BOS and EOS, denoting beginning and end of sequence respectively, are concatenated with each output string for reasons that will become apparent. The * operator is the Kleene closure (or Kleene star) operator over sets.

Definition 4.1 (Kleene closure)

Let $\mathcal{V}$ be a set and let ε denote the empty string. Define $\mathcal{V}^0 \stackrel{\text{def}}{=} \{\varepsilon\}$, $\mathcal{V}^1 \stackrel{\text{def}}{=} \mathcal{V}$ and by recursion

$$\mathcal{V}^{i+1} \stackrel{\text{def}}{=} \{w \circ v : w \in \mathcal{V}^i, v \in \mathcal{V}\} \quad \forall i \in \mathbb{N} \tag{4.2}$$

The Kleene closure on $\mathcal{V}$ is then

$$\mathcal{V}^* \stackrel{\text{def}}{=} \bigcup_{\forall i \in \mathbb{N}} \mathcal{V}^i = \mathcal{V}^0 \cup \mathcal{V}^1 \cup \mathcal{V}^2 \cup \cdots \tag{4.3}$$

♣

Note that $\mathcal{V}^*$ is infinitely large. Even when assuming an upper bound N on the sequence length, resulting in the set $\mathcal{V}^{\leq N} = \bigcup_{i=0}^{N} \mathcal{V}^i$, we get a cardinality that grows faster than exponentially with N.

Example 5 *Consider the vocabulary $\mathcal{V} = \{foo, bar\}$. Following the definition of $\mathcal{Y}$ above, we get*

$$\begin{aligned} \mathcal{Y} = \{&\textsc{bos}\ \varepsilon\ \textsc{eos}\} \cup \{\textsc{bos}\ foo\ \textsc{eos}, \textsc{bos}\ bar\ \textsc{eos}\} \cup \\ \cup \{&\textsc{bos}\ foo\ foo\ \textsc{eos}, \textsc{bos}\ foo\ bar\ \textsc{eos}, \textsc{bos}\ bar\ foo\ \textsc{eos}, \textsc{bos}\ bar\ bar\ \textsc{eos}\} \cup \cdots = \\ = \{&\textsc{bos}\ \varepsilon\ \textsc{eos}, \textsc{bos}\ foo\ \textsc{eos}, \textsc{bos}\ bar\ \textsc{eos}, \textsc{bos}\ foo\ foo\ \textsc{eos}, \textsc{bos}\ foo\ bar\ \textsc{eos}, \ldots\} \end{aligned} \tag{4.4}$$

An alternative perspective on the language modeling task is as a normalized weighting of the edges in a prefix tree (also called a trie), which in this case is instantiated as having BOS as the root node and each element in the set $\mathcal{V} \cup \{\textsc{eos}\}$ as child nodes, recursively for all nodes but EOS. Explicitly, all nodes have the children $\mathcal{V} \cup \{\textsc{eos}\}$, except all instances of EOS which are leaf nodes. There is thus a countably infinite number of paths in

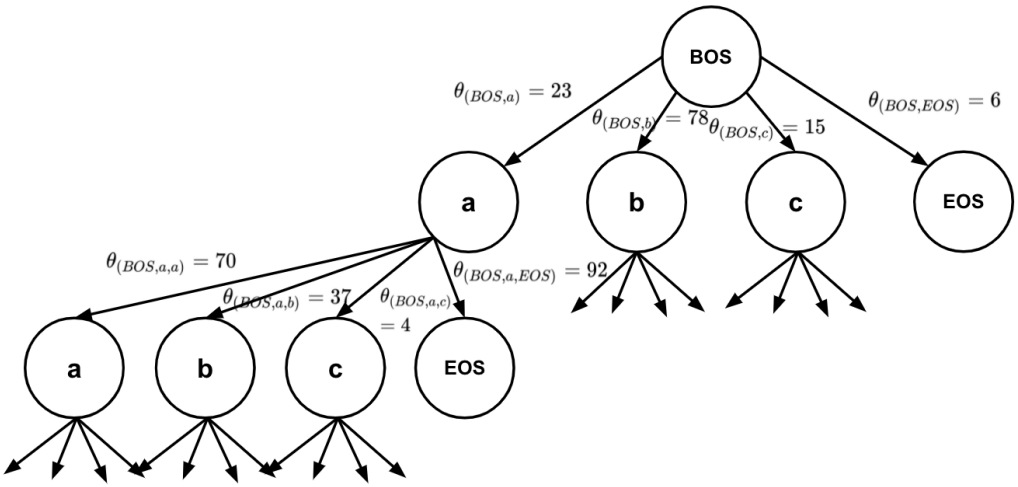


Figure 4.1: Example of a prefix tree.

the prefix tree, all with finite length. The weightings of the edges consist of values $\theta \in \mathbb{R}^+$. See Fig. 4.1 for an example.

Under this paradigm, we can take the probability of a string $\mathbf{y} \in \mathcal{Y}$ to be proportional to the weights along the path in the prefix tree:

$$p(\mathbf{y}) \propto \prod_{t=1}^{|\mathbf{y}|} \theta_{\mathbf{y}_{\leq t}} \tag{4.5}$$

Here, $\mathbf{y}_{\leq t} := (y_1, ..., y_t)$ denotes the first t elements of the sequence $\mathbf{y}$, which uniquely identifies the paths in the prefix tree. As such, $\theta_{\mathbf{y}_{\leq t}}$ is the weight associated with the transition from y_{t-1} to y_t, conditioned on having already taken the path $y_1, y_2, ..., y_{t-1}$. In order to compute the exact distribution $p(\mathbf{y})$, we must normalize Equation (4.5) such that the probability of all $\mathbf{y} \in \mathcal{Y}$ sums to 1. This normalization constant Z can be computed as:

$$Z = \sum_{\mathbf{y}' \in \mathcal{Y}} \prod_{t=1}^{|\mathbf{y}'|} \theta_{\mathbf{y}'_{\leq t}} \tag{4.6}$$

Note that since $|\mathcal{Y}|$ is infinite, Z is an infinite sum. To avoid the need to explicitly compute Z at a global level, which in some cases is impractical, if not infeasible[1], we can instead turn to local normalization. The local normalization scheme takes advantage of the fact the the probability of a sequence $p(\mathbf{y})$ can be computed as

$$p(\mathbf{y}) = \prod_{t=1}^{|\mathbf{y}|} p(y_t \mid \mathbf{y}_{<t}) \tag{4.7}$$

by the *chain rule of probability*. Accordingly, we can instead model the transition out of each node in our prefix tree as a probability distribution in and of itself. More formally, for any given node in our prefix tree, the weights of all outgoing edges are normalized to sum to 1. See Fig. 4.2 for an example. Consequently, normalization need take place solely at each transition in the prefix tree:

$$p(\mathbf{y}) = \prod_{t=1}^{|\mathbf{y}|} \frac{\theta_{\mathbf{y}_{\leq t}}}{Z_t(\mathbf{y}_{<t})} \tag{4.8}$$

[1]For Markovian models it is possible to compute such a constant using dynamic programming, but there does not currently exist any general-purpose algorithm for it.

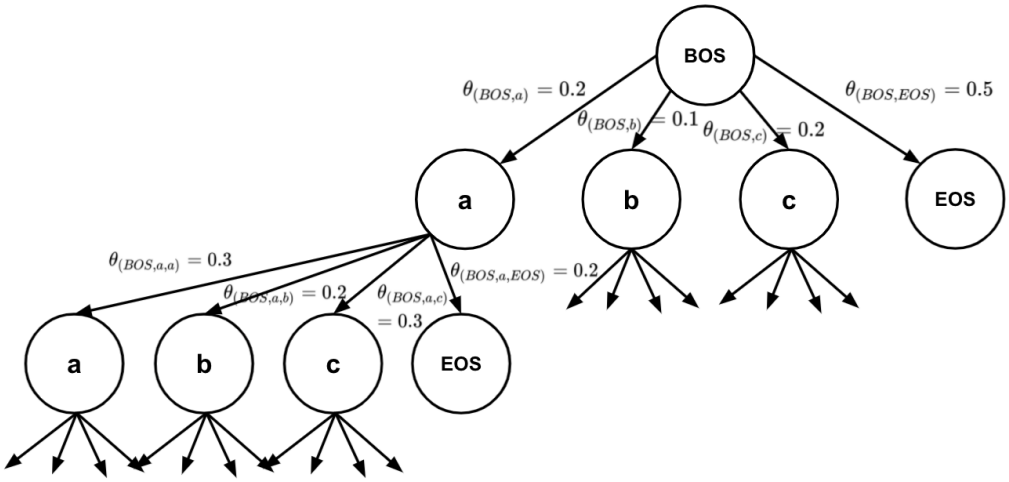


Figure 4.2: Example of a locally normalized language model represented as a prefix tree.

with the normalizing factor $Z_t(\mathbf{y}_{<t})$ now being dependent on the path along the prefix tree

$$Z_t(\mathbf{y}_{<t}) = \sum_{\mathbf{y}' \in \mathcal{V}} \theta_{(y_1,\ldots,y_{t-1},y')} \tag{4.9}$$

Note that this leads to a globally normalized model with $Z = 1$. We have the necessary condition that every node has EOS as descendant. If this condition is not met, we get a deficient distribution, i.e. one where the probability does not sum to one.

Definition 4.2 (Tight and non-tight language model)

A locally normalized language model is called tight. ♣

4.2 n-gram Language Modeling

In the above formulation, the weights for transition to y_t are computed conditioned on the entire prefix $\mathbf{y}_{<t}$. In order to make parameter estimation of the language model more efficient, we can make a simplifying assumption that each token only depends on a finite history. We call this the **n-gram assumption**.

Definition 4.3 (n-gram assumption)

For a language model $p(\mathbf{y})$, we define the n-gram assumption as

$$p(y_t \mid \mathbf{y}_{<t}) \stackrel{\text{def}}{=} p(y_t \mid y_{t-1}, \ldots, y_{t-n+1}) \tag{4.10}$$

where $(y_{t-1}, \ldots, y_{t-n+1})$ is called the history. In plain English, this means that the probability of a token only depends on the previous $n-1$ tokens. ♣

These probabilities can then be plugged into Equation (4.7) to compute the probability of an entire sequence $\mathbf{y}$. The question now is, how can we estimate the model weights such that these conditional probabilities can be determined? Parameter estimation can be done with relative frequency counts or with neural approaches, discussed below in §4.2.1 and §4.2.2 respectively.

4.2.1 Pre-neural n-gram language modeling

The traditional and most simple approach to language modeling uses relative frequencies, as estimated from a training corpus. Specifically, we estimate the probability of a particular token occurring after the prior context $\mathbf{y}_{<t}$ (or under the n-gram assumption, after $(y_{t-n+1}, ..., y_{t-1})$) as the proportion of times the context was followed by that token in the training corpus:

$$\hat{p}(y_t \mid y_{t-1}, ..., y_{t-n+1}) = \frac{\text{count}(y_{t-n+1}, ..., y_{t-1}, y_t)}{\text{count}(y_{t-n+1}, ..., y_{t-1})} \tag{4.11}$$

where $\text{count}(\cdot)$ is simply a function that counts the occurrences of a sequence of words $\cdot$ (dependence on the training corpus is left implicit). Note that this corresponds to the *maximum likelihood estimate* of $p(y_t \mid y_{t-1}, ..., y_{t-n+1})$.

✍ **Exercise 4.1** Show that for the unigram model, $\hat{p}(y) = \frac{\text{count}(y)}{\sum_{y' \in \mathcal{V}} \text{count}(y')}$ is the maximum likelihood estimator for $p(y)$. Assume that each token is sampled independently from a categorical distribution over the vocabulary $\mathcal{V}$. Apply Equation (2.5) and use Lagrange multipliers to solve the optimization problem.

We illustrate the estimation procedure with an example.

Example 6 *Consider the following training corpus:*

BOS I do love nlp EOS
BOS nlp you love EOS
BOS do you love nlp EOS
BOS you like nlp EOS

We can estimate the probability of the sequence "you love nlp" with a bigram language model as

$$\begin{aligned}
\hat{p}(\textit{BOS you love nlp EOS}) &= \hat{p}(\textit{you} \mid \textit{BOS}) \cdot \hat{p}(\textit{love} \mid \textit{you}, \textit{BOS}) \cdot \\
&\quad \cdot \hat{p}(\textit{nlp} \mid \textit{love, you}, \textit{BOS}) \cdot \hat{p}(\textit{EOS} \mid \textit{nlp,love,you}, \textit{BOS}) \\
&= \hat{p}(\textit{you} \mid \textit{BOS}) \cdot \hat{p}(\textit{love} \mid \textit{you}) \cdot \hat{p}(\textit{nlp} \mid \textit{love}) \cdot \hat{p}(\textit{EOS} \mid \textit{nlp}) \\
&= \frac{\text{count}(\textit{BOS you})}{\text{count}(\textit{BOS})} \cdot \frac{\text{count}(\textit{you love})}{\text{count}(\textit{you})} \cdot \frac{\text{count}(\textit{love nlp})}{\text{count}(\textit{love})} \cdot \frac{\text{count}(\textit{nlp EOS})}{\text{count}(\textit{nlp})} \\
&= \frac{1}{4} \cdot \frac{2}{3} \cdot \frac{2}{3} \cdot \frac{3}{4} = \frac{1}{12}
\end{aligned}$$

where we first used the chain rule of probability, then the bigram assumption and finally inserted the relative frequency estimation.

Note that in the setting of Example 6, we would assign a probability of zero to the sentence "I love nlp", since "love" does not occur after "I" in the training corpus. Naturally, we would like our model to be able to generalize to contexts unseen in the training corpus. One solution would be through **smoothing techniques**. One very simple smoothing technique is Laplace (or add-λ) smoothing. Under this method, we add pseudo counts λ to all n-grams in our corpus. Our estimates then become:

$$\hat{p}_\lambda(y_t \mid y_{t-1}, ..., y_{t-n+1}) = \frac{\text{count}(y_{t-n+1}, ..., y_{t-1}, y_t) + \lambda}{\sum_{w' \in \mathcal{V}} \text{count}(y_{t-n+1}, ..., y_{t-1}, w') + |\mathcal{V}|\lambda} \tag{4.12}$$

This estimate has a Bayesian interpretation, namely as the mode of the posterior distribution—called the *maximum a posteriori* (MAP) estimate—over the probability weights when we consider a symmetric Dirichlet distribution with parameter λ as our prior distribution over the weights. Another way to solve the zero frequency n-gram

problem is with **back-off**, where we resort to estimates from n-grams of lower order if we have no statistical basis for the higher-order n-gram estimate. An example is the back-off method presented in Katz 1987:

$$\hat{p}_{bo}(y_t \mid y_{t-1}, ..., y_{t-n+1}) = \begin{cases} p^*(y_t \mid y_{t-1}, ..., y_{t-n+1}), & \text{if count}(y_{t-n+1}, ..., y_{t-1}, y_t) > 0 \\ \alpha_{n-1} p_{bo}(y_t \mid y_{t-1}, ..., y_{t-(n-1)+1}), & \text{otherwise} \end{cases} \tag{4.13}$$

where p^* are discounted[2] relative frequency count estimates (in order to save probability mass for lower order n-grams). In yet another approach, **interpolation**, we combine the estimates from different n-gram orders

$$\hat{p}_i(y_t \mid y_{t-1}, ..., y_{t-n+1}) = \alpha_1 \hat{p}(y_t) + \alpha_2 \hat{p}(y_t \mid y_{t-1}) + ... + \alpha_n \hat{p}(y_t \mid y_{t-1}, ..., y_{t-n+1}) \tag{4.14}$$

While all sensible workarounds, these traditional statistical approaches require very large corpora if we are to gain the benefits of using n-grams of high order. More specifically, we need to model $|\mathcal{V}|^{n-1}$ histories, which cannot be estimated from a realistically-sized sample of text.

✍ **Exercise 4.2** How many parameters are needed to model an n-gram language model from a vocabulary $\mathcal{V}$?

4.2.2 Neural n-gram language modeling

The key idea behind neural n-gram models is to tie the parameters between contexts, by the use of word embeddings. In so doing we can jointly estimate a large number of log-linear models

$$p(y_t \mid y_{t-n+1}, ..., y_{t-1}) = \frac{\exp(\boldsymbol{\omega}_{y_t} \cdot \mathbf{h}_t)}{\sum_{y' \in V} \exp(\boldsymbol{\omega}_{y'} \cdot \mathbf{h}_t)} \tag{4.15}$$

with $\boldsymbol{\omega}_y \in \mathbb{R}^d$ being *word vectors* and $\mathbf{h}_t \in \mathbb{R}^d$ being n-gram *context vectors* encoding the context of the previous $n-1$ words. The vectors $\boldsymbol{\omega}_y$ are parameters that we can learn with gradient-based optimization and backpropagation, by maximizing the log-likelihood

$$\sum_{\mathbf{y}^{(i)} \in \mathcal{D}} \log p(\mathbf{y}^{(i)}) = \sum_{\mathbf{y}^{(i)} \in \mathcal{D}} \sum_{t=1}^{|\mathbf{y}^{(i)}|} \left(\boldsymbol{\omega}_{y_t^{(i)}} \cdot \mathbf{h}_t - \log(\sum_{y' \in V} \exp(\boldsymbol{\omega}_{y'} \cdot \mathbf{h}_t)) \right) \tag{4.16}$$

where $\mathcal{D}$ is a corpus of sequences $\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, ..., \mathbf{y}^{(N)}$.

In addition, we need to model the context vectors. This can be done in many ways, but as an illustrating example we present the first successful neural n-gram model, which was introduced in Bengio et al. 2003. Consider a 4-gram model with contexts y_{t-1}, y_{t-2} and y_{t-3}, a function mapping one-hot encodings of words to embeddings $\mathbf{e}(\cdot)$ and an arbitrary MLP $\mathbf{f}(\cdot)$ with learned parameters $\boldsymbol{\theta}$. The context vectors are modeled as

$$\mathbf{h}_t = \mathbf{f}(\mathbf{e}(\text{history})) = \mathbf{f}([\mathbf{e}(y_{t-1}); \mathbf{e}(y_{t-2}); \mathbf{e}(y_{t-3})]) \tag{4.17}$$

i.e., we feed a concatenation of the context word embeddings as input to the MLP. The parameters of $\mathbf{f}(\cdot)$ and the mapping $\mathbf{e}(\cdot)$ are learned jointly with the model (although there is nothing stopping us from using any other pre-trained representations as our embedding map). Do note the similarity to the skip-gram model introduced in §3.2.2.

✍ **Exercise 4.3** Consider a neural n-gram model with (1) a fixed embedding map $\mathbf{e} : \mathcal{V} \to \mathbb{R}^m$, (2) context vectors $\mathbf{h}_t \in \mathbb{R}^d$ parameterized by an MLP with one hidden layer having dimension h and arbitrary activation functions $\sigma(\cdot)$, and (3) word vectors $\boldsymbol{\omega}_y \in \mathbb{R}^d$ for each $y \in \mathcal{V}$. Write down the full formula for the log-likelihood function

[2] Discounting refers to the technique of assigning probabilities to unseen events. Using discounting, the events that are unseen during training can still be predicted with non-zero probabilities during inference time.

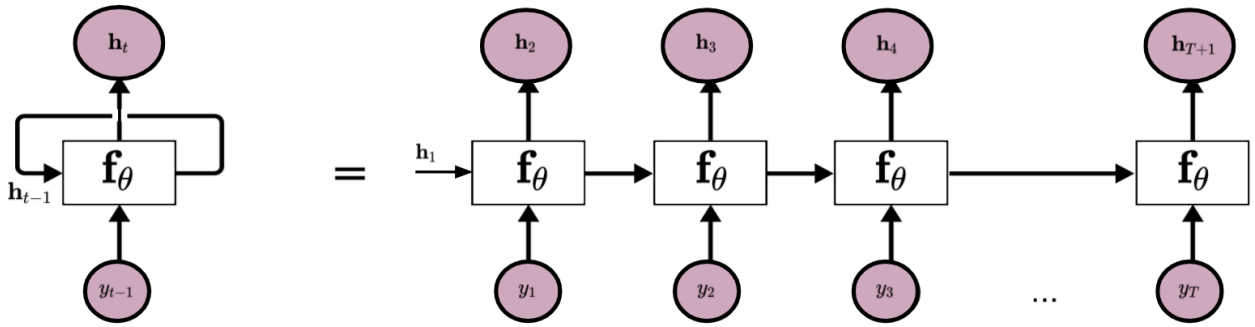


Figure 4.3: The recurrent neural network architecture.

of a corpus $\mathcal{D}$. How many parameters does this model have?

4.3 Recurrent Neural Networks

The n-gram assumption is limiting in many ways. Consider for instance the German language, where the second part of a compound verb phrase always goes last in the sentence, while still being dependent on the earlier part. With MLPs we are inherently limited to making the n-gram assumption since they require a fixed input. Although we technically could model long histories with the neural n-gram model presented above, the fixed input dimension of the MLP would grow quite large.

Instead, let us consider a model where the input dimension does not increase with the sequence length. To accomplish this, we feed the inputs sequentially to the model. We can then encode the entire history $(y_1, ..., y_{t-1})$ in a context vector $\mathbf{h}_t$. With this setup, we have a language model that can (theoretically) capture infinite history by the principles of *parameter sharing*. That is, we can now model

$$p(y_t \mid y_1, ..., y_{t-1}) = \frac{\exp(\boldsymbol{\omega}_{y_t} \cdot \mathbf{h}_t)}{\sum_{y' \in V} \exp(\boldsymbol{\omega}_{y'} \cdot \mathbf{h}_t)} \tag{4.18}$$

for an arbitrary t. This is the idea behind **recurrent neural networks** (RNNs).

In RNNs, each hidden state $\mathbf{h}_t$ is recurrently modeled as

$$\mathbf{h}_t = \mathbf{f}(y_{t-1}, \mathbf{h}_{t-1}) \tag{4.19}$$

with the inputs y_{t-1} being fed sequentially to the model. See Fig. 4.3 for an illustration, where we observe how the recurrence function is unrolled over the input sequence. One instantiation of such a function, the Elman recurrence (Elman 1990), is

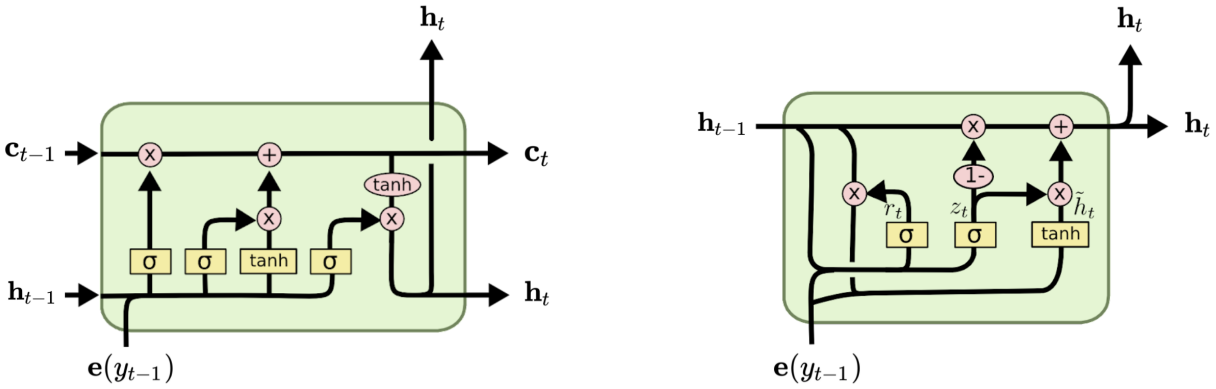
$$\mathbf{h}_t = \sigma(W_1 \mathbf{h}_{t-1} + W_2 \mathbf{e}(y_{t-1})) \tag{4.20}$$

with weight matrices $W_1 \in \mathbb{R}^{d \times d}, W_2 \in \mathbb{R}^{d \times m}$. Alternatively the inputs can be concatenated as

$$\mathbf{h}_t = \sigma(W[\mathbf{h}_{t-1}; \mathbf{e}(y_{t-1})]) \tag{4.21}$$

with weight matrix $W \in \mathbb{R}^{d \times (d+m)}$.

The derivatives of recurrent neural networks can be computed with **backpropagation through time** (Werbos 1990), which is really just backpropagation on the unrolled version of the network in Equation (4.19). Consider for instance the derivative of $\mathbf{h}_t$ with respect to the parameters $\boldsymbol{\theta}$, for recurrence $\mathbf{h}_t = \sigma(\mathbf{h}'_t) = \sigma(g(y_{t-1}, \mathbf{h}_{t-1}))$, with $g(\cdot, \cdot)$ being some function of the hidden state and input. We sum over all the paths in the computation



(a) The LSTM cell. **(b)** The GRU cell.

Figure 4.4: More advanced recurrent neural network architectures.

graph as follows

$$\frac{\partial \mathbf{h}_t}{\partial \boldsymbol{\theta}} = \sum_{i=2}^{t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_i} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}'_i} \frac{\partial \mathbf{h}'_i}{\partial \boldsymbol{\theta}} \tag{4.22}$$

$$= \sum_{i=2}^{t} \left(\prod_{j=t}^{i+1} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}'_j} \frac{\partial \mathbf{h}'_j}{\partial \mathbf{h}_{j-1}} \right) \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}'_i} \frac{\partial \mathbf{h}'_i}{\partial \boldsymbol{\theta}} \tag{4.23}$$

$$= \sum_{i=2}^{t} \left(\prod_{j=t}^{i+1} \text{diag}(\sigma'(\mathbf{h}'_j)) \frac{\partial g_{\boldsymbol{\theta}}(y_{j-1}, \mathbf{h}_{j-1})}{\partial \mathbf{h}_{j-1}} \right) \text{diag}(\sigma'(\mathbf{h}'_i)) \cdot \frac{\partial g(y_{i-1}, \mathbf{h}_{i-1})}{\partial \boldsymbol{\theta}} \tag{4.24}$$

where we unroll $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_i}$ for each path. Note that $\partial g(y_{j-1}, \mathbf{h}_{j-1}) / \partial \mathbf{h}_{j-1}$ does not depend on the time step $j-1$ for the Elman recurrence. Taking for instance the version in Equation (4.20), we get that

$$\frac{\partial g(y_{j-1}, \mathbf{h}_{j-1})}{\partial \mathbf{h}_{j-1}} = W_1 \tag{4.25}$$

which is independent of time step.

This introduces a practical problem when training with longer sequences. As we take the derivative, the same parameters are multiplied several times, as will be the case with W_1 above. If this parameter matrix is small, the small values will be compounded as the distance in time steps increases. The derivatives will then shrink, causing what is known as the **vanishing gradients problem**. The converse is true if the parameter matrix is large, causing the derivatives to explode (**exploding gradients problem**). These issues are frequently encountered when training standard vanilla RNNs like the Elman recurrence, and lead to instabilities in the gradient update step.

The problems really stem from the recursive derivative of $\mathbf{h}_t$ with respect to earlier states $\mathbf{h}_i$, where the same values create a compound effect. In simple terms, we would like the gradients to be able to balance out (as is the case in e.g. MLPs where parameters from different layers have different values). One way to allow for this, and thereby at least partially remedy the issues, is via more complex architectures. One common such instance is the **long short-term memory** LSTM (Hochreiter and Schmidhuber 1997), which has both a hidden state $\mathbf{h}_t$ and a memory cell state $\mathbf{c}_t$, depicted to the left in Fig. 4.4 and parameterized as

$$\begin{aligned}
\mathbf{f}_t &= \sigma(W_f[\mathbf{h}_{t-1}; \mathbf{e}(y_{t-1})] + \mathbf{b}_f) && \textit{forget gate} \\
\mathbf{i}_t &= \sigma(W_i[\mathbf{h}_{t-1}; \mathbf{e}(y_{t-1})] + \mathbf{b}_i) && \textit{input gate} \\
\tilde{\mathbf{c}}_t &= \tanh(W_c[\mathbf{h}_{t-1}; \mathbf{e}(y_{t-1})] + \mathbf{b}_c) && \textit{update candidate} \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t && \textit{update cell state} \\
\mathbf{o}_t &= \sigma(W_o[\mathbf{h}_{t-1}; \mathbf{e}(y_{t-1})] + \mathbf{b}_o) && \textit{output gate} \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t) && \textit{update hidden state}
\end{aligned}$$

Note that with this structure, $\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}}$, and thereby also $\frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_{t-1}}$ and $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$, can differ in value between time steps. This is allowed by the gating functions of the network.

✍ **Exercise 4.4** Derive the derivative $\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}}$ for the LSTM architecture. Show how this derivative can take on different values across time steps.

Another, somewhat simpler, alternative is the **gated recurrent unit** GRU (Cho et al. 2014), depicted to the right in Fig. 4.4 and parameterized as

$$\begin{aligned}
\mathbf{z}_t &= \sigma(W_z[\mathbf{h}_{t-1}; \mathbf{e}(y_{t-1})]) \\
\mathbf{r}_t &= \sigma(W_r[\mathbf{h}_{t-1}; \mathbf{e}(y_{t-1})]) \\
\tilde{\mathbf{h}}_t &= \tanh(W_{\tilde{h}}[\mathbf{r}_t \odot \mathbf{h}_{t-1}; \mathbf{e}(y_{t-1})]) \\
\mathbf{h}_t &= (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t
\end{aligned}$$

where we note similar benefits.

Chapter 5 Part-of-speech tagging

In this chapter, we will analyze the problem of part-of-speech tagging. Part-of-speech tagging is an instance of the more general problem of **Sequence Labeling**, where the goal is assigning discrete labels to every element of a sequence. We will introduce efficient algorithmic methods that can be used to model this class of tasks, demonstrating specifically how they can be used for part-of-speech tagging.

5.1 Part-of-speech tagging

For this task, we consider as our labels the set of possible part-of-speech tags.

Definition 5.1

*A **part of speech** (POS) is a category of words, where each word in this category possesses similar syntactic properties, i.e., they play similar roles within the grammatical structure of sentences.* ♣

Common examples of POS tags include NOUN and VERB, however, there is not one unique set of possible POS tags. Different linguistic classifications can employ tagsets that differ in number or complexity. The reason why POS tagging is not a straightforward problem is that at least for a sizeable subset of languages, many words are syntactically ambiguous, making it impossible to obtain good results by simply assigning the most likely POS tag to each word in a sentence. For example, by doing so, we would not be able to correctly classify both *"I duck"* and *"A duck"*, since the word *duck* can be used both as a noun and as a verb. To obtain more accurate results, our model should be able to assign a POS tag to a word by taking context into account.

POS tagging can be seen as a shortest-path-search problem on a specific kind of graph known as trellis.

Definition 5.2

*A **trellis** is a directed acyclic graph in which nodes are divided into vertical slices, where each slice is associated with e.g., a time step or sequence position. In a trellis, each node corresponds to a distinct state in a slice, and each arrow represents a transition to some new state at the next slice.* ♣

As an example, let's consider the sequence of words *The girl drinks water* together with the POS tagset $\mathcal{T} = \{D, N, V\}$, where the tags stand for DETERMINER, NOUN and VERB, respectively. The corresponding trellis for this sentence and the tagset $\mathcal{T}$ is shown in Fig. 5.1.

The trellis in Fig. 5.1 has a row for each tag in $\mathcal{T}$ and a vertical slice for every word of the sequence. In addition, there are also an initial and final vertical slice for the special BOS (beginning of sequence, or BOS) tag and for the EOS (end of sequence, or EOS) tag, respectively. Each node in the trellis, except for EOS, is connected to every node in the next vertical slice on its immediate right. On the trellis, a valid POS tagging of the input sentence corresponds to a path starting from the BOSnode and ending at the EOSnode. The grammatically correct tagging, $\langle D, N, V, N \rangle$, is highlighted in green in Fig. 5.1, while a valid but grammatically incorrect tagging is highlighted in red. If we assign a score to each path from BOSto EOS, finding the best POS tagging for an input sequence of words becomes equivalent to finding the highest-scoring path on the corresponding trellis—a problem which can also be formulated as finding the shortest path in the graph.

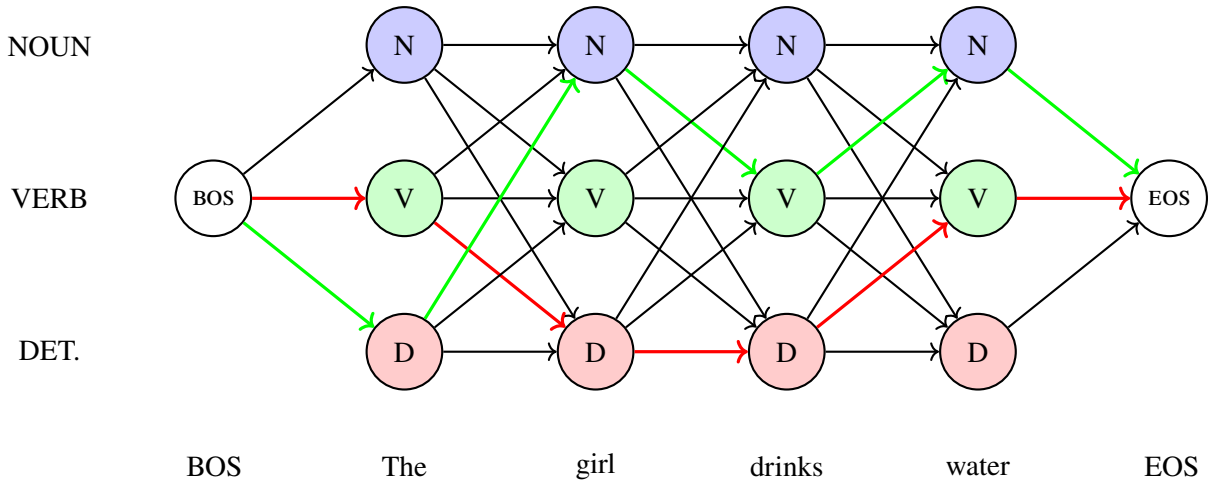


Figure 5.1: POS trellis for the sentence *The girl drinks water*

5.2 Conditional Random Fields

One way to approach the part-of-speech tagging problem is by turning it into a structured classification problem. Formally, we take a sequence of words $\mathbf{w} = \langle w_1, \ldots, w_N \rangle$ as our input. A tagging $\mathbf{t} \in \mathcal{T}^N$ is then a (structured) label. As with any other classification task, our goal is to assign the correct class to our input, which can be done using the probabilistic modeling paradigm. However, since our set of possible labels is exponentially-sized, we employ **structured prediction** techniques to decompose the problem so that we can efficiently solve it.

We first consider our standard definition of a log-linear model for estimating the probability of a certain tag sequence $\mathbf{t}$ given an input $\mathbf{w}$:

$$p(\mathbf{t} \mid \mathbf{w}) = \frac{\exp(\text{score}(\mathbf{t}, \mathbf{w}))}{\sum_{\mathbf{t}' \in \mathcal{T}^N} (\exp(\text{score}(\mathbf{t}', \mathbf{w}))} \tag{5.1}$$

where $\text{score} : V^N \times \mathcal{T}^N \rightarrow \mathbb{R}$ is a scoring function that takes as input N words from vocabulary V along with a valid tagging sequence and outputs a score for how well this tag sequence fits our input. Ultimately, we are interested in finding the sequence $\mathbf{t}^*$ that maximizes the scoring function $\text{score}(\mathbf{w}, \mathbf{t})$:

$$\mathbf{t}^* \stackrel{\text{def}}{=} \underset{\mathbf{t} \in \mathcal{T}^N}{\text{argmax}}\, \text{score}(\mathbf{w}, \mathbf{t}) \tag{5.2}$$

As with prior problems, we want $\text{score}(\mathbf{t}, \mathbf{w})$ to serve as a measure for how well the tag $\mathbf{t}$ describes the input $\mathbf{w}$. For example, using our example in Fig. 5.1, we would hope that:

$$\text{score}(\langle D, N, V, N \rangle, \mathbf{w} = \textit{the girl drinks water}) > \text{score}(\langle V, D, D, V \rangle, \mathbf{w} = \textit{the girl drinks water}) \tag{5.3}$$

Note that score can be any function of our choosing, e.g., a dot product of a weight vector and a feature function or even a neural network. In this chapter, we consider the class of score functions that adhere to specific structural assumptions, in which case our model in Equation (5.1) is formally known as a conditional random field.

Definition 5.3

*A **conditional random field** is an extension of the logistic regression classifier, i.e., it is a conditional probabilistic model for structured prediction.* ♣

Equation (5.1) looks a lot like our log-linear modeling equations from earlier in course. However now, the

summand in the denominator contains an exponential number of items. Naively computing the normalizer Z would take $\mathcal{O}(\mathcal{T}^N)$ time. Even by considering only sequences of length 20 with a set of tags of cardinality 10 we would have to score 10^{20} individual sequences, which is computationally intractable. We can overcome this issue by making a structural assumption in our problem. Assuming combinatorial structure in our problem allows us to use combinatorial algorithms for computing Z, which can lead to great gains in efficiency. For example, let's consider a scoring function that is additively decomposable over tag bigrams:

$$\text{score}(\mathbf{t}, \mathbf{w}) = \sum_{n=1}^{N} \text{score}(\langle t_{n-1}, t_n \rangle, \mathbf{w}) \tag{5.4}$$

where t_0 is the BOStag and t_{N+1} is the EOStag. In our graph representation of the POS tagging problem, this assumption implies that we go from having a score for each *path* from BOSto EOS, to having a score on each *edge* of the trellis, as in Fig. 5.1. Using the score decomposition assumption, Equation (5.1) can be rewritten as follows:

$$p(\mathbf{t} \mid \mathbf{w}) = \frac{\exp(\sum_{n=1}^{N} \text{score}(\langle t_{n-1}, t_n \rangle, \mathbf{w}))}{\sum_{\mathbf{t}' \in \mathcal{T}^N} (\exp(\sum_{n=1}^{N} \text{score}(\langle t'_{n-1}, t'_n \rangle, \mathbf{w}))} \tag{5.5}$$

The summation in the denominator is still present in Equation (5.5), but now we can simplify it to a sum of a linear number of terms by making use of the distributive property of product over addition. In the following series of equations, we will ignore the EOStag at the end of the tagging sequence to simplify the notation. We start by using the properties of the exponential function

$$Z = \sum_{\mathbf{t}_{1:N} \in \mathcal{T}^N} \exp \left\{ \sum_{n=1}^{N} \text{score}(\langle t_{n-1}, t_n \rangle, \mathbf{w}) \right\} \tag{5.6}$$

$$= \sum_{\mathbf{t}_{1:N} \in \mathcal{T}^N} \prod_{n=1}^{N} \exp \{\text{score}(\langle t_{n-1}, t_n \rangle, \mathbf{w})\} \tag{5.7}$$

$$= \sum_{\mathbf{t}_{1:N-1} \in \mathcal{T}^{N-1}} \sum_{t_N \in \mathcal{T}} \prod_{n=1}^{N} \exp \{\text{score}(\langle t_{n-1}, t_n \rangle, \mathbf{w})\} \tag{5.8}$$

where the last line takes advantage of the fact that the last tag of the sequence does not depend on its predecessors. We can now apply the distributive property of the product over the sum and isolate the last summation over t_N.

$$Z = \sum_{\mathbf{t}_{1:N-1} \in \mathcal{T}^{N-1}} \prod_{n=1}^{N-1} \exp \{\text{score}(\langle t_{n-1}, t_n \rangle, \mathbf{w})\} \times \sum_{t_N \in \mathcal{T}} \exp \{\text{score}(\langle t_{N-1}, t_N \rangle, \mathbf{w})\} \tag{5.9}$$

$$= \sum_{t_1 \in \mathcal{T}} \exp \{\text{score}(\langle t_0, t_1 \rangle, \mathbf{w})\} \times \Bigg(\sum_{t_2 \in \mathcal{T}} \exp \{\text{score}(\langle t_1, t_2 \rangle, \mathbf{w})\} \times \dots$$

$$\cdots \times \Bigg(\sum_{t_N \in \mathcal{T}} \exp \{\text{score}(\langle t_{N-1}, t_N \rangle, \mathbf{w})\} \Bigg) \Bigg) \tag{5.10}$$

where after having applied the distributive property for every tag of the sequence, we obtain the product over a linear number of sums, shown in Equation (5.10). The above computation can also be seen as a dynamic program, the same category of algorithms that backpropagation falls into. There are a number of dynamic programs that can be used with CRFs (and similarly structured models) for inference and decoding.

5.2.1 The Backward Algorithm

The algorithm for computing the normalizing constant in the denominator of Equation (5.5) is known as the Backward algorithm and is outlined below.

Algorithm 5.1

def *BACKWARD ALGORITHM*$(\mathbf{w}, \mathcal{T}, N)$*:*
  for $t_{N-1} \in \mathcal{T}$*:*
    $\beta(\mathbf{w}, t_{N-1}, N-1) \leftarrow \exp(\text{score}(\langle t_{N-1}, \textit{EOS}\rangle, \mathbf{w}))$ ▷ *Handles the scores of the incoming edges of node* EOS
  end for
  for $n \in N-2, \ldots, 1$*:* ▷ *Handles the scores over normal edges*
    for $t_n \in \mathcal{T}$*:*
      $\beta(\mathbf{w}, t_n, n) \leftarrow \sum_{t_{n+1} \in \mathcal{T}} \exp(\text{score}(\langle t_n, t_{n+1}\rangle, \mathbf{w})) \times \beta(\mathbf{w}, t_{n+1}, n+1)$
    end for
  end for
  $\beta(\mathbf{w}, \textit{BOS}, 0) \leftarrow 0$
  for $t_1 \in \mathcal{T}$*:* ▷ *Handles the scores over the outgoing edges of the* BOS *node*
    $\beta(\mathbf{w}, \textit{BOS}, 0) \leftarrow \beta(\mathbf{w}, \textit{BOS}, 0) + \exp(\text{score}(\langle \textit{BOS}, t_1\rangle, \mathbf{w})) \times \beta(\mathbf{w}, t_1, 1)$
  end for
  return $\beta(\mathbf{w}, \textit{BOS}, 0)$

The $\beta(\mathbf{w}, t_n, n)$ variables are known as **backward variables** and contain the sum of the scores of all paths starting from EOSand ending at tag t_n in position n. Therefore, $\beta(\mathbf{w}, \text{BOS}, 0)$ is equal to the normalization constant Z. In the case of score function that decomposes over bigrams, the algorithm involves computing $|\mathcal{T}| \times N$ backward variables, and for each of them, we compute the sum over $|\mathcal{T}|$ successor tags. As a result, the overall time complexity of the algorithm is $O(N|\mathcal{T}|^2)$, which is no longer exponential but only linear in the length of the input sequence N. Further, the space complexity in this setting is $O(N|\mathcal{T}|)$ since we have to keep $N \times |\mathcal{T}|$ backward variables in memory. It is worth noting that is also possible to compute Z with a forward recurrence starting from BOS instead of the backward recurrence shown in Algorithm 5.1, obtaining the same results with the same time complexity. In this case, we will work with tag pairs $\langle t_n, t_{n-1}\rangle$, starting from$\langle t_1, \text{BOS}\rangle$, and going forward towards EOS. The resulting algorithm is known as Forward Algorithm, or Sum-product Algorithm, and its associated auxiliary variables are called Forward Variables.

5.2.2 The Viterbi Algorithm

Let's assume that for now, we are not interested in calculating the exact probability $p(\mathbf{t} \mid \mathbf{w})$; rather we just want to know which sequence of tags $\mathbf{t}$ has the highest score over the sentence $\mathbf{w}$. In this case, it is not necessary to compute the normalization constant Z since $p(\mathbf{t} \mid \mathbf{w}) = \dfrac{\exp(\text{score}(\mathbf{t}, \mathbf{w}))}{Z} \propto \exp(\text{score}(\mathbf{t}, \mathbf{w}))$. Therefore, to find the highest-scoring POS tagging, we just need to solve the following problem:

$$\mathbf{t}^* = \underset{\mathbf{t} \in \mathcal{T}^N}{\text{argmax}} \exp(\text{score}(\mathbf{t}, \mathbf{w})) = \underset{\mathbf{t} \in \mathcal{T}^N}{\text{argmax}} \prod_{n=1}^{N} \exp\{\text{score}(\langle t_{n-1}, t_n\rangle, \mathbf{w})\} \tag{5.11}$$

The Viterbi algorithm computes the score of the best sequence $\mathbf{t}^*$ while also recovering the tag sequence itself using a series of backpointers b, where $b(t_n, n)$ points to the t_{n+1} tag in $\mathbf{t}^*$.

Algorithm 5.2

def *VITERBI ALGORITHM*$(\mathbf{w}, \mathcal{T}, N)$***:***
  for $t_{N-1} \in \mathcal{T}$***:***
    $v(\mathbf{w}, t_{N-1}, N-1) \leftarrow \exp(\text{score}(\langle t_{N-1}, \textit{EOS}\rangle, \mathbf{w}))$
  end for
  for $n \in N-2, \ldots, 1$***:***
    for $t_n \in \mathcal{T}$***:***
      $v(\mathbf{w}, t_n, n) \leftarrow \max_{t_{n+1}\in\mathcal{T}} \exp(\text{score}(\langle t_n, t_{n+1}\rangle, \mathbf{w})) \times v(\mathbf{w}, t_{n+1}, n+1)$
      $b(t_n, n) \leftarrow \text{argmax}_{t_{n+1}\in\mathcal{T}} \exp(\text{score}(\langle t_n, t_{n+1}\rangle, \mathbf{w})) \times v(\mathbf{w}, t_{n+1}, n+1)$ ▷ *Keeps track of the best tags*
    end for
  end for
  $v(\mathbf{w}, \textit{BOS}, 0) \leftarrow \max_{t_1\in\mathcal{T}}(v(\mathbf{w}, \textit{BOS}, 0), \exp(\text{score}(\langle \textit{BOS}, t_1\rangle, \mathbf{w})) \times v(\mathbf{w}, t_1, 1))$
  $b(\textit{BOS}, 0) \leftarrow \text{argmax}_{t_1\in\mathcal{T}}(v(\mathbf{w}, \textit{BOS}, 0), \exp(\text{score}(\langle \textit{BOS}, t_1\rangle, \mathbf{w})) \times v(\mathbf{w}, t_1, 1))$
  for $n \in 1, \ldots, N$***:***
    $t_n \leftarrow b(t_{n-1}, n-1)$ ▷ *Recovers the best tagging sequence using backpointers*
  end for
  return $\mathbf{t}_{1:N}$, $v(\mathbf{w}, \textit{BOS}, 0)$

The auxiliary variables $v(\mathbf{w}, t_n, n)$ are called **Viterbi variables** and contain the score of the best tag sequence starting from the last character of the sequence and ending with tag t_n in position n. Computing the best-scoring sequence from the backpointers is done in the last for loop in $O(N)$ time, so the overall time complexity of the algorithm remains $O(N|\mathcal{T}|^2)$. Considering triples of tags instead of pairs, the time complexity of Viterbi becomes $O(N|\mathcal{T}|^3)$. This is because, for each transition, we have to extend the trellis to contain pairs of tags instead of individual tags. For example, the transition $(N, V) \rightarrow (V, N)$ will be equivalent to the tagging sequence N, V, N. However, a transition $(t_1, t_2) \rightarrow (t_3, t_4)$ will be valid only if $t_2 = t_3$. Given this constraint, there are three degrees of freedom to choose our tags from, therefore, the overall time complexity is $O(N|\mathcal{T}|^3)$. This result can be extended in the general case for k-order dependencies, giving the complexity of $O(N|\mathcal{T}|^{k+1})$

As with the Backward algorithm, the Viterbi algorithm can be equivalently executed by traversing the input starting from the first tag and traversing to the last. In this case, the backpointer $b(t_n, n)$ will point backwards in the sequence to the optimal tag t_{n-1} of $\mathbf{t}^*$.

5.2.3 Scoring Functions

So far we have kept the scoring function $\text{score}(\mathbf{t}, \mathbf{w})$ as generic as possible. When assuming that the scoring function is additively decomposable, a minimal example of score could include word-tag pairs (known as **emission features**) and tag-tag pairs (or **transition features**). With this scoring function, the score of the sentence "*We study NLP*" with its correct POS tagging can be calculated as follows:

$$
\begin{aligned}
&\text{score}(\mathbf{w} = \textit{We study NLP}, \mathbf{t} = \langle \mathbf{N}, \mathbf{V}, \mathbf{N} \rangle) = \\
&= \sum_{n=1}^{N+1} \text{score}(\langle t_{n-1}, t_n, \rangle, \mathbf{w}) \\
&= \text{score}(\langle \textsc{bos}, N, \rangle, \mathbf{w}) \\
&\quad + \text{score}(\langle N, V \rangle, \mathbf{w}) \\
&\quad + \text{score}(\langle V, N \rangle, \mathbf{w}) \\
&\quad + \text{score}(\langle N, \textsc{eos} \rangle, \mathbf{w}) \\
&= (w_1 = \text{We}, y_1 = N) + (y_1 = N, y_0 = \textsc{bos}) \\
&\quad + (w_2 = \text{study}, y_2 = V) + (y_2 = V, y_1 = N) \\
&\quad + (w_3 = \text{NLP}, y_3 = N) + (y_3 = N, y_2 = V) \\
&\quad + (y_4 = \textsc{eos}, y_3 = N)
\end{aligned}
$$

While transition and emission features on their own can already achieve high accuracy on English part-of-speech tagging, it is also possible to create more complex architectures by modeling the score function with neural networks, i.e., $\text{score}(\langle t_{n-1}, t_n, \rangle, \mathbf{w}) = \text{NN}_\theta(\langle t_{n-1}, t_n, \rangle, \mathbf{w})$.

5.2.4 Parameter Estimation

We now focus on estimating the parameters of a CRF. We assume that we have at our disposal a dataset $\mathcal{D}$ consisting of K i.i.d. data points $(\mathbf{t}^{(i)}, \mathbf{w}^{(i)})$. Plugging in our definition in Equation (5.1), we can estimate the parameters of our CRF model by maximizing the log-likelihood of $\mathcal{D}$, which is defined as:

$$
\log \prod_{i=1}^{K} p(\mathbf{t}^{(i)} \mid \mathbf{w}^{(i)}) = \sum_{i=1}^{K} \log p(\mathbf{t}^{(i)} \mid \mathbf{w}^{(i)}) \tag{5.12}
$$

$$
= \sum_{i=1}^{K} \log \left(\frac{\exp(\text{score}(\mathbf{t}^{(i)}, \mathbf{w}^{(i)}))}{\sum_{\mathbf{t}' \in \mathcal{T}^N} (\exp(\text{score}(\mathbf{t}', \mathbf{w}^{(i)}))} \right) \tag{5.13}
$$

$$
= \sum_{i=1}^{I} \left(\text{score}\left(\mathbf{t}^{(i)}, \mathbf{w}^{(i)}\right) - \log \sum_{\mathbf{t}' \in \mathcal{T}^N} \exp \text{score}\left(\mathbf{t}', \mathbf{w}^{(i)}\right) \right) \tag{5.14}
$$

Under structural assumptions, the gradient of Equation (5.14) with respect to the parameters of score can be computed using backpropagation in linear time since, thanks to the Generalized Viterbi algorithm, the forward pass can be computed linearly in N.

5.2.5 Relationship to the Structured Perceptron

As suggested by Equation (5.1), a conditional random field is really just a softmax over an exponentially large number of elements. Accordingly, we can use a temperature parameter in its computation. Under this paradigm Equation (5.14) becomes

$$
\sum_{i=1}^{I} \left(\text{score}\left(\mathbf{t}^{(i)}, \mathbf{w}^{(i)}\right) - T \log \sum_{\mathbf{t}' \in \mathcal{T}^N} \exp \frac{\text{score}\left(\mathbf{t}', \mathbf{w}^{(i)}\right)}{T} \right) \tag{5.15}
$$

which in the limit $T \to 0$ turns to

$$\sum_{i=1}^{I} \left(\text{score}\left(\mathbf{t}^{(i)}, \mathbf{w}^{(i)}\right) - \max_{\mathbf{t}' \in \mathcal{T}^N} \text{score}\left(\mathbf{t}', \mathbf{w}^{(i)}\right) \right) \tag{5.16}$$

When the minibatch size and the learning rate are both equal to 1, Equation (5.16) is known as the structured perceptron update rule. It references the structured perceptron, a modification of the standard perceptron architecture that can produce predictions over a structured output space (Collins 2002). Equation (5.15) shows that the structured perceptron is nothing more than an annealed conditional random field with $T \to 0$.

5.3 Generalized Viterbi algorithm

So far we have used the Viterbi algorithm and Backward algorithm independently to find the highest scoring POS tagging and to calculate the normalization constant Z, respectively. However, a closer look at the two algorithms reveals that they are very similar to each other. Indeed, they are two particular instances of a more generic algorithm. By defining this algorithm using a notation known as semiring algebra, we can represent both the Viterbi and Backward algorithms (among others) simply by changing our semiring.

Definition 5.4 (Semiring)

*A **semiring** is an algebraic structure defined as a 5-tuple $S = (A, \oplus, \otimes, \bar{0}, \bar{1})$ with the following properties:*

1. *$(A, \oplus, \bar{0})$ is a commutative monoid.*
2. *$(A, \otimes, \bar{1})$ is a monoid.*
3. *$\otimes$ distributes over $\oplus$: for all $a, b, c \in A$,*

$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$$
$$c \otimes (a \oplus b) = (c \otimes a) \oplus (c \otimes b)$$

4. *$\bar{0}$ is an annihilator for $\otimes$: for all $a \in A, \bar{0} \otimes a = a \otimes \bar{0} = \bar{0}$.*

♣

Every semiring is defined on a space over which we can use the two generic operators $\oplus$ and $\otimes$. The key property of each semiring is that $\otimes$ always distributes over $\oplus$ on the set A, allowing us to build a generic recurrence that—with different semirings—can be used for different applications. As an example, we show below Algorithm 5.1 rewritten in semiring notation.

$$\bigoplus_{\mathbf{t}_{1:N}\in\mathcal{T}^N}\bigotimes_{n=1}^{N}\exp\{\text{score}(\langle t_{n-1},t_n\rangle,\mathbf{w})\} \tag{5.17}$$

$$=\bigoplus_{\mathbf{t}_{1:N-1}\in\mathcal{T}^{N-1}}\bigoplus_{\mathbf{t}_N\in\mathcal{T}}\bigotimes_{n=1}^{N}\exp\{\text{score}(\langle t_{n-1},t_n\rangle,\mathbf{w})\}$$

$$=\bigoplus_{\mathbf{t}_{1:N-1}\in\mathcal{T}^{N-1}}\bigotimes_{n=1}^{N-1}\exp\{\text{score}(\langle t_{n-1},t_n\rangle,\mathbf{w})\}\otimes\bigoplus_{\mathbf{t}_N\in\mathcal{T}}\exp\{\text{score}(\langle t_{N-1},t_N\rangle,\mathbf{w})\}$$

$$=\bigoplus_{\mathbf{t}_1\in\mathcal{T}}\exp\{\text{score}(\langle t_0,t_1\rangle,\mathbf{w})\}\otimes\bigoplus_{\mathbf{t}_2\in\mathcal{T}}\exp\{\text{score}(\langle t_1,t_2\rangle,\mathbf{w})\}\otimes \tag{5.18}$$

$$\cdots\otimes\bigoplus_{\mathbf{t}_N\in\mathcal{T}^N}\exp\{\text{score}(\langle t_{N-1},t_N\rangle,\mathbf{w})\}$$

In Algorithm 5.1, we used the $\langle\mathbb{R}^+\cup\{+\infty\},+,\times,0,1\rangle$ (a.k.a. inside) semiring, while in Algorithm 5.2, we used the $\langle[0,1],\max,\times,0,1\rangle$ (a.k.a. the Viterbi) semiring. These two semirings can be employed when the individual scores returned by the scoring function represent probabilities. In practical scenarios, however, multiplying probabilities over long sequences can lead to numerical underflow. To avoid this issue, we can work in log-space. That is, we can use the Viterbi algorithm with log-probability scores using the $\langle\mathbb{R}^-\cup\{-\infty\},\max,+,-\infty,0\rangle$ semiring and we can directly compute the log normalizer $\log Z$ instead of Z using the $\langle\mathbb{R}\cup\{-\infty\},\log(e^a+e^b),a+b,-\infty,0\rangle$ semiring over log-probabilities. Table 5.1 (Huang 2008a) shows some examples of semirings together with their practical application in the POS tagging problem.

Semiring	Set	$\oplus$	$\otimes$	$\overline{0}$	$\overline{1}$	intuition/application
Boolean	$\{0,1\}$	$\vee$	$\wedge$	0	1	logical deduction, recognition
Viterbi	$[0,1]$	max	$\times$	0	1	prob. of the best derivation
Inside	$\mathbb{R}^+\cup\{+\infty\}$	$+$	$\times$	0	1	prob. of a string
Real	$\mathbb{R}\cup\{+\infty\}$	min	$+$	$+\infty$	0	shortest-distance
Tropical	$\mathbb{R}^+\cup\{+\infty\}$	min	$+$	$+\infty$	0	with non-negative weights
Counting	$\mathbb{N}$	$+$	$\times$	0	1	number of paths

Table 5.1: Examples of semirings

Semiring parametrization can also be applied to any shortest-path algorithm that works on a weighted directed graph. This means that it is possible to transform the Dijkstra's, Bellman-Ford, and Floyd–Warshall algorithms into a more general form using semirings. For instance, it is possible to calculate the highest-scoring POS tagging or the normalization constant Z on a trellis using Dijkstra's algorithm with a specific semiring instead of using the Viterbi or the Backward algorithm.

Chapter 6 Context-Free Parsing

In this chapter we will focus on context-free parsing, which is an instance of syntactic parsing. In the next chapter we will discuss dependency parsing as another instance of syntactic parsing. The goal of syntactic parsing is to assign a syntactic structure, i.e. a parse tree, to a sentence. Parse trees have a variety of uses, such as grammar checking, i.e., assessing whether a sentence can be generated by applying the rules of a grammar.

6.1 Syntactic Constituency

There is overwhelming evidence that human language is structured hierarchically. One way to model this hierarchical nature is by breaking down sentences into **constituents** and expressing their relationship.

Definition 6.1

*A **constituent** is a contiguous sequence of words that functions as coherent unit.* ♣

Example 7 *Consider the following sentences, where one example of a constituent is bracketed:*

(1) *Eleanor ate [**the pad see ew**]*

(2) *John loves [**the red car**]*

(3) *Papa eats caviar [**with a spoon**]*

While it might seem intuitive that these sequences of words can act as a single unit, i.e. form constituents, is there a formal way of confirming this intuition? Yes, as there exist some (language-specific) tests for constituency. These tests include:

- **Substitution**: a constituent can be substituted by other phrases of the same type. Applying this rule to the constituent in (1) results in: Eleanor ate [**it**].
- **Clefting**: a constituent can be moved from its normal position to the beginning of a cleft sentence, which takes the form: *It is/was* X *that...*, where X is the sequence of words we are testing for constituency. Applying this rule to the constituent in (2) results in: It is [**the red car**] that John loves.
- **Answer Ellipsis**: a constituent can appear alone as the answer to a single wh-question. Applying this rule to the constituent in (3) results in: How does papa eat caviar? [**With a spoon**]

6.2 Context-Free Grammars

The most widely used formal system for modeling constituency structures is the **context-free grammar** (CFG). A **grammar** defines an ordered set of **rules** or **productions**, by which one can form strings out of a language vocabulary. Such rules show how symbols of a language can be grouped and ordered together. The symbols that are used in CFGs are divided into two groups. The symbols that correspond to words in the language are called terminal symbols. The symbols that express abstractions over terminals are called non-terminals. The main differentiating factor of CFGs is that rules can be applied *regardless* of context. More formally, a CFG is defined as follows:

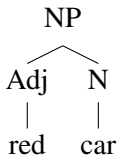


Figure 6.1: Phrase-structure tree for "red car"

Definition 6.2

*A **context-free grammar** G is a quadruple $\langle \mathcal{N}, S, \Sigma, \mathcal{R} \rangle$ consisting of:*

- *A finite set of non-terminal symbols $\mathcal{N}$; written in upper-case letters, e.g. N_1, N_2, N_3*
- *A distinguished start non-terminal S*
- *An alphabet of terminal symbols Σ; written as lower-case letters, e.g. a_1, a_2, a_3*
- *A set of production rules $\mathcal{R}$ of the form $N \to \alpha$, where $N \in \mathcal{N}$ and $\alpha \in (\mathcal{N} \cup \Sigma)^*$ (Kleene closure of $\mathcal{N} \cup \Sigma$)*

♣

A CFG gives us the set of rules to generate strings under the grammar. This sequence of rule expansions is called **derivation** of the string words.

Example 8 *The following rules express that a noun phrase (NP) is created either from a noun (N) or from an adjective (Adj) followed by a noun. "car" and "bus" are nouns, while "red" and "fast" are adjectives.*

$$\begin{aligned} \text{NP} &\to \text{N} \mid \text{Adj N} \\ \text{N} &\to \textit{car} \mid \textit{bus} \\ \text{Adj} &\to \textit{red} \mid \textit{fast} \end{aligned}$$

In Example 8, we see that the string *"red car"* can be derived from the non-terminal NP. It is also common to show the derivation with a parse tree, which in this case is called a **phrase-structure tree**. Phrase-structure trees (Chomsky 1956) represent both the derivation of the string under the grammar and the syntactic structure of the string, where each node represents a constituent. The phrase-structure tree for *"red car"* is shown in Fig. 6.1.

It can be useful to define normal forms of grammars, where production rules are of a particular structure. **Chomsky Normal Form** (CNF) is a one such form for CFGs.

Definition 6.3

*A grammar is in **Chomsky Normal Form (CNF)** if the right-hand side of every production rule includes either two non-terminals or a single terminal symbol:*

$$N_1 \to N_2\, N_3 \tag{6.1}$$

$$N \to a \tag{6.2}$$

♣

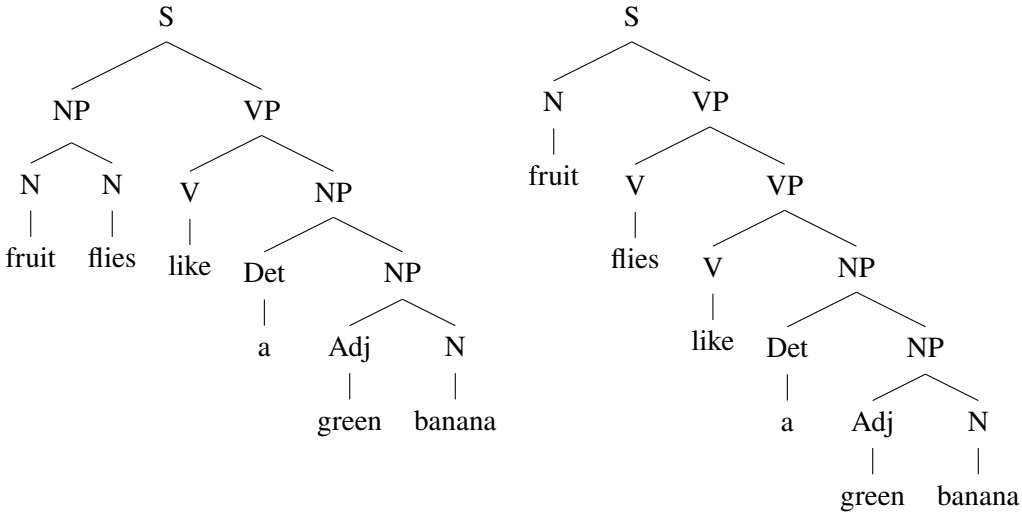
We call the set of rules in Equation (6.1) non-terminal productions and those in Equation (6.2) terminal productions. Chomsky normal form grammars are binary branching, i.e. parse trees under such grammar are binary trees. We will make use of this property later when implementing parsing algorithms.

✍ **Exercise 6.1** Two grammars are **weakly equivalent** if they generate the same set of strings. This is different from **strong equivalence**, where the grammars generate the same set of strings via applying the same rules. Show that all CFGs can be converted into a grammar in CNF that is weakly equivalent.

6.2.1 Ambiguity

In natural language, it is often the case that a single sentence can be interpreted in more than one way. Structural ambiguity occurs when the grammar can assign more than one phrase-structure tree to a sentence, i.e. the sentences can be decomposed into constituents in different ways. Consider the following sentence, which has two valid phrase-structure trees:

Fruit flies like a green banana.



Here we see that the sequence "fruit flies" can be interpreted as a constituent on its own, or as two separate constituents. The choice subsequently affects which structures are valid for parsing the rest of the sentence. This is one example of ambiguity in syntax. We formally define specific types of ambiguity below:

- **Attachment ambiguity**: when prepositions can attach to either verbs or direct objects:

  (4) I [shot [an elephant][in my pajamas]] vs. I [shot [an elephant [in my pajamas]]]

- **Complement structure**: when the complement can attach to either main verbs or the indirect objects:

  (5) Students [complained [to [the professor]][that [they didn't understand]]] vs. Students [complained [to [the [professor [that [they didn't understand]]]]]]

  In the first case, students do not understand and complain about this to the professor. In the second case, the professor is the person whom students do not understand.

- **Modifier scope**: when an adjective can either modify the first noun in a noun-noun compound or the entire compound:

  (6) [[plastic cup] holder] vs. [plastic [cup holder]]

6.2.2 Probabilistic Context-Free Grammars

In case of ambiguity, a string can be generated via applying more than one series of rules. In such cases, how do we know which derivation is preferable, or in other words, which derivation is more "likely?" One way to go about this is to model derivations probabilistically by assigning probabilities to each rule of the context-free grammar. A series of production rules, and consequently a phrase-structure tree, can then be assigned a joint probability. This modification creates a **probabilistic context-free grammar** (PCFG).

Definition 6.4

*Similar to a CFG, a **probabilistic CFG** is defined as a quintuple $\langle \mathcal{N}, S, \Sigma, \mathcal{R}, \mathcal{P} \rangle$ consisting of:*

- *A finite set of non-terminal symbols $\mathcal{N}$*
- *A distinguished start non-terminal S*
- *An alphabet of terminal symbols Σ*
- *A set of production rules $\mathcal{R}$ of the form $N \to \alpha$*
- *A set of probabilities $\mathcal{P}$ assigned to each production rule*

♣

Because these grammars are "context-free," production rule probabilities are independent of each other, Consequently, we can define the probability of a parse tree as simply the multiplication of rules that are used to create the tree:

$$p(\mathbf{t}) = \prod_{r \in \mathbf{t}} p(r) \tag{6.3}$$

These probabilities must be locally normalized, meaning that the distribution over each transition should be a valid probability distribution. Specifically, consider a series of rules $\mathrm{N} \to \alpha_1, \ldots, \mathrm{N} \to \alpha_K$ that are expansions of the node N. We should have:

$$\sum_{k=1}^{K} p(\mathrm{N} \to \alpha_k) = 1 \tag{6.4}$$

6.2.3 Weighted Context-Free Grammars

PCFGs are a special case of a more general category of **weighted context-free grammars** (WCFGs). In a WCFG, a generic non-negative weight is assigned to each production rule. These weights can be, for example, of the form of the exponent of a general scoring function. Note that WCFGs are globally normalized, meaning the scores of trees should be normalized by the sum of weights of all the possible trees under the grammar in order to form a valid probability distribution. More formally, we define a probability distribution over a WCFG as:

$$p(\mathbf{t}) = \frac{1}{Z} \prod_{r \in \mathbf{t}} \exp\{\mathrm{score}(r)\} \tag{6.5}$$

where the normalizing constant is:

$$Z = \sum_{\mathbf{t}' \in \mathcal{T}} \prod_{r' \in \mathbf{t}'} \exp\{\mathrm{score}(r')\} \tag{6.6}$$

6.3 Parsing

Syntactic parsing is the task of assigning a syntactic structure to a sentence. This chapter focuses on constituency parsing—and particularly on *probabilistic* constituency parsing—which WCFGs provide a set of rules for. Note that we will use the terms parsing and constituency parsing interchangeably here. In the next chapters, we will discuss alternative parsing structures.

✍ **Exercise 6.2** Consider the following minimal CFG with only one non-terminal X and the following rules:

$$\mathrm{X} \to \mathrm{X}\ \mathrm{X}$$
$$\mathrm{X} \to a, \quad a \in \Sigma$$

how many valid parse trees are there for a specific input string with n words under this grammar, i.e. what is the number of derivations?

The goal of probabilistic parsing is to determine the highest probability parse tree $\mathbf{t}$ given a sentence $\mathbf{s}$. In this course, we formulate the probability distribution over parses as a general log linear-model:

$$p(\mathbf{t} \mid \mathbf{s}) = \frac{1}{Z(\mathbf{s})} \prod_{r \in \mathbf{t}} \exp\{\text{score}(r)\} \tag{6.7}$$

Note that our definition makes use of the fact that—at least when computing the score of a tree—each tree can be seen as simply a bag, i.e. multi-set, of production rules. While the above equation looks quite a bit like Equation (6.5), it differs in that our probability is now conditioned on a specific string $\mathbf{s}$. Consequently, we only need to perform computations over $\mathcal{T}(\mathbf{s})$, the set of trees in the grammar that **yield** $\mathbf{s}$. This includes computing the normalizing constant: $Z(\mathbf{s}) = \sum_{\mathbf{t}' \in \mathcal{T}(\mathbf{s})} \prod_{r' \in \mathbf{t}'} \exp\{\text{score}(r')\}$.

Definition 6.5

*The **yield** of a parse tree is the (ordered) leaves of its prediction rules, i.e., the resulting string.* ♣

To train our log-linear model using the standard maximum-likelihood objective, we must compute $Z(\mathbf{s})$. This can be done efficiently by taking advantage of the structural assumptions made by CFGs in general, specifically that we can factor the score along the productions in the grammar. Assuming our grammar is in CNF, we can rewrite Equation (6.7) as:

$$p(\mathbf{t} \mid \mathbf{s}) = \frac{1}{Z(\mathbf{s})} \underbrace{\prod_{X \to Y\ Z \in \mathbf{t}} \exp\{\text{score}(X \to Y\ Z)\}}_{\text{non-terminal productions}} \cdot \underbrace{\prod_{X \to x \in \mathbf{t}} \exp\{\text{score}(X \to x)\}}_{\text{terminal productions}} \tag{6.8}$$

and our normalizer as:

$$Z(\mathbf{s}) = \sum_{\mathbf{t}' \in \mathcal{T}(\mathbf{s})} \underbrace{\prod_{X \to Y\ Z \in \mathbf{t}'} \exp\{\text{score}(X \to Y\ Z)\}}_{\text{non-terminal productions}} \cdot \underbrace{\prod_{X \to x \in \mathbf{t}'} \exp\{\text{score}(X \to x)\}}_{\text{terminal productions}} \tag{6.9}$$

6.3.1 CKY Algorithm

Given a grammar in CNF, the Cocke-Kasami-Younger (CKY) algorithm provides an efficient dynamic program for parsing strings. The algorithm was originally derived in the 1960s to solve the context-free **recognition problem**.

Definition 6.6

*Given an input string, the **recognition problem** is the problem of determining whether that string is accepted by a specific context-free grammar.* ♣

However, the application of CKY is not limited to the recognition problem; it can be used in a variety of tasks such as computing the normalizing constant of a probabilistic parser, finding the one best (or k-best) parse(s) under a probabilistic parser, computing the entropy of a probabilistic parse forest, and much more.

The CKY algorithm takes a bottom-up approach to parsing. The algorithm first forms small constituents, and then tries to merge them into larger constituents. Pseudocode for how CKY can be used to compute the normalization constant for $p(\cdot \mid \mathbf{s})$ is shown in Algorithm 6.1.

Algorithm 6.1

def *WeightedCKY*$(\mathbf{s}, \langle \mathcal{N}, S, \Sigma, \mathcal{R} \rangle, \text{score})$*:*

$N \leftarrow |\mathbf{s}|$

$chart \leftarrow 0$
for $n = 1, \ldots, N$*:*
  for $X \rightarrow \mathbf{s}_n \in \mathcal{R}$*:*
    $\text{chart}[n, n+1, X] \mathrel{+}= \exp\{\text{score}(X \rightarrow \mathbf{s}_n)\}$ ▷ *Handles single word tokens*
  end for
end for
for $span = 2, \ldots, N$*:*
  for $i = 1, \ldots, N - span + 1$*:* ▷ i *marks the beginning of the span*
    $k \leftarrow i + span$ ▷ k *marks the end of the span*
    for $j = i + 1, \ldots, k - 1$*:* ▷ j *marks the breaking point of the span*
      for $X \rightarrow Y\ Z \in \mathcal{R}$*:*
        $\text{chart}[i, k, X] \mathrel{+}= \exp\{\text{score}(X \rightarrow Y\ Z)\} \times \text{chart}[i, j, Y] \times \text{chart}[j, k, Z]$
      end for
    end for
  end for
end for
return $\text{chart}[1, N + 1, S]$

The main steps of the above algorithm can be summarized as follows:

- The algorithm maintains a chart to store the intermediate normalization factors (see Example 9).
- The first loop fills the diagonal of the chart by looking only at the leaves of the tree. That is, $\text{chart}[n, n+1]$ is filled with weights of terminal productions that yield the word in nth position in the input sequence.
- In the next loop, the algorithm fills the rest of the chart by incrementally increasing the length of the contiguous span of leaves that it considers. Specifically, when the span length is t, $\text{chart}[n, n+t]$ is filled with the normalization factor for the subtree at $[n, n+t]$, if it exists. This is done by combining the normalization factors across all valid ways of breaking the span positioned at $[n, n+t]$ into two valid constituents, i.e., over all non-terminal production rules X $\rightarrow$ Y Z such that Y can yield $[n, j]$ and Z can yield $[j+1, n+t]$. It then multiplies the normalization factors of non-terminals yielding each sub-span, which are already stored in the chart from former iterations.
- When there exists a valid parse of a string under the CFG, the position $[1, N+1]$ of the chart corresponding to the start non-terminal contains the normalization constant for the distribution over parses of the input string.

Example 9 *We use Algorithm 6.1 to parse the sentence "Fruit flies like a green banana" under the grammar Table 6.1. The completed CKY chart is shown in Fig. 6.2. There are two possible ways to parse this sentence, which can be seen in the completed chart: parse starting from S $\rightarrow$ N VP (shown in orange) and the other from S $\rightarrow$ NP VP (shown in blue).*

Non-Terminal Productions	Terminal Productions
S $\rightarrow$ NP VP	N $\rightarrow$ *fruit*
S $\rightarrow$ N VP	Adv $\rightarrow$ *like*
NP $\rightarrow$ Det N	V $\rightarrow$ *flies*
VP $\rightarrow$ V AdvP	N $\rightarrow$ *flies*
NP $\rightarrow$ Det NP	V $\rightarrow$ *like*
VP $\rightarrow$ V NP	Det $\rightarrow$ *a*
NP $\rightarrow$ N N	Adj $\rightarrow$ *green*
NP $\rightarrow$ Adj N	N $\rightarrow$ *banana*
AdvP $\rightarrow$ Adv NP	

Table 6.1: Grammar in Chomsky Normal Form (CNF)

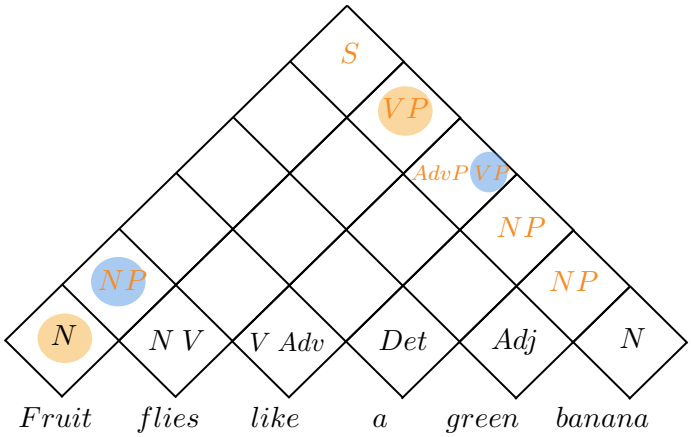


Figure 6.2: Complete CKY chart

When estimating the weights of our WCFG, we can use the normalization constant computed in Algorithm 6.1 to compute the log-likelihood of a ground-truth parse tree given a specific input sentence. We can then optimize for the joint log-likelihood of a dataset under our WCFG using standard gradient-descent methods. Note that as shown in the first chapter, the backpropagation algorithm can be used to compute the gradient with respect to the normalization constant with the same order that CKY computes the normalization constant itself.

Runtime Analysis. The runtime complexity of CKY is $O(N^3|\mathcal{R}|)$, where N is the length of the input sequence and $|\mathcal{R}|$ is the size of the rule set. The space complexity is $O(N^2|\mathcal{N}|)$, where $|\mathcal{N}|$ is the size of the set of non-terminals. The algorithm needs to maintain a datastructure of this magnitude to store intermediate results.

Generalized CKY. Similarly to the Viterbi algorithm discussed in the previous chapter, CKY can be formulated using semiring algebras (Goodman 1999). This formulation allows us to solve a number of parsing problems simply by employing a specific semiring. For example, one could use the boolean semiring to solve the recognition problem. The generalized version of the algorithm is shown in Algorithm 6.2.

Algorithm 6.2

def *SemiringCKY*($\mathbf{s}$, $\langle \mathcal{N}, S, \Sigma, \mathcal{R} \rangle$, score)*:*
  $N \leftarrow |\mathbf{s}|$
  chart $\leftarrow \mathbf{0}$
  for $n = 1, \ldots, N$*:*
    for $X \rightarrow \mathbf{s}_n \in \mathcal{R}$*:*
      chart$[n, n+1, X] \oplus=$ *exp*$\{\text{score}(X \rightarrow \mathbf{s}_n)\}$ ▷ *Handles the single word tokens*
    end for
  end for
  for *span* $= 2, \ldots, N$*:*
    for $i = 1, \ldots, N - span + 1$*:* ▷ i *marks the beginning of the span*
      $k \leftarrow i + span$ ▷ k *marks the end of the span*
      for $j = i + 1, \ldots, k - 1$*:* ▷ j *marks the breaking point of the span*
        for $X \rightarrow Y\ Z \in \mathcal{R}$*:*
          chart$[i, k, X] \oplus=$ *exp*$\{\text{score}(X \rightarrow Y\ Z)\} \otimes$ *chart*$[i, j, Y] \otimes$ *chart*$[j, k, Z]$
        end for
      end for
    end for
  end for
  return *chart*$[1, N+1, S]$

Chapter 7 Dependency Parsing

Last chapter looked at constituency parsing, where the goal was to identify the nested structure of sentences based on constituents using context-free grammars. We saw that such parses are often ambiguous, in part due to the question of **attachment**, i.e., which word in a constituent (or, in a production) should form its **head**—the word within the phrase which is grammatically most important (a more rigorous definition of the term follows shortly). Such relationships between words of a sentence can be modeled using a more lightweight structure which does not rely on full phrasal constituents and rules linking them but rather on the binary grammatical relationships between words. These dependency structures lend themselves perhaps more naturally to languages, as they can deal concisely with languages of varying typologies and even approximate a sort of semantic relationships between predicates and their arguments. This is the main idea behind **dependency grammars**.

7.1 Dependency Grammars

The linguistic notion of a **dependency relation** defines binary relations between the words in a sentence, e.g., how one word modifies another. First, let us formally define **head words.**[1]

Definition 7.1

Phrase p has a head h if the presence of h determines the range of syntactic functions that p can bear. For example, the constructions into which the phrase "on the table" can enter are determined by the presence of the preposition, on. Therefore the preposition is its head and, by that token, it is a prepositional phrase. ♣

Dependency parsing makes such relations explicit by linking a head word with its immediate **modifiers** or **dependents**. The latter modify and provide additional information on the head. The relations also provide additional information on the exact grammatical roles with a predefined set of grammatical types, and thus give form to the typed dependency structure of a sentence. Constituency and dependency parsing thus provide two different, albeit not incompatible, views on syntactic structure of sentences. Lately, a lot of effort has been put into the latter, in large part due to the free availability of labeled treebanks such as Universal Dependencies, which includes more than 60 languages[2].

7.1.1 Dependency Trees

We typically represent the dependency parse of a sentence as a tree. Formally, we can define a **dependency tree** (also called a dependency graph) for a sentence containing N tokens as a tree with N nodes representing the words (and optionally, punctuation) of the original sentence and one special node, the **root** node. Nodes are connected with labeled, directed edges, which connect the nodes of head words to their dependents. The root node, which does not represent any word in the sentence, must satisfy the constraint of having exactly one child node: the node corresponding to the root word. This encodes that each sentence has exactly one root word. We refer to this as the *single-root constraint*. Note that this constraint implies that each node in the dependency graph (apart from the root node) has exactly one parent. This means that the graph is a **directed spanning tree**, i.e., there is an undirected path between any pair of nodes. An example of a dependency tree is shown in Fig. 7.1.

[1] Taken from https://www.thoughtco.com/head-words-tern-1690922.

[2] https://universaldependencies.org/

There are two important classes of dependency trees: *projective* and *non-projective*. They are defined based on the type of edges they contain.

Definition 7.2

*We say an edge from node i to node j is **projective** iff $\forall k$ s.t. $i < k < j$, node k is a descendant of i. A projective tree is a dependency tree whose edges are all projective, i.e., no edges will cross each other. A tree is **non-projective** if it does not meet this criterion.* ♣

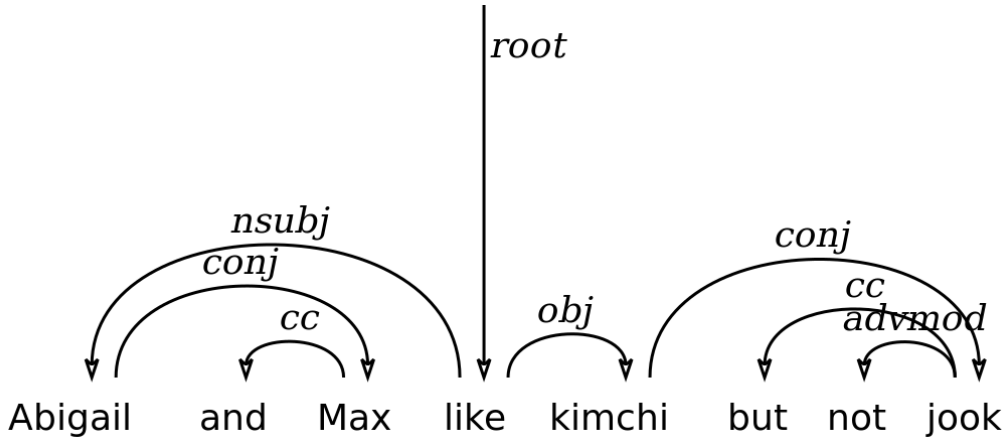


Figure 7.1: An example of a projective dependency tree. Adapted from Figure 11.2 from Eisenstein 2019.

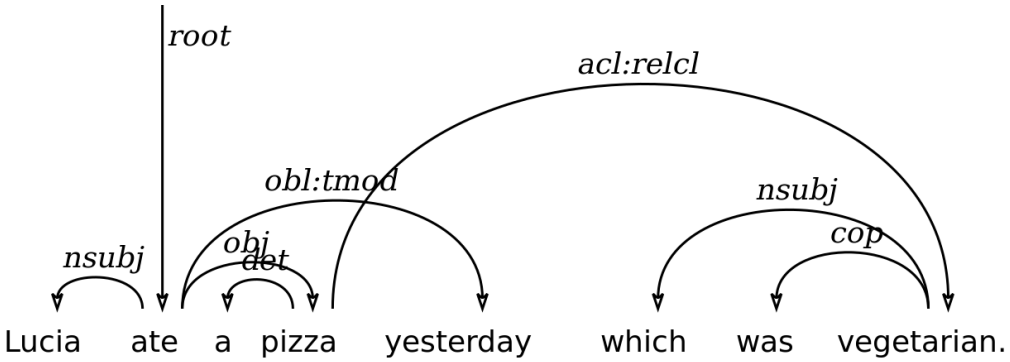


Figure 7.2: An example of a non-projective dependency tree. Adapted from Figure 11.5 from Eisenstein 2019.

Although the difference might seem negligible at first, it carries with it important algorithmic considerations. Since the edges of projective dependency trees do not span multiple constituents by definition, they can in fact be represented as an extended version of constituency parsing which includes head words in the grammar. Thus, the algorithms we use for parsing projective trees largely resemble the CKY algorithm. Non-projective trees, however, require an entirely different set of algorithms, which we discuss below.

7.2 Modeling Probability Distributions over Non-projective Trees

We now address the problem of finding the dependency parse for a given sentence. We focus on approximating exact best-parse inference approaches, which fall under the framework of **graph-based dependency parsing**. Without loss of generality, we ignore edge labels, i.e., we will focus on finding the best unlabeled dependency tree to make things cleaner and hopefully more instructive.

As usual, we will model the problem of finding the best parse of a sentence using a probabilistic approach. We assign probabilities to all possible parses $\mathbf{t}$ of a sentence $\mathbf{w}$. The compatibility of the parse $\mathbf{t}$ with the sentence $\mathbf{w}$ will be specified by a scoring function $\text{score}(\mathbf{t}, \mathbf{w})$. More precisely, we define:

$$p(\mathbf{t} \mid \mathbf{w}) = \frac{1}{Z} \exp\{\text{score}(\mathbf{t}, \mathbf{w})\}, \tag{7.1}$$

where

$$Z = \sum_{\mathbf{t}' \in \mathcal{T}(\mathbf{w})} \exp\left\{\text{score}\left(\mathbf{t}', \mathbf{w}\right)\right\} \tag{7.2}$$

and $\mathcal{T}(\mathbf{w})$ denotes the set of all admissible parses of sentence $\mathbf{w}$.

7.2.1 An Intractable Problem

As we can see, normalization (and later, maximization) requires computation over the whole set of possible parses. This is problematic as the number of undirected spanning trees in a complete graph on N nodes is[3] N^{N-2}. Even narrowing this set down to only spanning trees that satisfy the single root constraint, our set of possible parses is still $(N-1)^{N-2}$. Thus, exact inference using complete enumeration is inefficient, if not infeasible. Therefore we must make some (perhaps unrealistic) structural assumptions to make solving the problem tractable.

The assumption we make is that the scoring function, which quantifies how appropriate a parse tree is for the given sentence, *decomposes over the edges of the tree*. We call functions satisfying this constraint **edge-factored**.

Definition 7.3

*An **edge-factored scoring function** is of the form:*

$$\text{score}(\mathbf{t}, \mathbf{w}) = \sum_{(i \to j) \in \mathbf{t}} \text{score}(i \to j, \mathbf{w}) + \text{score}(r, \mathbf{w}), \tag{7.3}$$

where r is the root word according to the tree $\mathbf{t}$. Note that the score *function here has been overloaded to take edges and nodes.*

We will see that this assumption allows us to construct exact efficient algorithms to find best-scoring parses and train probabilistic models, while still allowing us to take into account many useful features that determine good parses.

7.2.2 Finding the Normalization Constant with the Matrix Tree Theorem (MTT)

Contrary to previous structured prediction problems, summation and maximization will unfortunately not be instances of a particular semiring. They will require two separate algorithms. Luckily, the edge-factored assumption will be enough to allow for efficient computation of both.

To find the normalizing constant Z, we will make use of an old theorem: the **Matrix Tree Theorem**. In short, this theorem tells us we can count the number of undirected spanning trees in an undirected graph on N nodes by simply taking the determinant of a specific $N \times N$ matrix, a computation that takes $\mathcal{O}\left(N^3\right)$ time. In that sense, it is a generalization of Cayley's formula mentioned above. Tutte's generalization of this algorithm

[3]This is known as Cayley's formula, see: https://en.wikipedia.org/wiki/Cayley%27s_formula

also applies to the case of *directed* graphs and thus, allows us to find the normalization constant Z in the same runtime.

We now define two quantities needed for this algorithm. First, let A be the **adjacency matrix** of the dependency graph, i.e., a matrix with entries $\mathbf{A_{ij}} = \exp\{\text{score}\,(i \to j, \mathbf{w})\}$ and let ρ be the vector of root scores of all the words, i.e., $\rho_i = \text{score}\,(i, \mathbf{w})$. Note that ρ_i denotes how suitable word i is to be the root word of the parse tree, i.e., the sole child of the root node. The algorithm works by defining the **graph Laplacian**, a matrix of the form:

Definition 7.4

*The **graph Laplacian** is defined as*

$$L_{ij} = \begin{cases} -A_{ij}, & \text{if } i \neq j \\ \sum_{k \neq i} A_{kj} & \text{else.} \end{cases} \tag{7.4}$$

♣

In words, this matrix contains the negative edge weights on the off-diagonal entries and the **in-degrees** (the sum of the weights of all the *incoming* edges of a node, which is the same as the *degree* in undirected graphs) of the nodes on the diagonal. Then, the normalization constant can be calculated as:

$$Z = |L| = \det(L)\,. \tag{7.5}$$

This equivalence is a result of the **Matrix Tree Theorem** (Chaiken and Kleitman 1978).

While the above computation is almost what we need, it does no account for the single-root constraint. Consequently, our normalization constant in Equation (7.5) includes inadmissible trees in its summation. A straightforward approach to address this problem would be to manually combine N sums, each time hard-coding one word as the root. However, this would raise the time complexity to $\mathcal{O}\left(N \cdot N^3\right) = \mathcal{O}\left(N^4\right)$. Luckily, Koo et al. 2007 generalize the result from the previous theorem to this exact setting while keeping the same time complexity. They achieve this through a light modification to the original Laplacian:

Definition 7.5

*The **modified graph Laplacian** is defined as*

$$L_{ij} = \begin{cases} \rho_j, & \text{if } i = 1 \\ -A_{ij}, & \text{if } i \neq j \\ \sum_{k \neq i} A_{kj} & \text{else.} \end{cases} \tag{7.6}$$

♣

This matrix differs from our original Laplacian only in that the first row is replaced with the root scores. We do not provide a proof of correctness here as the result is rather unintuitive; we direct the interested reader to Koo et al. 2007 for some useful ideas and illustrations.

The above result allows us to calculate the normalizing constant of any edge-factored scoring of our dependency graph. This in turn allows us to learn optimal scores via maximum likelihood estimation, as we have often done in the course. Formally, suppose that the score_θ function is some differentiable function of the parameters θ. Equation (7.1) formulates how we encode the probability of a parse given the score this function assigns. Given a labeled dataset $\mathcal{D}$ of size N, we can optimize the parameters $\boldsymbol{\theta}$ of our score function to maximize the (joint) log-likelihood of the examples in our dataset, where the log-likelihood is of the form:

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \log p\left(\mathbf{t}^{(i)} \mid \mathbf{w}^{(i)}\right) = \sum_{i=1}^{N} \text{score}\left(\mathbf{t}^{(i)}, \mathbf{w}^{(i)}\right) - \log Z\left(\mathbf{w}^{(i)}\right). \tag{7.7}$$

As usual, this optimization can be done using gradient-descent techniques. We can easily attain the gradient of

the above equation with respect to $\boldsymbol{\theta}$ using backpropagation techniques, since all components of our objective are differentiable.

7.3 Decoding: Finding the Best Parse of a Sentence

As previously mentioned, summation and maximization in a non-projective dependency graph cannot be formulated as two instances of the same problem. Consequently, we must look at a new algorithm—the Chu-Liu-Edmonds (CLE) algorithm—when searching for the optimal dependency parse.

7.3.1 Problem Definition

We now suppose that we are given some edge-factored scoring function over dependency trees. We want to find the directed spanning tree satisfying the single-root with the maximum score (equivalently, the most probable parse, given the formulation in Equation (7.1)). As a first step towards this goal, we first ignore the single-root constraint and develop the machinery for this simplified case. Later, we modify our approach to take this constraint into account.

7.3.2 A First Attempt: Kruskal's Algorithm

Given the graphical nature of our problem, a natural first approach to finding the optimal tree in a dependency graph would employ Kruskal's algorithm, one of the elementary graph algorithms taught in undergraduate algorithms classes. Kruskal's algorithm efficiently finds the highest scoring spanning tree in an *undirected* graph: Given an undirected graph with arbitrarily weighted edges, it starts forming the spanning tree from an empty graph and sequentially connects nodes such that no cycles are created. At each step, the edge with the highest weight in the original graph is chosen, under the condition that it does not create a cycle. Using some special data structures, it can be made efficient and runs in time $\mathcal{O}(E \log E)$, where E is the number of edges in the graph.[4]

Kruskal's algorithm breaks down on directed graphs, where the greedy decisions do not lead to globally optimal solutions anymore, as shown in the following example.

Example Consider the graph shown in Fig. 7.3. It is easy to see that the maximal spanning tree contains the edges $r \to 3$ and $3 \to 2$. However, the algorithm would greedily choose the edge $2 \to 3$ at first, thus preventing the choice of the other optimal edges. The fact that the edges are directed makes Kruskal's algorithm suboptimal.

7.3.3 The Chu-Liu-Edmonds (CLE) Algorithm

We instead turn to the Chu-Liu-Edmonds (CLE) algorithm to solve our problem. To motivate this algorithm, we start with a simple approach: since we know that in a spanning tree, each node has to have exactly 1 incoming edge, and we want the highest-scoring one, we just pick, for every node in the graph, the highest scoring incoming edge. We call the resulting graph the **greedy graph**. In general, this solution may not be appropriate as it need not be a tree. However, it's easy to see that if *it were a tree*, it would be the highest scoring one, which means we are done. So rather than giving up on our greedy solution, we can instead modify it to transform it into a tree. This is done through a series of **contractions** and **expansions**, as described below.

[4]For a more complete discussion, Wikipedia is a good source https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

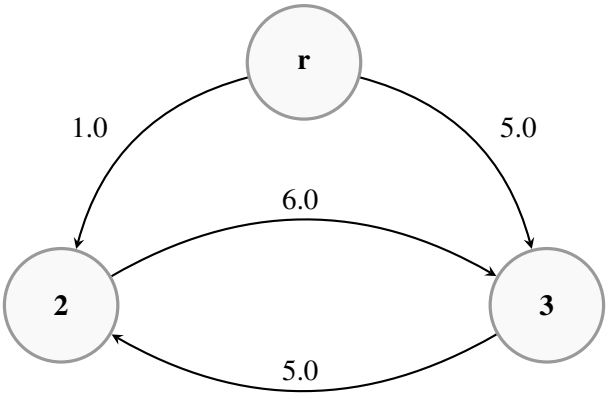


Figure 7.3: A simple directed graph.

Suppose we find that our greedy graph is not a tree. In this case, it must include at least one directed cycle, which we would like to get rid of. Let C be such a cycle. We first divide the edges of the graphs into several sets:

Definition 7.6

For a specific cycle C, we define the following types of edges:

- ***Exit edges** are the edges emanating from C.*
- ***Enter edges** are the edges pointing into C.*
- ***Dead edges** are the edges inside C. Besides the ones which are part of C, this also includes the edges which have both ends in C.*
- ***External edges** are the edges outside C which do not have any vertices in it.*

♣

If we want to get rid of the cycle, we will inevitably have to remove one of the dead edges. Which one should we remove? Of course, we have to remove one such that the set of nodes will still be reachable from outside, meaning we can still connect to the nodes in this cycle via an enter edge. This implies that *only* dead edges whose end point are also the end point of an enter edge are eligible for deletion. Note that for any given cycle there can only be one dead edge ending in the same node as an enter edge. This injective relationship has an important implication: we can frame the breaking of a cycle as *choosing* an enter edge, where the dead edge that enters the same node as the enter edge is the one we choose to remove.

Preferably we choose the enter edge that, together with its own weight, will result in the highest scoring set of remaining edges in the former cycle. Therefore, we *reweigh* the enter edges to reflect this. More precisely, for each enter edge, we set its new weight to be its original weight plus the sum of the weights of all remaining edges in the cycle. Intuitively, this new weight corresponds to the quality of the resulting subtree if we were to choose that particular enter edge: An enter edge will have a high score if it (a) has a very high weight originally, or (b) removes a relatively low-scoring edge in the cycle.

This reweighting allows us to *contract* the (now broken) cycle into a new *supernode*, which we treat as just a normal node from this point on. We can do this because our enter edge uniquely identifies the set of edges in the cycle that we keep for our final tree, thus we can abstract away this information for now. We now have a smaller graph with fewer cycles; any remaining cycles can be removed by recursively repeat the above procedure on the contracted graph until our graph is a tree. This will then be the optimal tree.

In order to recover this spanning tree from our contracted graph, we *expand* all the cycles we have contracted in the reverse direction, which is a relatively straightforward process. Any supernode we have created will have a single incoming edge, which as previously noted, uniquely identifies the edges inside the supernode—i.e., the

former cycle—that we will keep in the final spanning tree. Expanding this node entails restoring the edges in the former cycle that we did not remove. This is done until we arrive to the original graph with no more supernodes, at which point, our remaining edges form our optimal spanning tree. We now walk through an example of this full procedure.

Example Suppose we start with the graph in Fig. 7.4a. We start by constructing the greedy graph, shown in Fig. 7.4b, which contains a directed cycle on nodes $2, 3, 4$. We now focus on this cycle, which we label c. In Fig. 7.4c, the different classes of edges are also color-coded: blue edges are the enter edges, gray edges are the dead edges, red edges are the exit edges, and brown edges are external edges. In Fig. 7.4d we show how to reweigh the enter edges: the edge $\rho \rightarrow 2$ gets a weight 40 (original weight)$+60+70$ (remaining cycle edge weights) $= 170$. We obtain the new weight of edge $1 \rightarrow 3$ similarly. With that, we can contract the cycle, as shown in Fig. 7.4e.

With the graph contracted, we again construct the greedy graph, as shown in Fig. 7.4f. As our greedy graph is now a tree, it must be the optimal spanning tree and so we can stop. We now just have to expand the sole supernode C we have contracted. Since the optimal spanning tree in the contracted graph contains the edge $\rho \rightarrow C$ among the enter edges of C, we know exactly which edge from the cycle C we have to remove - $3 \rightarrow 2$. After expanding this supernode, we have the final maximum spanning tree, shown in Fig. 7.4g.

7.3.4 Satisfying the Single Root Constraint

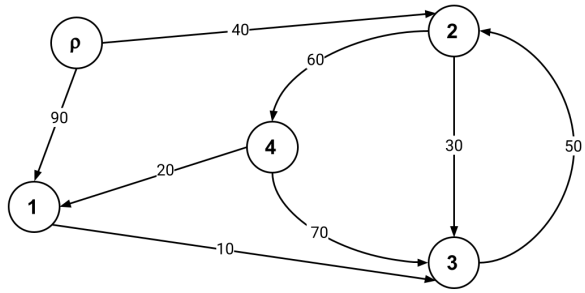
The original CLE Algorithm described above finds the optimal directed spanning tree *without* taking into account the single-root constraint. While similarly to computing our normalization constant, we could try a brute force approach that performs this optimization N times, where we fix the nth node to be the only child of the root node each time, this adds a factor of N to the computation. Again, fortunately for us, Zmigrod, Vieira, and Cotterell 2020 generalize the procedure to take into account the root constraint more efficiently. Their solution is as follows:

We find the optimal greedy tree according to the CLE algorithm. However, before expanding our contracted nodes, we must ensure there is only one edge emanating from the root node. For each outgoing edge from the root we:

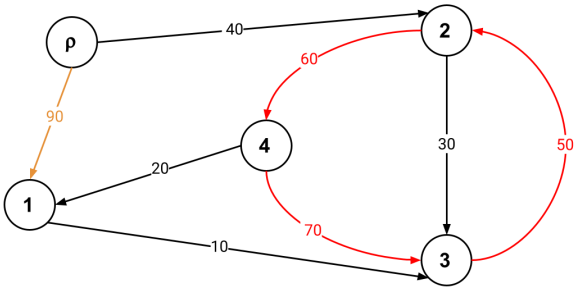
1. Look at the end node of the edge and find the next most highly weighted incoming edge for it.
2. Calculate the loss on the score of the entire parse that such a change would incur. This is the difference between the scores of the original edge and the second-best one just found.

Having calculated the losses for all of the root node's outgoing edges, we now consider removing the edge which incurs the lowest loss. If removal of this edge does not lead to a cycle in our current graph, we remove the edge. Otherwise, we contract the cycle that would be formed in the graph with this edge removed, but we keep the edge itself there. After contraction, we repeat the two steps above until we have removed all but one of the root edges, meaning we have found the best scoring suitable spanning tree. Then, we just expand the supernodes as before and we are left with the optimal tree that satisfies the root constraint. Again, this procedure is best seen through an example:

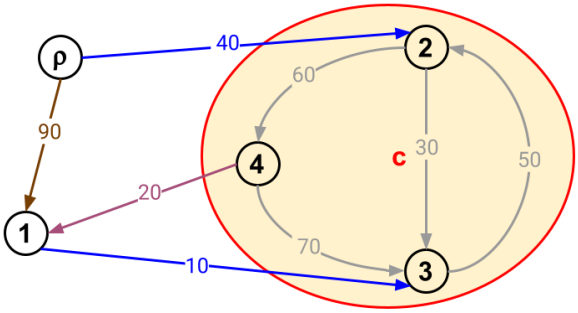
Example We start as in the previous example but once we find a greedy graph that is a tree, we stop and modify it to satisfy the single-root constraint, as shown in Fig. 7.5a. We consider both edges emanating from the root: $\rho \rightarrow C$ and $\rho \rightarrow 1$. We must find alternative incoming edges for the nodes they are pointing to, as shown in Fig. 7.5b and Fig. 7.5c. For the edge $\rho \rightarrow C$, we see that the alternative edge into C would be the edge



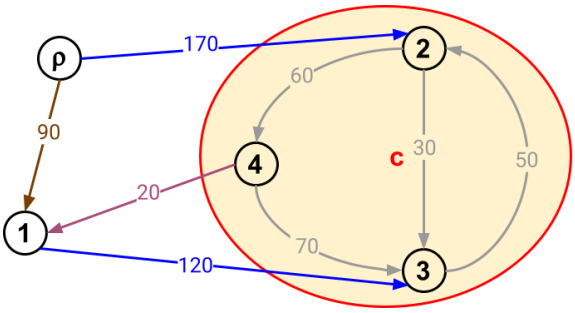
(a) Original Graph.



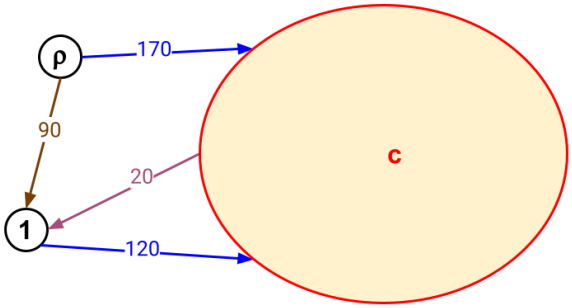
(b) Step 1



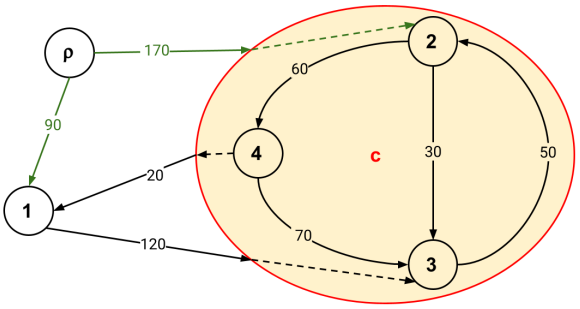
(c) Step 2



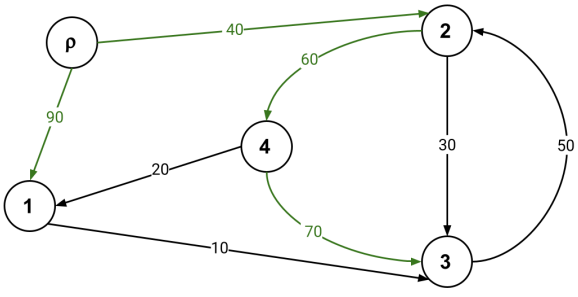
(d) Step 3



(e) Step 4

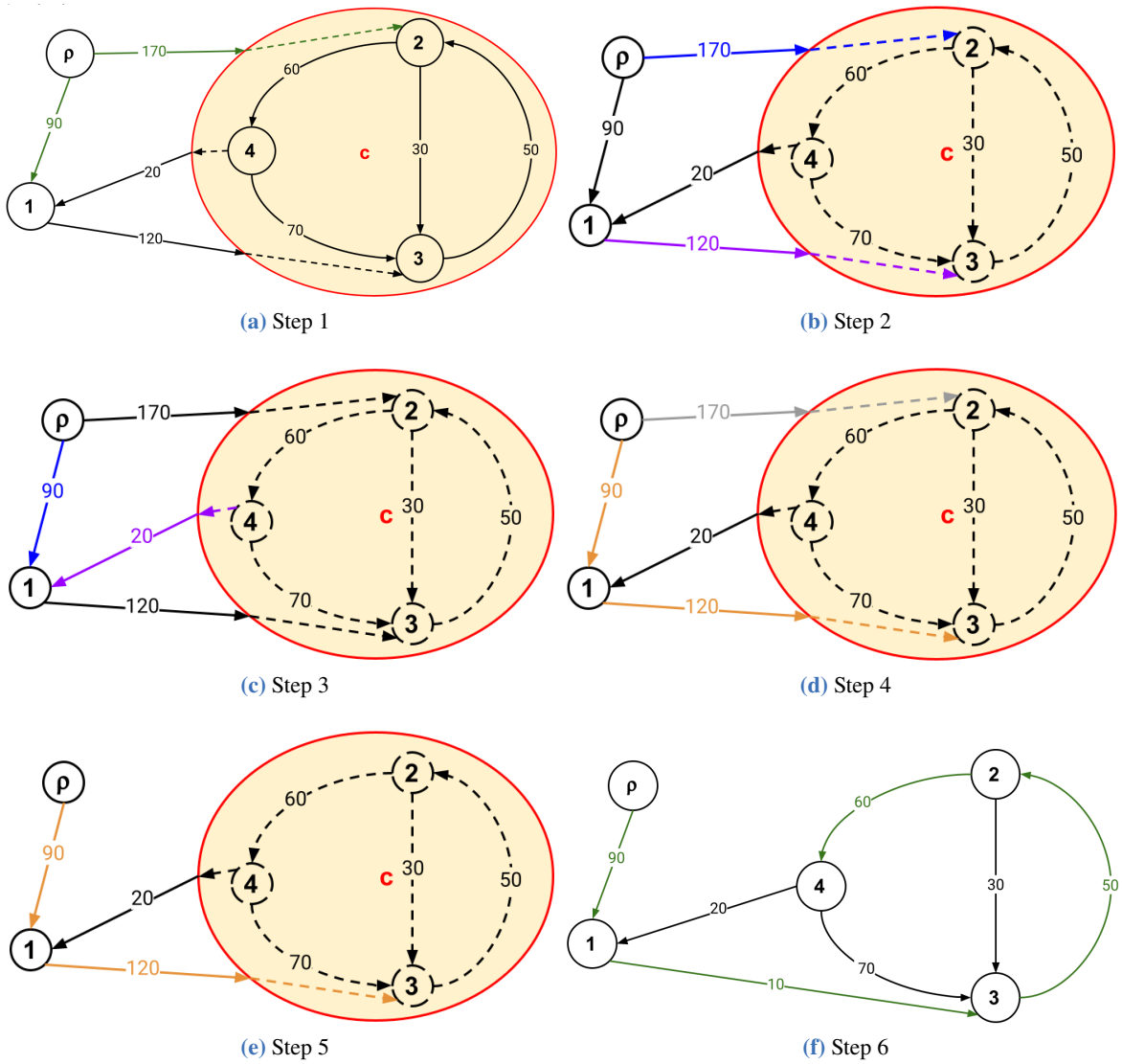


(f) Step 5



(g) Step 6

Figure 7.4: The original graph and subsequent steps of the original CLE algorithm.



(a) Step 1 (b) Step 2

(c) Step 3 (d) Step 4

(e) Step 5 (f) Step 6

Figure 7.5: The steps to ensure compliance with the single-root constraint.

$1 \rightarrow C$ with a weight of 120. Therefore, if we chose this incoming edge for C instead the original one, our loss would be $170 - 120 = 50$. Similarly, we find that for the edge $\rho \rightarrow 1$, our loss would be $90 - 20 = 70$. Thus, deleting edge $\rho \rightarrow C$ incurs a smaller loss and we consider removing it, as shown in Fig. 7.5d. Since the greedy graph with the edge removed does not contain any cycles, we can remove it straight away. The resulting graph is shown in Fig. 7.5e, and since it is a tree with a single edge coming from the root, we have found the appropriate single-root maximal spanning tree. We just expand our supernodes as before, ending up with the final spanning tree shown in Fig. 7.5f.

7.3.5 Wrapping Up

We have now presented the complete procedure for determining the highest scoring admissible parse efficiently. The pseudocode for the entire algorithm is outlined in Algorithm 7.1. As we can see, it is relatively straight-forward, but care must be taken with a few fancy tricks and efficient data structures to make it efficient. As originally proposed by Edmonds 1968 (it was co-discovered by Chu and Liu), the CLE algorithm ran in $\mathcal{O}(EN) = \mathcal{O}(N^3)$ time, but can be made more efficient to run in $\mathcal{O}(E + N \log N)$ time, which is due to

Tarjan 1977.

Algorithm 7.1

```
def MST(G):
    if CYCLE IN GREEDY (G):
        return EXPAND (CONSTRAIN (MST (CONTRACT (G, CYCLE))))
    else
        return CONSTRAIN (GREEDY (G))
    end if

def CONSTRAIN(G):
    if NUMBER OF ROOT EDGES (GREEDY (G)) > 1:
        e ← ROOT EDGE TO REMOVE (G)
        if CYCLE IN GREEDY (G − e):
            return CONSTRAIN (CONTRACT (G, CYCLE))
        else
            return CONSTRAIN (G − e)
        end if
    else
        return GREEDY (G)
    end if
```

7.4 Alternative Methods for Dependency Parsing

Throughout the chapter, we focused exclusively on *graph-based dependency parsing*. It offers exact best-parse inference with the trade-off of making (perhaps unrealistic) structural assumptions. Here, we note that this is not the only way to look for appropriate dependency parses. Another well known approach is **transition-based parsing**, which tries to alleviate three shortcomings of the graph-based methods: the inadequacy of the edge-factored assumption, the fact that it parses the *whole sentence* at the same time whereas humans incrementally build meaning in sequential order, and the relatively high computational complexity.

This approach is based on shift-reduce parsers, i.e., those often in parsing programming languages. It uses a stack on which specific substructures are sequentially added and at each step incorporated into a growing tree or saved for processing later. Appealingly, it runs in linear time in the length of the string. However, it can only produce *projective* parse trees and is thus less powerful than the graph-based approaches described above. We will not discuss transition-based parsing in more detail, but you can find more information on it in Section 11.3 of Eisenstein 2019.

Chapter 8 Semantics

8.1 Meaning in Linguistics

Many of the systems we build in NLP should understand the meaning of language rather than solely its structure. For example, if we want to build a robot that can follow natural language directions, while the task of speech recognition determines which words were said (surface), parsing words to commands would require deriving instructions from those words (core). Another important point why semantics are important is that syntax systems alone do not perfectly describe the human language. For example, a context-free grammar can produce meaningless sentences, such as: "Colorless green dreams sleep furiously." While semantic analysis is very important for many tasks, it is often difficult to perform due to the ambiguity of sentences, complex interactions between words in a sentence, as well as large input/output spaces:

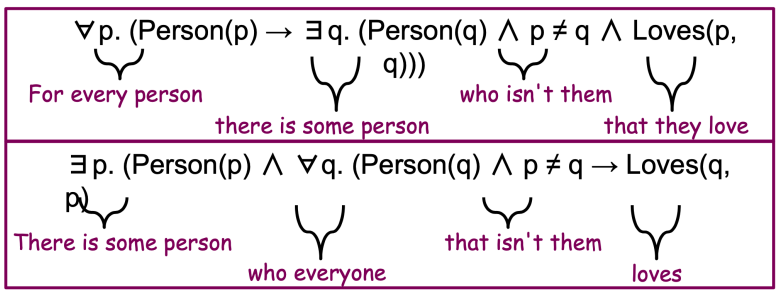
- **Ambiguity:** one sentence can have more than one meaning
  - "Everybody loves someone else."
    - $\forall p.(\text{Person}(p) \rightarrow \exists q.(\text{Person}(q) \wedge p \neq q \wedge \text{Loves}(p, q)))$
    - $\exists p.(\text{Person}(p) \wedge \forall q.(\text{Person}(q) \wedge p \neq q \rightarrow \text{Loves}(q, p)))$
- **Complex interactions:**
  - The presence of different words can change the meaning of others.
- **Large input/output spaces:**
  - The space of all strings is huge! There are many inputs to learn and many outputs (i.e., meanings) to choose from.

8.2 Logical Forms

A logical form is a representation of the meaning of a sentence. As an analogy, a parse tree is to syntax as a logical form is to semantics:

Natural language		**Logical form**
"Everyone loves someone else."	$\longrightarrow$	$\forall p.(\text{Person}(p) \rightarrow \exists q.(\text{Person}(q) \wedge p \neq q \wedge \text{Loves}(p, q)))$

Logical forms are composed of variables, predicates, quantifiers and boolean expressions. Each expression should map to a single meaning (unlike in natural language)!



8.3 The Basics of Lambda Calculus

The main goal of semantic analysis is to extract the meaning of a sentence and represent it in logical form (typically in first-order logic). The basic framework in which this can be done is **lambda calculus**. Let's consider the semantic representation of the sentence **"Alex likes Brit"** is the logical form LIKES(ALEX,BRIT). Here, ALEX and BRIT are logical constants representing Alex and Brit. LIKES is a 2-ary relation symbol such that LIKES(x, y) represents the fact that x likes y. In order to obtain the semantic representation LIKES(ALEX, BRIT) from the sentence "Alex likes Brit", we use the principle of compositionality—the principle upon which lambda calculus is based.

8.3.1 Principle of compositionality

Definition 8.1

Principle of compositionality: *The meaning of a complex expression is a function of the meanings of that expression's constituent parts.*

♣

In order to get the meaning of a sentence, we start by (syntatically) parsing the sentence into simpler and simpler constituents. We then construct the semantic representations bottom-up. If the syntatic parse of a sentence contains a production $x \to yz$, then $x.\text{sem}$ is a function of $y.\text{sem}$ and $z.\text{sem}$.

8.3.2 Applying the principle of compositionality

In order to explain the principle of compositionality better, let us walk through an example demonstrating how we can apply this principle. Let us focus again on the sentence **"Alex likes Brit"**.

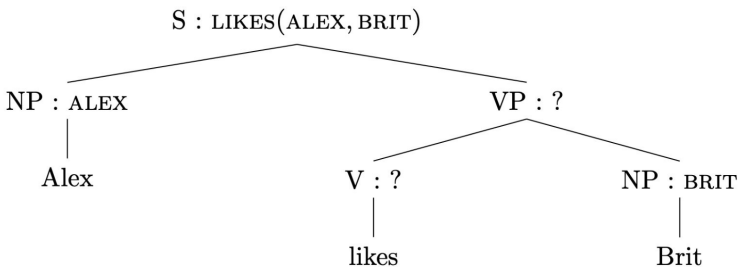


Figure 8.1: Figure taken from Eisenstein's book *Natural Language Processing*

- The semantics of the noun phrases "Alex" and "Brit" are the logical constants ALEX and BRIT, respectively.
- The semantics of the verb phrase "likes Brit" is a function of the semantics of the verb "likes" and the semantics of the noun phrase "Brit".
- The semantics of the sentence "Alex likes Brit" is a function of the semantics of the noun phrase "Alex" and the semantics of the verb phrase "likes Brit".

The semantic analysis procedure should eventually produce the logical form LIKES(ALEX,BRIT) as a representation of the meaning of the sentence "Alex likes Brit". This situation raises the following question: *what should be the semantic representations of the verb "likes" and the verb phrase "likes Brit"?*

- **Likes Brit:** The verb phrase "likes Brit" expects a noun phrase on its left, which will be the subject of the verb "likes". If the (subject) noun phrase has the semantic representation NP.sem, then the semantic

representation of the whole sentence should be LIKES(NP.sem, BRIT). We can represent the semantics of the verb phrase "likes Brit" as a function whose input is x, and whose output is LIKES(x, BRIT).

- **Likes:** the verb "likes" expects a noun phrase on its right, which will be its object. If the (object) noun phrase has the semantic representation NP.sem, then the semantic representation of the verb phrase will be a function that takes x as input, and produces LIKES(x, NP.sem) as output. We can represent the semantics of the verb "likes" as a function whose input is y, and whose output is a function that takes x as input and produces LIKES(x, y) as output.

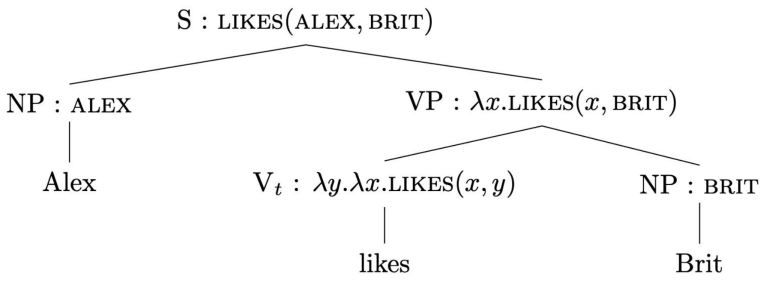


Figure 8.2: Figure taken from Eisenstein's book *Natural Language Processing*

As you can see from the example above, we need a formalism that allows us to encode and manipulate such functions in a simple and concise way. This is where the lambda operator comes into play. We can use the $\lambda x.f(x)$ to represent a function that takes x as input, and produces the expression $f(x)$ as output, where:

- x is a variable.
- $\lambda x.f(x)$ and $\lambda y.f(y)$ are equivalent and they represent the same function.
- When we apply the function $\lambda x.f(x)$ to some expression M, we replace every (free) occurrence of x in f(x) with M.

Lambda calculus is a formal system to model computation, that allows higher-order functions, i.e., functions that can take functions as input and functions that can produce functions as output.

8.3.3 Standard lambda calculus

Terms in standard lambda calculus are built recursively. The **basic terms** are variables: $x, y, z, \ldots$ New terms can then be constructed from previously constructed terms using the following two recursive rules:

- **Abstraction:**
  - If M is a term and x is a variable, then $\lambda x.M$ is a term.
  - $\lambda x.M$ can be understood a the function that takes x as input and produces M as output.
  - More precisely, if the expression N is the input, then the output is the expression $M[x := N]$, where we replace every (free) occurrence of x in M by N.
- **Application:**
  - If M and N are terms, then (MN) is a term.
  - (MN) can be understood as applying M to N.

It is important to understand the **scope** of an abstraction, which are the variables over which an abstraction operates. Essentially, the scope of λx in the expression $\lambda x.M$ is M.

- Example: $\lambda x.\lambda y.(x\ ((\lambda x.x\ x)\ y))$.
  - The scope of the first abstraction λx in $\lambda x.\lambda y.(x\ ((\lambda x.x\ x)\ y))$ is the sub-term $\lambda y.(x\ ((\lambda x.x\ x)\ y))$.
  - The scope of the abstraction λy in $\lambda x.\lambda y.(x\ ((\lambda x.x\ x)\ y))$ is the sub-term $(x\ ((\lambda x.x\ x)\ y))$.

- The scope of the second abstraction λx in $\lambda x.\lambda y.(x\ ((\lambda x.x\ x)\ y))$ is x.

8.3.4 Free and bound variables:

If a variable does not occur in the scope of any abstraction that holds its name, we say that its occurrence is **free**. Otherwise, we say that its occurrence is **bound**. If it belongs to several scopes of abstractions holding its name, we assign it to the smallest scope. We can define the notions of free and bound occurrences by induction on the complexity of the expression.

- Basic terms:
  - The (only) occurrence of the variable x in the (basic) term x is free.
- Abstractions:
  - If x occurs free in M, then the same occurrence of x in $\lambda x.M$ is bound to the abstraction λx that appears at the beginning of the expression $\lambda x.M$.
  - If an occurrence of x is bound in M, then the same occurrence of x in $\lambda x.M$ is bound to the same abstraction to which it were bound in M.
- Applications:
  - If x occurs free in M or N, then the same occurrence of x in (MN) is free.
  - If an occurrence of x is bound in M or N, then the same occurrence of x in (MN) is bound to the same abstraction to which it were bound in M or N.
- Example: $((\lambda x.\lambda y.(x\ ((\lambda x.x\ x)\ y))\ \lambda x.x)\ z)$.
  - Here, the occurrence of z is free, whereas the occurrences of the other variables are bound to the abstraction of the same color.

8.3.5 $\alpha-$conversion

Some programs are essentially the same. Consider for example, $\lambda xyz.zyx$ and $\lambda abc.cba$. Although the terms are different, they denote the same program: a program that reverses the input. We formalize this equivalence through ***α*-conversion**. In α-conversion, two programs are equivalent if one can be obtained from the other by replacing variables according to some rules.

Formally, ***α*-conversion** is the process of renaming a variable in a lambda term. We rename a variable in an abstraction, together with all the occurrences of the variable in the scope of the abstraction, which are bound to the same abstraction. Since $\alpha-$conversions are supposed to produce equivalent terms, the renaming should not cause the free occurrence of a variable to become bound, or cause a bound occurrence of a variable to become bound to a different abstraction.

According to the definition above, all of the following are valid $\alpha-$conversions:

- $\lambda x.x \to \lambda y.y$
- $\lambda x.(x\ x)\ y \to \lambda t.(t\ t)\ y$
- $\lambda x.\lambda y.(x\ ((\lambda x.x\ x)\ y)) \to \lambda z.\lambda y(z\ ((\lambda x.x\ z)\ y))$

Some examples for cases where renaming a variable does not produce a valid $\alpha-$conversion can be given as the following:

- $\lambda x.(x\ y) \to \lambda y.(y\ y)$ is not allowed because the second occurrence of y was free and became bound.
- $\lambda x.\lambda y.(x\ y) \to \lambda y.\lambda y.(y\ y)$ is not allowed because the renamed variable was bound to the first abstraction and became bound to the second abstraction.

- $\lambda x.\lambda y.(x\,y) \rightarrow \lambda x.\lambda x.(x\,x)$ is not allowed because the first occurrence of the variable x was bound to the first abstraction and became bound to the second abstraction.

8.3.6 $\beta-$reduction

Programs consume inputs and produce outputs. In the same way, lambda terms produce outputs and we compute such outputs using **β-reductions**. $\beta-$reduction is the process of applying one lambda term to another. The $\beta-$reduction of a term $(\lambda x.M\,N)$ is obtained by applying the function $\lambda x.M$ to N, i.e., replacing every free occurrence of x in M by N. We denote the obtained expression as $M[x := N]$. For instance, the following are $\beta-$reductions:

- $(\lambda x.\lambda\, y.(x\,((\lambda x.x\,x)\,y))\,z) \rightarrow \lambda y.(z\,((\lambda x.x\,z)\,y))$
- $\lambda y.(z\,((\lambda x.x\,z)\,y)) \rightarrow \lambda y.(z\,(z\,y))$

The replacement of the free occurrences of x in M should not cause the free occurrence of any variable in N to become bound in $M[x := N]$. We can resolve such issues by first applying $\alpha-$conversions to M. Some examples can be given as the following:

- $(\lambda x.\lambda y.(x\,y)(y\,z)) \rightarrow \lambda y.((y\,z)\,y)$ is not allowed because y was free in $(y\,z)$ and now it is bound in $\lambda y.((y\,z)\,y)$.
- In order to correctly apply the $\beta-$reduction to $(\lambda x.\lambda y.(x\,y)(y\,z))$, we do the following:
  - Apply an $\alpha-$conversion $(\lambda x.\lambda y.(x\,y)(y\,z)) \rightarrow (\lambda x.\lambda t.(x\,t)(y\,z))$.
  - Now apply the $\beta-$reduction $(\lambda x.\lambda t.(x\,t)(y\,z)) \rightarrow \lambda t.((y\,z)\,t)$.

Note that repeatedly applying $\beta-$reductions may or may not terminate. This is analogous to the halting problem for Turing machines. For example, let $F = \lambda x.((x\,x)\,x)$, then $(F\,F) = (\lambda x.((x\,x)\,x)\,F) \rightarrow ((F\,F)\,F) \rightarrow ((((F\,F)\,F)\,F)\,F) \rightarrow \ldots)$. This can be seen as an infinite while loop.

8.3.7 Equivalence

Definition 8.2

*Two lambda terms are **equivalent** if one can be obtained from the other after a sequence of $\alpha-$conversions and $\beta-$reductions.*

♣

For example,

- $(\lambda x.\lambda y.(x\,((\lambda x.x\,x)\,y))\,z)$ is equivalent to $\lambda y.(z\,(z\,y))$
- $(\lambda x.\lambda y.(x\,y)(y\,z))$ is equivalent to $\lambda t.((y\,z)\,t)$.

Please note that the problem of deciding whether two lambda terms are equivalent is undecidable. There is no algorithm that can solve this problem.

8.4 Enriched lambda calculus

The most basic definition of lambda calculus—as given in §8.3.3—has limitations when it comes to expressing relations between objects. For semantic analysis, we would like lambda calculus to be able to produce expressions such as: LIKES(ALEX,BRIT), which represents the semantics of the sentence "Alex likes Brit." Thus, we introduce an enriched version of lambda calculus. For the rest of this chapter, we will employ these paradigms, and will simply refer to them as lambda calculus.

The expressions in the enriched lambda calculus are built recursively from basic expressions, where basic expressions are formed from the following components:

- logical constants,
- variables,
- literals.

8.4.1 Logical constants

Logical constants represent objects and relations between objects in the real world. For example,

- Logical constants representing humans: ALEX, BRIT, . . .
- Logical constants representing cities: NEW-YORK-CITY, ZURICH, . . .
- Logical constants representing relations between humans: LIKES, TEACHER, . . .

Every relation symbol (a.k.a. predicate) has an **arity** that determines the number of objects that it relates.

- LIKES has arity 2, and LIKES(x, y) means that x likes y.
- TEACHER has arity 1, and TEACHER(x) means that x is a teacher.

8.4.2 Variables

Variables can represent undetermined objects (or undetermined relations). We typically use lower-case letters $(x, y, z, \dots)$ to denote variables that represent objects, and we use upper-case letters $(P, Q, \dots)$ to denote variables that represent relations. More generally, variables can represent general undetermined lambda terms. We typically use upper-case letters to denote variables that abstract functions.

8.4.3 Literals

Literals are formed by applying relations to objects. For example,

- LIKES(ALEX,BRIT), which means that Alex likes Brit.
- TEACHER(ALEX), which means that Alex is a teacher.

We can use variables instead of logical constants in order to construct literals, such as

- LIKES(ALEX, y), which means that Alex likes an undetermined person y.
- P(ALEX,BRIT), which means that an undetermined relation P holds between Alex and Brit.
- $P(x, y)$, which means that an undetermined relation P holds between the undetermined objects x and y.

8.4.4 Basic logical forms

Starting from literals, we can construct basic logical forms using logical connectives and quantifiers. For instance, TEACHER$(x)\wedge$LIKES(x,BRIT), which means that the undetermined object x is a teacher that likes Brit. $\exists y$. LIKES(ALEX, y)$\wedge$TEACHER(y), which means that Alex likes some teacher.

8.4.5 Lambda terms

Starting from the basic expressions (logical constants, variables, literals, and basic logical forms), we build the lambda terms by recursively applying the process of abstraction, i.e., using the lambda operator. For instance,

- λx.LIKES(x,BRIT) is a function that takes (an object) x as input and produces LIKES(x,BRIT) as output.
  $\Longrightarrow$ if ALEX is the input, then LIKES(ALEX, BRIT) is produced.

- λP.P(ALEX,BRIT) is a function that takes (a relation) P as input and produces P(ALEX,BRIT) as output.
  $\Longrightarrow$ if LIKES is the input, then LIKES(ALEX,BRIT) is produced.

$\alpha-$conversions and $\beta-$reductions can be applied exactly as in the standard lambda calculus.

8.4.6 Combinatory Logic

Combinatory logic is an alternative to lambda calculus that formalizes the concept of computation and the construction of computable functions. Unlike lambda calculus, combinatory logic does not use abstractions. Instead of using abstractions, combinatory logic constructs complex functions using a few primitive higher-order functions.

Definition 8.3

Terms in combinatory logic are defined recursively as follows:

- *The basic terms are:*
  - *Variables:* $x, y, z, \ldots$
  - *Primitive functions (or, combinators): These are higher order functions,* $I, S, K, \ldots$.
- *Terms are then recursively constructed using the rule of application:*
  - *If* M *and* N *are combinatory terms, then* (MN) *is a term that means applying* M *to* N.
- *Parantheses are left associative, e.g.* $(K\,x\,y)$ *means* $((K\,x)\,y)$ *and* $(S\,x\,y\,z)$ *means* $((S\,x)\,y\,z)$.

♣

It is often useful and convenient to introduce and use several combinators.

Definition 8.4

The $\mathbf{I}$ ***combinator*** *represents the identity function, and is defined as* $(I\,x) = x$ *for every* x.

♣

Definition 8.5

The $\mathbf{K}$ ***combinator*** *is defined as* $(\mathbf{K}\,x\,y) = ((\mathbf{K}\,x), y) = x, \forall x, y$. *The* $\mathbf{K}$ *combinator produces constant functions, i.e.,* $(\mathbf{K}\,x)$ *is the constant function that always outputs* x.

♣

Definition 8.6

The $\mathbf{S}$ ***combinator*** *is defined as* $(\mathbf{S}\,x\,y\,z) = ((x\,z\,(y\,z)) = ((x\,z)(y\,z)), \forall x, y, z$.

♣

Definition 8.7

The $\mathbf{B}$ ***combinator*** *is defined as* $(\mathbf{B}\,x\,y\,z) = ((x\,(y\,z)), \forall x, y, z$. *The* $\mathbf{B}$ *combinator can be understood as being a composition operator, i.e., if* f *and* g *are functions, then* $(\mathbf{B}\,f\,g)$ *is equivalent to* $f \circ g$.

♣

Definition 8.8

The $\mathbf{C}$ ***combinator*** *is defined as* $(\mathbf{C}\,x\,y\,z) = ((x\,z), y) \forall x, y, z$.

♣

Definition 8.9

The $\mathbf{T}$ ***combinator*** *is defined as* $(\mathbf{T}\,x\,y) = (y\,x) \forall x, y$. *The*$\mathbf{T}$ *combinator is analogous to the canonical embedding of a vector space* V *into* V^{**}, *the dual of its dual. If* f *is a function, then* $((\mathbf{T}x)\,f) = (f\,x)$.

♣

The $\mathbf{B}$ and $\mathbf{T}$ combinators are used in the combinatory categorial grammars (CCG), which we will discuss next. In CCG, the $\mathbf{B}$ combinator is called the **composition** combinator, and $\mathbf{T}$ combinators is called the **type-raising** combinator.

8.5 Combinatory Categorial Grammars (CCG)

This group of grammars are mildly context-sensitive, meaning they come closer to formalizing natural languages. They are efficiently parsable, yet also a linguistically expressive formalism: parsing CCG can be done in polynomial time. Some examples of CCGs are TAG, LIG, and Head grammars. CCGs can deal with coordination and cross-serial dependencies, unlike CFGs. Furthermore, CCGs can encode. For data-driven parsing, wide-coverage semantic construction and machine translation, CCGs can be utilized. Another benefit CCG is that it has the advantage that the inference rules can be fixed once and for all and therefore specification of a particular language grammar is entirely determined by the lexicon. CCGs are lexicalized, meaning that only a small number of rules are employed and all other syntactic phenomena are derived from the lexical entries of specific words.

8.5.1 Motivation: why do we need CCGs?

Expressive power of natural grammars must be at least that of CFGs, but many believe that it requires greater power. This raises the question of how much greater power is needed to capture natural grammars.

An important reason why we need CCGs is that they help modelling sentences constructed with **coordination**. For instance, the sentence

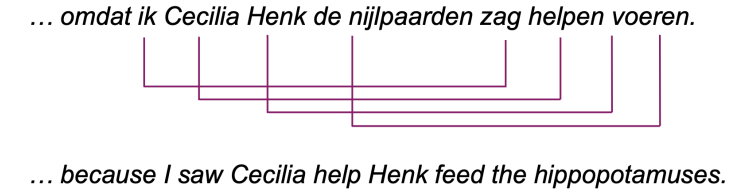
I like to play bridge and Sara likes to play handball.

can be generated with CFGs. On the other hand, the sentence

I like to play bridge and Sara handball.

cannot be generated with CFGs. Sentence construction of the latter example is called **coordination**. This phenomena can be described using "mildly context-sensitive grammars".

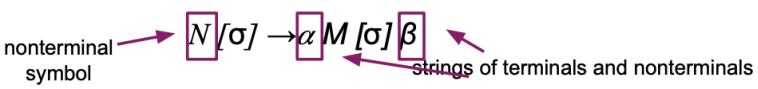
Another important reason why we need CCGs is the languages containing **cross-serial dependencies**. For instance, in Dutch subordinate clauses, the dependencies **cross** rather than nest as they do in English, e.g.



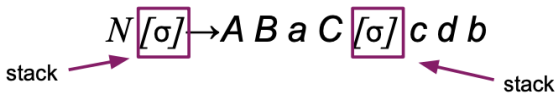
Languages containing **cross-serial dependencies** are not context-free. CCGs help us model these types of languages with cross-serial dependencies.

8.5.2 A Basic Example: Linear Indexed Grammars (LIG)

Structure of **linear indexed grammars** are similar to that of CFG:



except that it has stack associated with non-terminal symbol on the left side which is passed to **exactly one non-terminal** on the right side:



Let us now formalize the definition of linear indexed grammars:

Definition 8.10

*A **linear indexed grammar** G is a quintuple:*

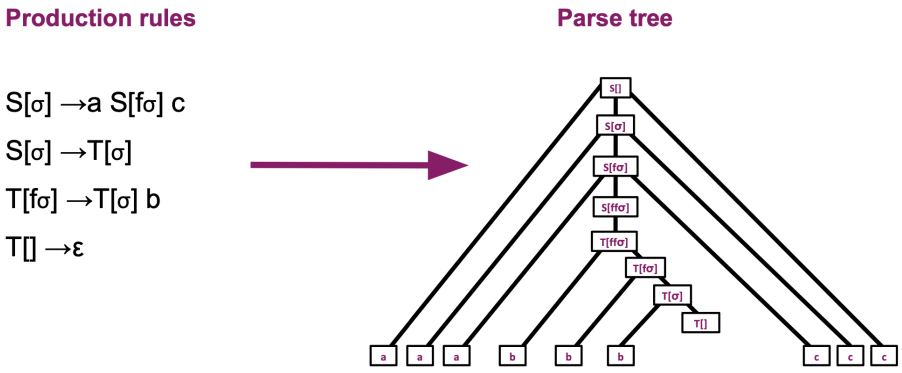
$$G =< \mathcal{N}, S, I, \Sigma, \mathcal{R} > \tag{8.1}$$

with

1. *A set of nonterminal symbols $\mathcal{N}$; written as upper-case letters, e.g., $\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3, \ldots$*
2. *A distinguished start non-terminal S*
3. *A finite set of indices I ; written as usual function letters, e.g., f, g, h, ...*
4. *An alphabet of terminal symbols Σ; written as lower-case letters, e.g., $a_1, \ldots$*
5. *A set of production rules $\mathcal{R}$ of one of the following forms:*
   - $N[\sigma] \rightarrow \alpha M[\sigma]\beta$
   - $N[\sigma] \rightarrow \alpha M[f\sigma]\beta$
   - $N[f\sigma] \rightarrow \alpha M[\sigma]\beta$

♣

Parsing $a^n b^n c^n$ with LIG is performed in the following way:



8.5.3 Definition of CCG

A CCG has two main parts: a **lexicon** that associates words with categories and **rules** that specify how categories can be combined into other categories. Together, they build derivation trees.

CCG lexicon is a finite set of word-category pairs and all information specific to a given language is specified in its lexicon, including valency, word-order, and semantics. CCG represents words with **categories**. The information about the structure is encoded in categories, unlike a CFG which encodes information about the structure with a set of rules. Here, category is like a function which takes as argument a certain type and produces resulting category, which can be seen as a function mapping from one category to another.

Lexical category is assigned to each word in a sentence, for example *chair* is of type *noun* and that means pair *(chair, N)* could be found in lexicon. Atomic categories are categories such as S, N, NP, PP. Complex categories, on the other hand, are built recursively from atomic categories. One of the examples of slash categories is S\NP.

Definition 8.11

*A **category** is a syntactic type that identifies a constituent as either **complete** or **incomplete**.* ♣

A category of a **complete** constituent is an atomic category, e.g.

Harry := NP

walks := S \NP,

where 'S' is the result, '\' is the direction, and 'NP' is the argument. The aboce can bee read as "the verb *walks* is a function seeking *a noun phrase (NP)* to its left and returning a complete *sentence (S)*.

Definition 8.12

For a finite set V and two distinguished symbols "/" and "\", the set $C(V)$ of categories *of V is inductively defined as follows:*

- $V \subseteq C(V)$.
- *If $v_1, v_2 \in V$, then $v_1/v_2 \in C(V)$ and $v_1 \setminus v_2 \in C(V)$.*

♣

Definition 8.13

*A **combinatory categorial grammar (CCG)** G is a quintuple*

$$\langle V_T, V_N, S, f, R \rangle \tag{8.2}$$

that consists of the following:

- *A finite set of terminals V_T (lexicon).*
- *A finite set of non-terminals V_N (atomic categories).*
- *A distinguished category $S \in V_N$.*
- *A function f mapping $V_T \cup \{\epsilon\}$ to finite subsets of $C(V_N)$, where $C(V_N)$ is the set of categories.*
- *A finite set of combinatory rules R, described below.*

♣

The language generated by a combinatory categorial grammar is the set of all sequences $a_1 \dots a_n \in V_T^*$ for which there are $c_1 \dots c_n \in C(V_N)$ such that I can transform $c_1 \dots c_n$ by repeated application of the combinatory rules.

Definition 8.14

Let $|_i$, for $i \leq n$, denote either \or /. For categories x, y, z, the combinatory rules are the following:

1. *Forward application:*

$$(x/y) \qquad y \mapsto x \tag{8.3}$$

2. *Backward application:*

$$y \qquad (x \setminus y) \mapsto x \tag{8.4}$$

3. *Generalized forward composition:*

$$(x/y) \qquad (\dots(y \,|_1\, z_1) \,|_2 \dots |_n\, z_n) \mapsto (\dots(x \,|_1\, z_1) \,|_2 \dots |_n\, z_n) \tag{8.5}$$

4. *Generalized backward composition:*

$$(\dots(y \,|_1\, z_1) \,|_2 \dots |_n\, z_n) \qquad (x \setminus y) \mapsto (\dots(x \,|_1\, z_1) \,|_2 \dots |_n\, z_n) \tag{8.6}$$

♣

Let us consider the following natural language query:

What states border Texas?

We can derive the meaning of this query using CCG as the following:

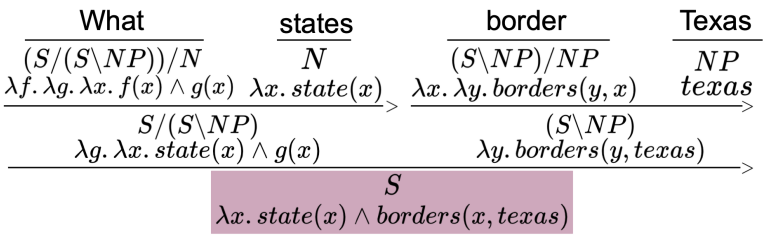


Figure 8.3: Deriving the meaning using CCG

8.5.4 Relationship to CFGs

Every CCG grammar restricts itself to a finite set of rules. If we consider higher-order composition rules, for every degree, there are forward rules and backward rules. However, each rule may give rise to infinitely many rule instances. We obtain a rule instance by substituting concrete categories for the variables. By contrast, CFGs have a finite set of non-terminals.

8.6 Parsing CCGs

There exist several algorithms for parsing CCG in the literature. Most of these algorithms have running time exponential in the length of the input string. Vijay-Shanker and Weir (1993) introduced the first polynomial-time algorithm, which decomposes CCG derivations into smaller parts that can be shared among different derivations. This idea is popular for parsing other equivalent mildly context-sensitive formalisms. It runs in $\mathcal{O}(n^6)$, where n is the length of the input string. However, this algorithm is complex, hard to understand and to prove correct. A simpler algorithm for CCG parsing was proposed by Kuhlmann and Satta (2014). This algorithm runs in $\mathcal{O}(n^6)$, having the same running time as the algorithm from Vijay-Shanker and Weir (1993). This algorithm reuses the idea of decomposing the CCG derivations into shareable parts, and is based on different decompositions of CCG derivations. This approach significantly reduces the number of steps required for parsing. However, it is still an open question whether CCG parsing can be done in polynomial time when the grammar is considered as part of the input.

8.6.1 Notation

Parsing can be viewed as a deductive process that seeks to prove claims about the grammatical status of a string. Here, we present parsing as deduction and parsing algorithm as a deduction system. The general form of a rule of inference is $\frac{A_1 \ldots A_k}{B}$. A grammatical deduction system is defined by the set of rules of inference and a set of axioms. We also say that B is consequence of $A_1, \ldots, A_k$.

Derivation trees consist of *unary* and *binary branchings*. Unary branchings, represented with dotted line in Figure 8.4 correspond to lexicon entries, while binary branchings correspond to composition rules. Binary branchings can have primary and secondary input. Primary input is the one that is "accepting" or in the previously used notation the part that defines a function. In Figure 8.4, primary input to binary branching is shaded. The yield of a derivation tree is, as previously, the sequence of its leaves from left to right while the type of a tree is the category at its root.

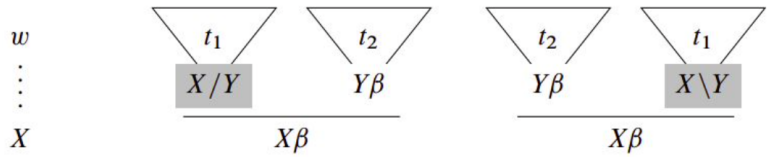


Figure 8.4: Recursive definition of derivation trees. Nodes labeled with primary input categories are shaded.

8.6.2 Straightforward CKY-style parsing algorithm

We now introduce the straightforward, CKY-style parsing algorithm for CCGs. The algorithm is specified in terms of grammatical deduction system. $\mathbf{w}$ is a string to be parsed, and each w_i is a lexical token. $w[i,j]$ denotes the substring $w_{i+1}...w_j$, and $w[i,i]$ is an empty string.

8.6.2.1 Items

The goal of the algorithm is the construction of the item $[S,0,n]$ which can be read as "we can construct a sentence from the string between 1 and n, namely whole string $\mathbf{w}$". The derivation tree leading to this outcome will usually have internal nodes of type $[X,i,j]$ which could be read in similar way: "we can construct a category X from the string between $i+1$ and j".

8.6.2.2 Axioms and inference rules

The steps of the algorithm are specified by means of inference rules over items. These rules implement the recursive definition of derivation trees. The construction starts with axioms of the form $[X,i,i+1]$ where $w_{i+1} := X$ is a lexicon entry; these items assert the existence of a unary-branching derivation tree for each lexical token w_{i+1}. There is one inference rule for every forward rule (application or composition):

$$\frac{[X/Y,i,j][Y\beta,j,k]}{X\beta,i,k}\ X/Y\ Y\beta \implies X\beta. \tag{8.7}$$

A symmetrical rule is used for backward application and composition. Here, we only specify the forward version of each rule, and leave the backward version implicit.

8.6.2.3 CCG parsing example:

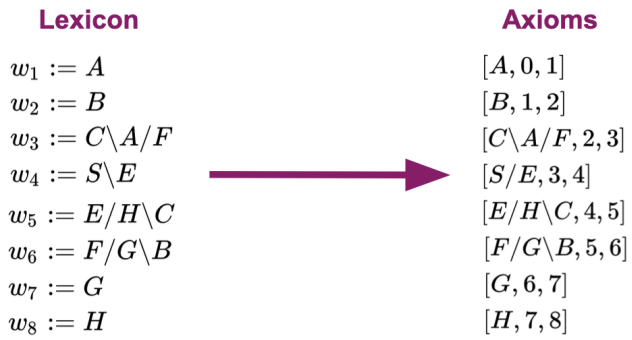
Consider the following toy lexicon:

$$\begin{aligned} w_1 &:= A & w_5 &:= E/H\backslash C \\ w_2 &:= B & w_6 &:= F/G\backslash B \\ w_3 &:= C\backslash A/F & w_7 &:= G \\ w_4 &:= S\backslash E & w_8 &:= H \end{aligned}$$

We want to parse the input string $\mathbf{w} = w_1 w2 \dots w_8$.

For the given toy lexicon, the axioms have the form $[X,i,i+1]$ where $w_{i+1} := X$ is a lexicon entry, as shown in the following:

.



We have the following inference rule:

$$\frac{[X/Y, i, j][Y\beta, j, k]}{X\beta, i, k}\, X/Y\ Y\beta \implies X\beta \tag{8.8}$$

Through CCG parsing, we obtain the following result:

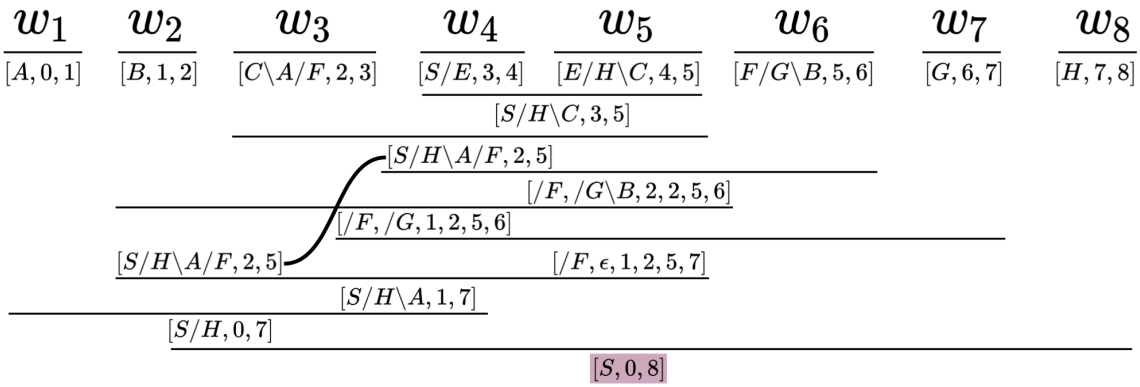


Figure 8.5: A sample derivation of the grammatical deduction system. First inference step triggers a new context item from a tree item; next inference step reuses the tree item (as indicated by the curve shown in bold), recombining it with the (modified) context item. Please see the lecture slides for the intermediate steps.

Chapter 9 Transliteration with WFSTs

In this chapter, we approach the task of transliteration—the transformation of a string from one character set to another character set. To this end, we use tools from formal language theory, specifically **finite-state machines**. This framework allows us to model transliteration probabilistically; decoding can then be viewed as solving a shortest path problem, for which a set of efficient algorithms can be employed.

As instances of finite-state machines, we introduce **Finite-State Acceptors (FSA)**, **Finite-State Transducers (FST)** and their weighted variants **Weighted Finite-State Acceptors (WFSA)** and **Weighted Finite-State Transducers (WFST)**. WFSAs and WFSTs are often depicted as graphs with weighted edges. Thus, any strings accepted or transduced can be represented as a path on a graph. This allows casting the task of finding the most likely string as a path-finding or transitive closure problem. Put more explicitly, we may be interested in finding "minimum" or "maximum"-weight paths between any two states respectively. For this purpose, we revisit the **Floyd-Warshall (FW)** algorithm which solves this problem with a run time complexity of $O(N^3)$. Next, we slow down the FW algorithm and present a matrix multiplication-based variant which runs in $O(N^4)$. This detour helps motivating that the FW algorithm is part of the family of dynamic programming and can thus be rewritten in a semiring. For this reason, we need to introduce a new operator on semirings, the Kleene star. Eventually, this "closes the loop" and results in **Lehmann's algorithm** which generalizes the FW algorithm to any closed semiring, running in $O(N^3)$. As we have encountered in various instances of this course, the computation of normalization constants can be intractable. Lehmann's algorithm helps to efficiently normalize a weighted finite-state transducer.

9.1 Transliteration

Transliteration is the task of mapping strings from one character set (i.e., an **alphabet**) to strings in another character set, where the goal is to replace characters with their approximate phonetic equivalents. Transliteration is used as part of many multilingual applications, such as corpus alignment or machine translation (see Fig. 9.1). Transliteration is one instance of the group of tasks called transduction.

Definition 9.1

Transduction *is the process of mapping (transducing) one sequence into another. A* ***transducer*** *is some program that performs such a mapping.* ♣

Figure 9.1: Example of the task of transliteration: transducing from English into Japanese.

We can formally model the task of transliteration as a probability distribution over strings built from an output vocabulary Ω given strings in an input vocabulary Σ. Finite-state machines provide one formalism for modeling this distribution.

9.2 Finite-State Machines

Definition 9.2

Finite-state machines *(FSM) or* ***finite-state automata*** *are theoretical models of computation that involve transitions between a finite number of states on* ***regular languages****. A regular language is any language that can be defined by a regular expression (see Ch.9 of Eisenstein 2019, for more detailed explanation).* ♣

FSMs can be divided into several sub-categories. Finite-state acceptors (FSA) are one-tape FSMs, i.e., they accept one sequence/string at a time. Finite-state transducers (FST) model the mapping/transduction of two sequences/strings. When the transitions are equipped with weights, FSAs and FSTs are denoted weighted finite-state acceptors (WFSA) and weighted finite-state transducers (WFST), respectively. An overview of FSMs is given in Fig. 9.2.

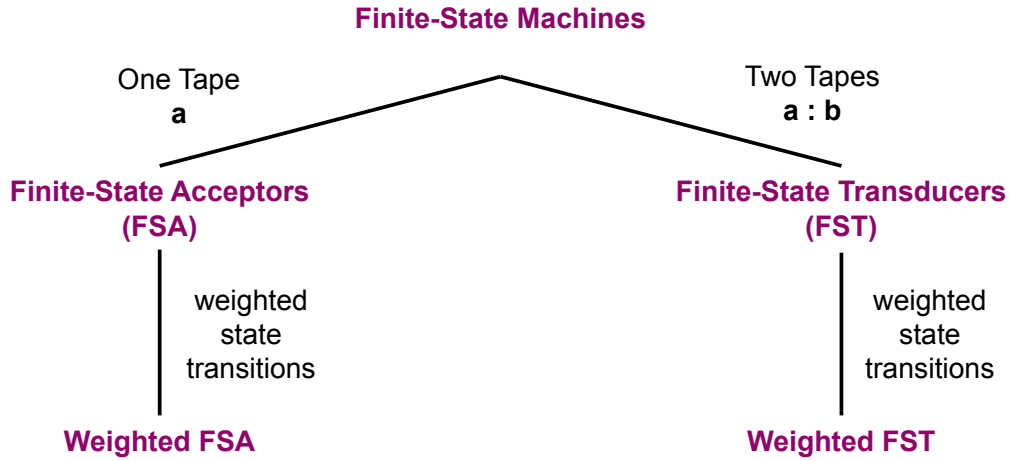


Figure 9.2: Overview over **Finite-State Machines (FSM)**

9.2.1 Weighted Finite-State Transducer

A weighted finite-state transducer (WFST) is a computationally tractable generalization of a FST, in which each accepting path is assigned a score, computed from the transitions, the initial state, and the final state (see Fig. 9.3). In the context of language, the states of a WFST can represent, e.g., terminals and non-terminals in a grammar; weights would then give us a sense of how likely each transition is. As we just hinted, a WFST therefore has an interpretation as a probabilistic model that can map sequences of finite length from input vocabulary Σ to an output vocabulary Ω.

Formally, we can represent a WFST as a tuple.

Definition 9.3

A WFST is a tuple $T = \langle Q, \Sigma, \Omega, \lambda, \rho, \delta \rangle$ *where:*

- Q *defines the finite set of states, including initial and ending state(s)*
- Σ *and* Ω *are the input and output vocabularies, respectively*
- $\lambda : Q \rightarrow \mathbb{R}$ *is a function that maps states to initial scores;* $\rho : Q \rightarrow \mathbb{R}$ *maps states to final scores*
- *The transition function* $\delta : Q \times (\Sigma \cup \epsilon) \times (\Omega \cup \epsilon) \times Q \rightarrow \mathbb{R}$ *maps transitions to scores, where* ϵ *is the empty symbol*

♣

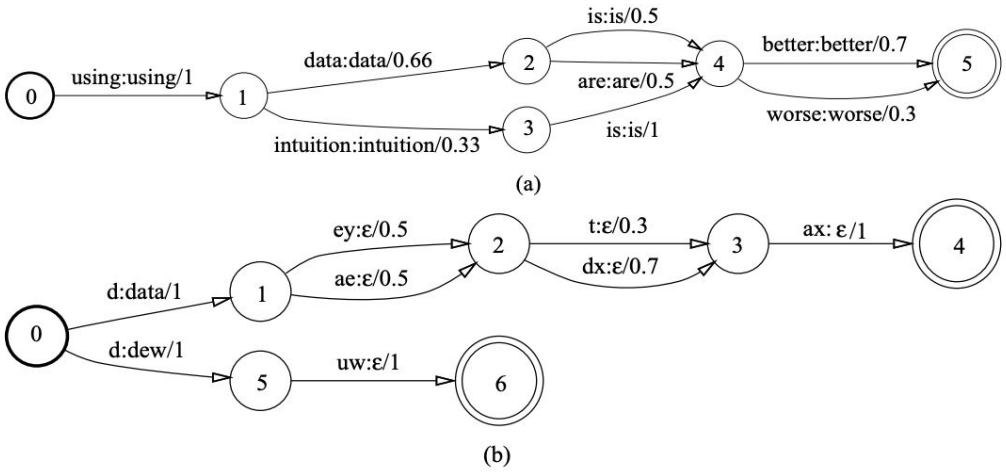


Figure 9.3: Two examples of Weighted Finite-State Transducers taken from Mohri, Pereira, and Riley 2008.

WFSTs (and WFSAs) are often depicted as graphs with weighted edges. In the context of transliteration, strings are then paths in this graph. This framing allows allows us to cast the task of finding the most likely string as a path-finding or transitive closure problem. Put more explicitly, we may be interested in finding "minimum" or "maximum"-weight paths between any two states respectively.

9.2.2 Scoring Paths in a WFST

Let $\mathbf{x} \in \Sigma^*$ and $\mathbf{y} \in \Omega^*$ be two strings from the input and output vocabulary, respectively. Recall that Σ^* refers to the Kleene closure of Σ, which is the set of any finitely long string that can be constructed from symbols contained within the vocabulary Σ. For an arbitrary WFST T, let denote the set of paths through T that accept $\mathbf{x}$ and $\mathbf{y}$, where formally, a **path** is a sequence of state transitions, which we denote as $\boldsymbol{\pi} = q_{\text{start}} - a : b \rightarrow q_1 - c : d \rightarrow \ldots q_{\text{end}}$. $\text{yield}_{\text{top}}(\boldsymbol{\pi}) \in \Sigma^*$ is the concatenation of the top symbols and $\text{yield}_{\text{bottom}}(\boldsymbol{\pi}) \in \Omega^*$ is the concatenation of the bottom symbols. Note that in general, there may be a large number of paths in $\Pi(\mathbf{x}, \mathbf{y})$. WFSTs with this property are said to be **ambiguous**. In the following, we focus on *unambiguous* WFSTs, meaning that each pair of strings has at most 1 accepting path, i.e., $|\Pi(\mathbf{x}, \mathbf{y})| \leq 1$.

We define the scoring function as the sum of weights $\text{score}(\boldsymbol{\pi}) = \lambda(q_{\text{start}}) + \delta(q_{\text{start}} - a : b \rightarrow q_1) + \ldots + \delta(q_n - c : d \rightarrow \ldots q_{\text{end}}) + \rho(q_{\text{end}})$. For simplicity, we will define the scoring function using underspecified notation and write τ for transition (including start and end weights).

Definition 9.4

$$\text{score}(\boldsymbol{\pi}) = \sum_{n=1}^{|\boldsymbol{\pi}|} \text{score}(\tau_n) \tag{9.1}$$

♣

Importantly, the score function decomposes over transitions which imposes a structure assumption.

9.2.3 WFSTs as Log-Linear Models

Our goal is to model the conditional distribution $p(\mathbf{y} \mid \mathbf{x})$. Given our definition of a scoring function in Equation (9.1), it is easy to see that we can use WFSTs in our log-linear modeling paradigm.

Definition 9.5 (WFST Log-Linear Model)

$$p(\mathbf{y} \mid \mathbf{x}) = \frac{1}{Z} \exp\{\text{score}(\mathbf{y}, \mathbf{x})\} \tag{9.2}$$

$$= \frac{1}{Z} \sum_{\boldsymbol{\pi} \in \Pi(\mathbf{x}, \mathbf{y})} \exp\left\{ \sum_{n=1}^{|\boldsymbol{\pi}|} \text{score}(\tau_n) \right\} \tag{9.3}$$

♣

We can then calculate our normalizing constant in the standard manner

$$Z = \sum_{\mathbf{y}' \in \Omega^*} \exp\{\text{score}(\mathbf{y}', \mathbf{x})\} \tag{9.4}$$

Unfortunately, Ω^* may be infinitely large. However, our structural assumptions allow us to still compute Z in an efficient manner.

9.3 Floyd-Warshall Algorithm

The Floyd–Warshall (FW) algorithm is a dynamic program for finding the transitive closure of a graph G with N vertices in a run time of $O(N^3)$ (see Algorithm 9.1). Put more simply, it is a shortest path-finding algorithm which operates on directed weighted graphs with positive or negative edge weights. Importantly, to ensure termination, there can be no negative cycles in the graph. In this section, we will point out the close connection between Floyd-Warshall, matrix multiplication and dynamic programming under semirings. We will subsequently use this connection to motivate **Lehmann's** algorithms, an algorithm that works for a large number of semirings and runs in $O(N^3)$.

Algorithm 9.1

```
def FLOYD-WARSHALL ALGORITHM(G):
    let W be a N × N adjacency matrix of a graph G
    let dist be a N × N array of minimum distances initialized to infinity.
    for each edge (u, v):                          ▷ "pre-processing" O(N)
        dist[u][v] ← W[u][v]                       ▷ Assign the weight of the edge (u, v)
    end for
    for each vertex v:                             ▷ O(N)
        dist[v][v] ← 0
    end for
    for k from 1 to N:                             ▷ O(N^3)
        for i from 1 to N:
            for j from 1 to N:
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] ← dist[i][k] + dist[k][j]
                end if
            end for
        end for
    end for
```

return dist

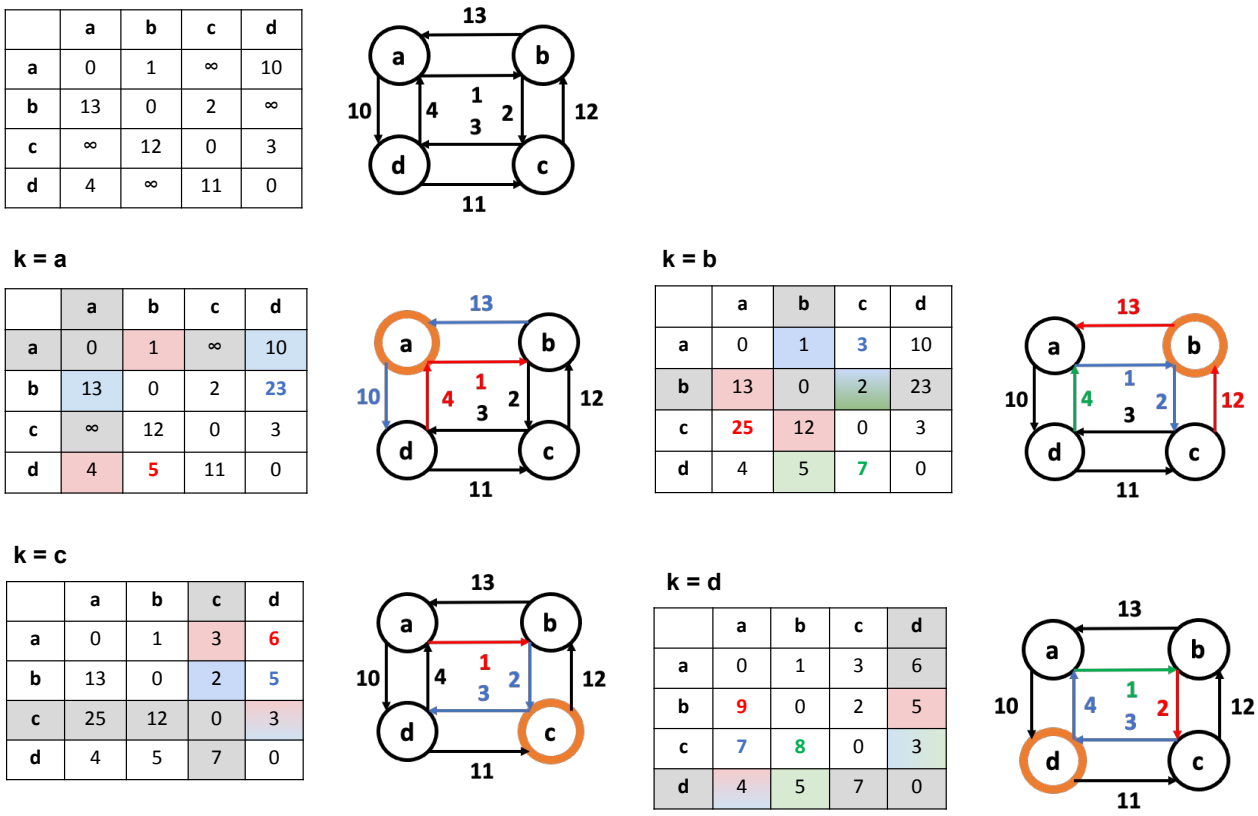


Figure 9.4: Example of Floyd-Warshall Algorithm applied to graph with $|N| = 4$ vertices

9.3.1 Matrix Multiplication

Multiplying an $N \times M$ matrix A by an $M \times P$ matrix B results in an $N \times P$ matrix C. In particular, this operation (in its simplest implementation) requires looping over the indices n from 1 through N and p from 1 through P in a nested loop as given by Equation (9.5).

$$C_{np} = \sum_{m=1}^{M} A_{nm} \times B_{mp} \tag{9.5}$$

Assuming A and B are square matrices of size $N \times N$, matrix multiplication has a run time complexity of $O(N^3)$.

Semiring-ify matrix multiplication. It turns out that we can semiring-ify matrix multiplication using the **Inside semiring** which is defined as $\langle \mathbb{R} \cup \{\infty\}, +, \times, 0, 1 \rangle$ (Huang 2008b). This can be seen in Algorithm 9.2, where using the operators specified by the Inside semiring clearly recovers matrix multiplication.

Algorithm 9.2

def *SemiringMatrixMultiplication(A, B):*
let A and B be square matrices of size $N \times N$
let C be an empty $N \times N$ matrix

for n *from 1 to* N*:*
  for p *from 1 to* N*:*
    $\text{sum} \leftarrow \mathbf{0}$
    for m *from 1 to* N*:*
      $\text{sum} \leftarrow \text{sum} \bigoplus A[n][m] \bigotimes B[m][p]$
    end for
    $C[n][p] \leftarrow sum$
  end for
end for
return C

Shortest-path matrix multiplication. Having semiring-ified matrix multiplication, we may now explore what problems we can solve by choosing other semirings. Under the **Tropical semiring**, given by the tuple $< \mathbb{R}^+ \cup \{\infty\}, \min, +, \infty, 0 >$, a matrix W naturally encodes the shortest path between two vertices of *exactly* length 1. Then, W^2 encodes the shortest path between two vertices of *exactly* length 2 and so on. Put in general terms, matrix W^k encodes the shortest path between two vertices of *exactly* length k. This is displayed visually in Fig. 9.5.

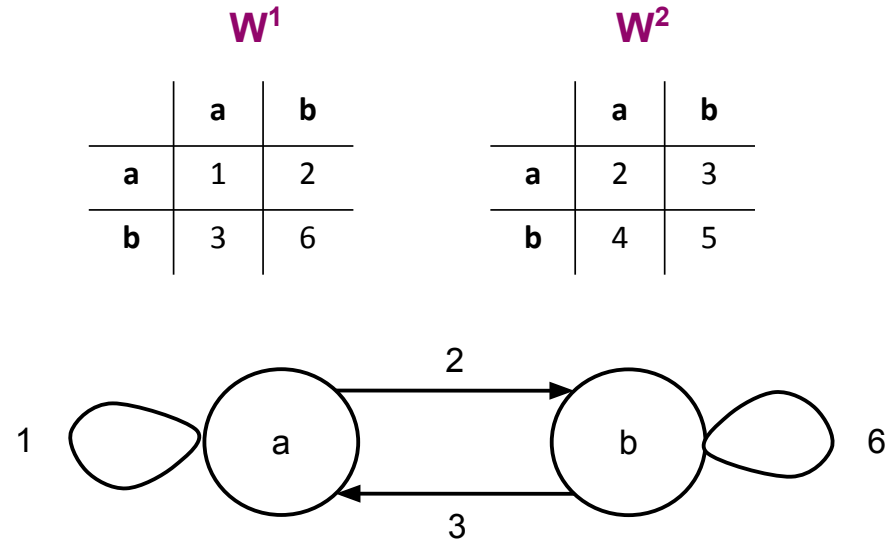


Figure 9.5: The matrix W^1 is encoding all shortest path of *exactly* length 1, W^2 is encoding shortest path of *exactly* length 2. The squared operation in this context refers to squaring under the Tropical semiring.

Ultimately, we are interested in the shortest path between any two vertices overall, irrespective of the number of steps. The matrix of all-pairs shortest paths can then be represented as $\min(W^0, W^1, \ldots, W^\infty)$ where the min is point-wise. This translates into an infinite matrix sum $W^0 \bigoplus W^1 \bigoplus \ldots \bigoplus W^\infty$. Consequently, repeated matrix multiplication can find the minimal path with a specific number of maximal edges (for example, to find minimal paths between all pairs of vertices with number of edges $\leq k$). In the same settings in which we use the Floyd-Warshall algorithm, this algorithm naively has a run time of $O(N^4)$. This is because we must run the matrix multiplication algorithm (which takes $O(N^3)$ time) a total of N times (see Fig. 9.6), where we need to consider only at most $k = N$ (or W^N) because there are no negative cycles and the shortest path will

consequently traverse at most N edges. Pseudocode is given in Algorithm 9.3.

Algorithm 9.3

def *MATRIX MULTIPLICATION FLOYD-WARSHALL(W):*
  let W^1 be the initial adjacency matrix featuring paths of length 1
  for *each vertex k in N:*
    $W^k = 0$ ▷ $= W^0$ *(zero-path length)*
  end for
  for *each vertex k in N:* ▷ $O(N)$
    $sum = \min(sum, W^{k-1}[n][m] + W^1[m][p])$ ▷ *matrix multiplication* $O(N^3)$
    $W^k[n][p] = sum$
  end for
  return W^k

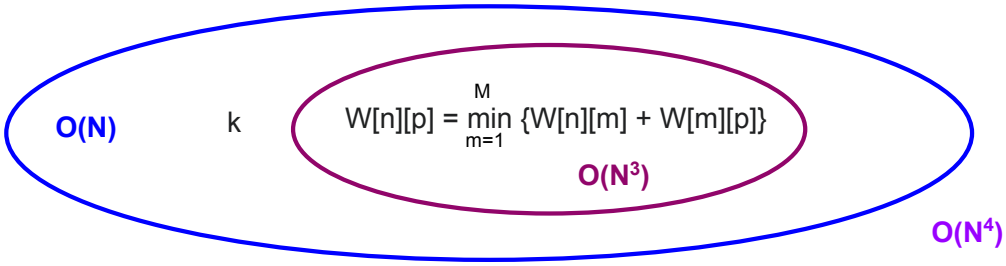


Figure 9.6: Illustration of the run time complexity of a matrix multiplication-based Floyd-Warshall shortest path algorithm. We have to loop at most N times over matrix multiplication with a Tropical semiring ($O(N^3)$).

9.4 Lehmann's algorithm

9.4.1 Kleene star operator

We have seen a matrix multiplication-based Floyd-Warshall algorithm which runs in $O(N^4)$. However, the run time complexity of the original Floyd-Warshall algorithm is only $O(N^3)$ — why is this? The Floyd-Warshall algorithm is closely related to Kleene's algorithm for converting a deterministic finite automaton into a regular expression. The infinite matrix sum $W^0 \bigoplus W^1 \bigoplus \ldots \bigoplus W^\infty$ is known as the Kleene closure of a matrix W^*. Under the Inside semiring $\langle \mathbb{R} \cup \{\infty\}, +, \times, 0, 1 \rangle$, we may derive Equation (9.6).

$$W^* := \bigoplus_{k=0}^{\infty} W^k = I + W \bigoplus_{k=0}^{\infty} W^k = I + WW^* \tag{9.6}$$

$$W^* - WW^* = I \iff (I - W)\, W^* = I \iff W^* := (I - W)^{-1} \tag{9.7}$$

$W^* := \bigoplus_{k=0}^{\infty} W^k$ holds by definition. The second equality holds also for standard summands as it partially expands the sum: $\sum_{n=0}^{\infty} x^n = 1 + \sum_{n=1}^{\infty} = 1 + x \sum_{n=0}^{\infty}$. The last equality $W^* := (I - W)^{-1}$ follows by definition again. Equation (9.6) defines a matrix inverse in the **Inside semiring**. But how can we define a matrix inverse in any given semiring? To answer this question, we need an additional operator r^*, which we call the Kleene star. We call a semiring closed, if it has a Kleene operator r^*. For the **Inside semiring**, we define

Semiring	$\langle \text{set}, \bigoplus, \bigotimes, \bar{0}, \bar{1} \rangle$	Infinite Semiring sum	Kleene star
Inside semiring	$\langle \mathbb{R} \cup \{\infty\}, +, \times, 0, 1 \rangle$	$I + W^1 + W^2 \dots$	$r^* = 1/(1-r)$
Counting semiring	$\langle \mathbb{N}, +, \times, 0, 1 \rangle$	$I + W^1 + W^2 \dots$	$r^* = \infty$
Tropical semiring	$\langle \mathbb{R}^+ \cup \{\infty\}, \min, +, \infty, 0 \rangle$	$\min(0, 1W, 2W, \dots)$	$r^* = 0$
Viterbi semiring	$\langle [0,1], \max, \times, 0, 1 \rangle$	$\max(I, W^1, W^2, \dots)$	$r^* = 1$
Boolean semiring	$\langle \{0,1\}, \text{OR}, \text{AND}, 0, 1 \rangle$	$\text{OR}(\text{True}, A, AANDA, \dots)$	$0^* = 1^* = \text{True}$

Table 9.1: Overview over semirings and their Kleene star operators r^* (see Abdali and Saunders 1985)

$r^* = \frac{1}{(1-r)} \Leftrightarrow r < 1$, which describes an infinite geometric series.

$$\bigoplus_{k=0}^{\infty} r^k = \sum_{k=0}^{\infty} r^k = \frac{1}{1-r} = (1-r)^{-1} \Leftrightarrow r < 1 \tag{9.8}$$

Analogously, if the largest eigenvalue of a matrix W is < 1, then we can transfer this concept to matrices and rewrite the geometric sum as

$$\bigoplus_{k=0}^{\infty} W^k = \sum_{k=0}^{\infty} W^k = (I - W)^{-1} \tag{9.9}$$

Following this idea, we can define a Kleene star operator for each semiring as shown in §9.4.1. If we consider the third column "Infinite Semiring sum", the derivation of the Kleene star operator for different semirings becomes more evident. Taking the general semiring $\bigoplus_{k=0}^{\infty} W \bigotimes$ as a starting point, we plug in the realizations for each semiring.

9.4.2 Generalizing Floyd-Warshall to any semiring

The previous exposition lays the foundation for **Lehmann's algorithm** (Lehmann 1977), the generalization of the Floyd-Warshall algorithm to any closed semiring. The algorithm runs in $O(N^3)$. When using the tropical semiring, it finds W^*; note that this instantiation of Lehmann's algorithm has a close relationship to the LU decomposition, which is often used to find matrix inverses. Pseudocode is shown in Algorithm 9.4, where $\text{dist}[k][k]^*$ is just the result of applying the Kleene star to $\text{dist}[k][k]$.

Algorithm 9.4

def *LEHMANN'S ALGORITHM(W):*
  W be a $N \times N$ array of minimum distances initialized to 0 infinity
  for *each edge (u, v):*
    $W[u][v] \leftarrow W[u][v]$ ▷ *this corresponds to W^1 (one step path length)*
  end for
  for *each vertex v:*
    $W[v][v] \leftarrow W[v][v]$ ▷ *this corresponds to W^0*
  end for
  for *k from 1 to N:*
    for *i from 1 to N:*
      for *j from 1 to N:*
        $W[i][j] \leftarrow W[i][j] \bigoplus (W[i][k] \bigotimes W[k][k]^* \bigotimes W[k][j])$
      end for

end for
end for
return W

9.4.3 Finding the normalization constant

Having introduced Lehmann's algorithm, we can now come back to our initial problem of using a WFST to parameterize a log-linear model, as discussed in Equation (9.2). Previously, we noted that brute-force computation of the normalizing constant Z in Equation (9.2) would be intractable since the output vocabulary Ω^* may be infinitely large. To compute Z efficiently, we take a linear-algebraic view on finding Z, i.e., computing the normalizing constant of $p(\mathbf{y} \mid \mathbf{x})$—which is encoded by a weighted finite-state acceptor—can be seen as a transitive closure problem. Formally, suppose our WFSA has N states: using notation from linear algebra, we may think of representing our WFSA as a vector α of start weights, where α_n is the weight of starting in state n, a vector β features end weights, where β_n is the weight of ending in state n, and a set of labeled transition matrices $W^{(\omega)}$ where $\omega \in \Omega \cup \epsilon$. The entry $W^{(\omega)}_{nm}$ is the weight given to the arc from state n to state m with the transition label ω.

	a	b	c
a	0.0	1.0	0.0
b	0.0	0.5	1.0
c	0.0	0.0	0.0

$$Z = \sum_{n=0}^{\infty} \left(\frac{1}{2}\right)^n = 2$$

$$\boldsymbol{\alpha} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \boldsymbol{\beta} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

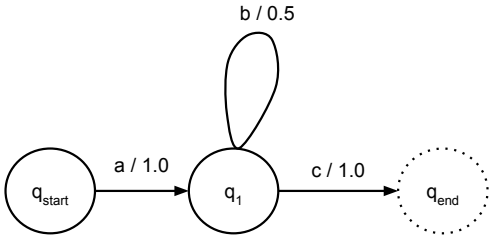


Figure 9.7: We take a linear-algebraic view on finding the normalizing constant Z of a WFST. Lehmann's algorithm and the Kleene star operator help us computing Z in $O(N^3)$.

Finding the normalizing constant Z is now a transitive closure problem as given by Equation (9.10).

$$Z = \sum_{\mathbf{y}' \in \Omega^*} \exp\{\text{score}(\mathbf{y}', \mathbf{x})\} = \sum_{\mathbf{y}' \in \Omega^*} \sum_{\boldsymbol{\pi}' \in \Pi(\mathbf{x}, \mathbf{y}')} \exp\left\{\sum_{n=1}^{|\boldsymbol{\pi}'|} \text{score}(\tau_n)\right\} = \boldsymbol{\alpha}^\top \left(\sum_{\omega \in \Omega \cup \{\epsilon\}} W^{(\omega)}\right)^* \boldsymbol{\beta} \quad (9.10)$$

Importantly, Lehmann's algorithm gives us the ability to efficiently normalize a distribution $p(\mathbf{y} \mid \mathbf{x})$ parameterized by a weighted finite-state transducer in $O(N^3)$ in the log-linear modeling paradigm. We this have the ability to optimize the log-likelihood in Equation (9.11) with gradient-based optimization to estimate the weights of our WFSA (or WFST).

$$\sum_{i=1}^{I} \log p(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}) = \sum_{i=1}^{I} \text{score}(\mathbf{y}^{(i)}, \mathbf{x}^{(i)}) - \log Z(\mathbf{x}^{(i)}) \quad (9.11)$$

Chapter 10 Machine Translation with Transformers

In this chapter, we introduce Machine Translation, the task of translating a sentence in a source language into a target language. We will focus on neural approaches and start with a general overview of the typical setup, from modeling to decoding. Then, we discuss the Transformer, a neural architecture upon which most current state-of-the-art models are based on. Finally, at the end of this chapter, we will briefly introduce and explain commonly used decoding strategies—algorithms for generating strings from probabilistic models.

10.1 Translation

Translation is the task of mapping strings from a source language into strings of a target language (see Fig. 10.1) that preserve the meaning as much as possible. Translation is widely used and is an important tool for human communication and the spread of information, knowledge and ideas.

Definition 10.1

Translation *is the process of mapping (translating) a sequence in a source language into a semantically preserving sequence in a target language.* ***Machine Translation*** *is when a program performs such a mapping.*

Figure 10.1: Example of translation: translating from English into Spanish.

We can formally model the task of machine translation as an optimization problem, where the goal is to find a sequence $\mathbf{y}^*$ built from an output vocabulary $\mathcal{Y}$ that maximizes some scoring function conditioned on an input sequence x build from an input vocabulary $\mathcal{X}$ (see Equation (10.1)).

$$\mathbf{y}^* = \underset{\mathbf{y} \in \mathcal{Y}}{\operatorname{argmax}} \operatorname{score}(\mathbf{x}, \mathbf{y}), \ \mathcal{Y} := \{\textsc{bos} \circ \mathbf{v} \circ \textsc{eos} \mid \mathbf{v} \in \Sigma^*\} \tag{10.1}$$

As for transliteration, the former can be achieved by developing a probabilistic model $p(y \mid x)$, that maps strings x from the input space $\mathcal{X}$ to strings y in the output space $\mathcal{Y}$. The output space is commonly capped to some maximal length and each string within this space is composed of strings from the vocabulary's Kleene closure and special BOS and EOS tokens, indicating beginning and end of sequences. Translation can then be represented as a two part problem, where we first create a scoring function, i.e. the probability distribution $p(y \mid x)$, which we then use to generate strings. The latter is also known as decoding. It is interesting to note that unlike transliteration, translation does not have one correct answer as there may be many ways to express the same meaning in a language.

10.2 Sequence-to-sequence Models

A common framework that has emerged in the context of neural Machine Translation is that of **sequence-to-sequence models**. These models are characterized by two parts, an encoder which takes the input $\mathbf{x}$ and maps it into a latent representation $\mathbf{z}$, and a decoder which is our learned probability distribution over all $\mathbf{y}$ and takes $\mathbf{z}$ as an input. Furthermore, considering that modeling $p(\mathbf{y} \mid \mathbf{x})$ over all possible strings is intractible due to the large search space, these models iteratively model the conditional probability distribution over each word; see Equation (10.2).

$$p(y \mid x) = \prod_{t=1}^{T} p(y_y \mid \mathbf{x}, y_1, ..., t_{t-1}) \tag{10.2}$$

Definition 10.2

Sequence-to-sequence models *typically consist of an* ***encoder*** *and a* ***decoder*** *and map sequences from one domain $\mathcal{X}$ into probability distributions over sequences of another domain $\mathcal{Y}$.*

- $\mathbf{z} = \text{encoder}(\mathbf{x}), \mathbf{x} \in \mathcal{X}$
- $p(\mathbf{y} \mid \mathbf{x}) = \text{decoder}(\mathbf{z}), \mathbf{y} \in \mathcal{Y}$

$$\underset{\boldsymbol{\theta}}{\text{argmax}} \sum_{i=1}^{N} \log p(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}; \boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\text{argmax}} \sum_{i=1}^{N} \sum_{t=1}^{|\mathbf{y}^{(i)}|} \log p(y_t^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{y}_{<t}^{(i)}; \boldsymbol{\theta}) \tag{10.3}$$

The parameters of these models are typically estimated by maximizing the log-likelihood of the data under the model p using gradient decent, where gradients can be computed using *backpropagation*. In other words, we try to solve the optimization problem in Equation (10.3). In order to make predictions with our locally normalized model, we generate one word at a time, each time conditioning the prediction on our input and all previously generated words (see Fig. 10.2). We discuss this process in more detail in §10.5

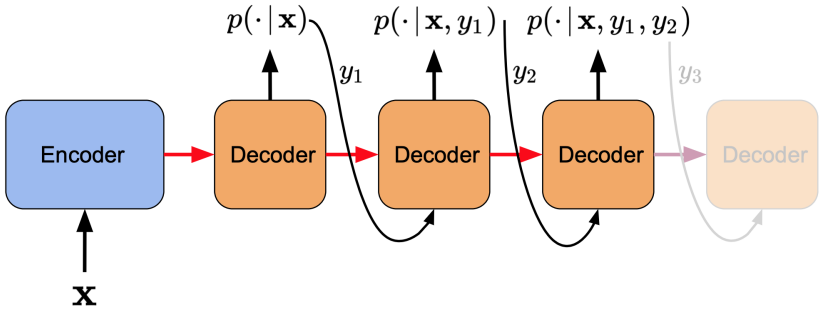


Figure 10.2: Inference with a sequence-to-sequence model.

10.3 The Attention Mechanism

The first neural sequence-to-sequence models were parameterized by recurrent neural networks (RNN). However, under the standard RNN paradigm, the generation is solely based on the most recent output of this model and is limited to the information that is forwarded though this fixed-size vector. Not only is it more difficult to compress information from longer and longer sequences into this vector, but the data that may be relevant at

different steps can change too. As a solution to this problem, the **Attention Mechanism** was introduced. The attention mechanism can be seen as a "soft version of a hash table" that allows the model to place emphasis on outputs from different time steps. More formally, it can be considered as a differentiable relaxation of the hard assignment to a specific value from a hash table, where a custom context vector is computed for each time step.

Thus, the attention mechanism computes the context, a convex combination $\mathbf{c}$ of attended steps' values V, where the coefficients are the normalized scores between a query $\mathbf{q}$ from the current and the keys K of the attended positions (see Equation (10.4)).

Definition 10.3

*The **Attention Mechanism** enables a model to "attend" to information from different time steps by taking a convex combination of a model's states to build the context vector. It can be seen as a soft version of a hash table, which enables direct access to the relevant information in attended positions. It can be formalized as:*

$$\mathbf{c} = \text{softmax}(\text{score}(\mathbf{q}, K) * V \tag{10.4}$$

where $\mathbf{q}$ is the query, K the keys, V the values and $\mathbf{c}$ the resulting context.

Intuitively, the scoring function assesses the importance of previous states for the current prediction by returning high scores for more relevant parts and lower scores for less relevant ones. The softmax normalization then allows us to build a convex combination of states, thus forming individualized contexts at each step. Note that different approaches can be taken for building the queries, keys and values. Perhaps the simplest method is to directly assign them as hidden states, i.e. $\mathbf{h_i^{(e)}} = \mathbf{k_i} = \mathbf{v_i}$ (from the encoder) and $\mathbf{h_i^{(d)}} = \mathbf{q_i}$ (from the decoder); see such an example with the decoder attending the encoder steps in Fig. 10.3. However, they can also be more elaborate, e.g. by constructing different linear projections from these hidden states. An example of such projections can be seen in Table 10.1 where $\mathbf{q}$ is the query and K, V correspond to vertically stacked attended states, and W_q, W_k and W_v are our projection matrices.

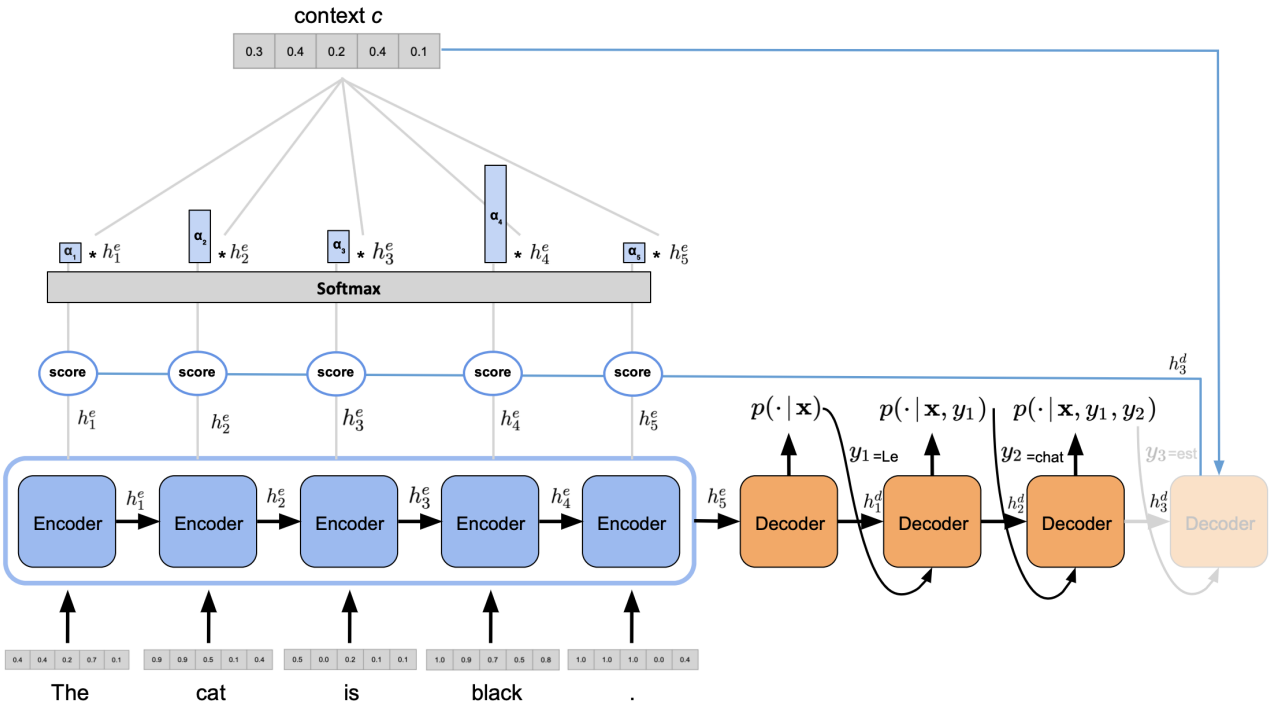


Figure 10.3: Example of cross-attention in a sequence-to-sequence model with RNN.

Furthermore, there different types of attention mechanisms. A common form is **cross-attention** (often interchangeably used with "attention"), where the decoder attends to hidden states of the encoder (see Fig. 10.3). This form propagates important information from encoder steps to the current decoding step t. Another form is **self-attention**, where the encoder and/or decoder attend different steps from their own hidden states, i.e. propagating relevant information from within the source/target sequence to the current encoding/decoding step t. Note that in Machine Translation the decoder will never attend to decoding steps $i > t$, as they are unknown at step t. Refer to Table 10.1 for an overview of what query, keys and values are in these different forms of attention.

	Cross-attention	Self-attention (encoder)	Self-attention (decoder)
Query, keys and values without projection	$\mathbf{q_t} = \mathbf{h_t^d}$ $\mathbf{k_i} = \mathbf{v_i} = \mathbf{h_i^e}, i \in 1...n$ $K = V = H^e$	$\mathbf{q_t} = \mathbf{h_t^e}$ $\mathbf{k_i} = \mathbf{v_i} = \mathbf{h_i^e}, i \in 1...n$ $K = V = H^e$	$\mathbf{q_t} = \mathbf{h_t^d}$ $\mathbf{k_i} = \mathbf{v_i} = \mathbf{h_i^d}, i \in 1...m$ $K = V = H^d$
Query, keys and values with linear projection $W_q, W_k, W_v \in \mathbb{R}^{d*d}$	$\mathbf{q_t} = \mathbf{h_t^d} * W_q^d$ $\mathbf{k_i} = \mathbf{h_i^e} * W_k^e, i \in 1...n$ $\mathbf{v_i} = \mathbf{h_i^e} * W_v^e, i \in 1...n$ $K = H^e * W_k^e$ $V = H^e * W_v^e$	$\mathbf{q_t} = \mathbf{h_t^e} * W_q^e$ $\mathbf{k_i} = \mathbf{h_i^e} * W_k^e, i \in 1...n$ $\mathbf{v_i} = \mathbf{h_i^e} * W_v^e, i \in 1...n$ $K = H^e * W_k^e$ $V = H^e * W_v^e$	$\mathbf{q_t} = \mathbf{h_t^d} * W_q^d$ $\mathbf{k_i} = \mathbf{h_i^d} * W_k^d, i \in 1...t$ $\mathbf{v_i} = \mathbf{h_i^d} * W_v^d, i \in 1...t$ $K = H^d * W_k^d$ $V = H^d * W_v^d$

- n is the length of the input sequence
- m the length of the output sequence
- the subscript $\in \{e, d\}$ specifies whether the hidden state is from the encoder or decoder
- the capital+bold notation represents horizontally stacked vectors
- $\mathbf{q_t}, \mathbf{k_i}, \mathbf{v_i}, \mathbf{h_i^e}, \mathbf{h_t^d} \in \mathbb{R}^{1*d}$
- $H^e \in \mathbb{R}^{n*d}, H^d \in \mathbb{R}^{m*d}$

Table 10.1: Variations of attention mechanisms and queries q, keys k and values v

Finally, there are a number of different scoring functions that can be used in the attention mechanism; the performance of these different functions has been widely studied. An overview of the most popular functions can be seen in Table 10.2.

Name	Score function	Citation
Content-base attention	$\text{score}(\mathbf{q_t}, \mathbf{k_i}) = cosine(\mathbf{q_t}, \mathbf{k_i})$	Graves, Wayne, and Danihelka 2014
Additive	$\text{score}(\mathbf{q_t}, \mathbf{k_i}) = \mathbf{v_a^T} tanh(W_a(\mathbf{q_t}, \mathbf{k_i}))$	Bahdanau, Cho, and Bengio 2016
Location-Base	$\text{score}(\mathbf{q_t}, \mathbf{k_i}) = \text{softmax}(W_a \mathbf{q_t})$	Luong, Pham, and Manning 2015
General	$\text{score}(\mathbf{q_t}, \mathbf{k_i}) = \mathbf{q_t^T} W_a \mathbf{k_i}$	Luong, Pham, and Manning 2015
Dot-Product	$\text{score}(\mathbf{q_t}, \mathbf{k_i}) = \mathbf{q_t^T} \mathbf{k_i}$	Luong, Pham, and Manning 2015
Scaled Dot-Product	$\text{score}(\mathbf{q_t}, \mathbf{k_i}) = \frac{\mathbf{q_t^T} \mathbf{k_i}}{\sqrt{n}}$	Vaswani et al. 2017

Table 10.2: Score functions for the attention mechanism

10.4 The Transformer

A recent breakthrough for neural Machine Translation has been brought about by the **Transformer** architecture. The Transformer is a model architecture introduced in the paper "Attention is all you need" by Vaswani et al. 2017, that moved away from recurrent layers and is solely based on the attention mechanism (see Fig. 10.4). Thus, in the absence of the slow sequential recurrence, the Transformer is also characterized by its better parallelizability and efficiency to train when compared to RNNs.

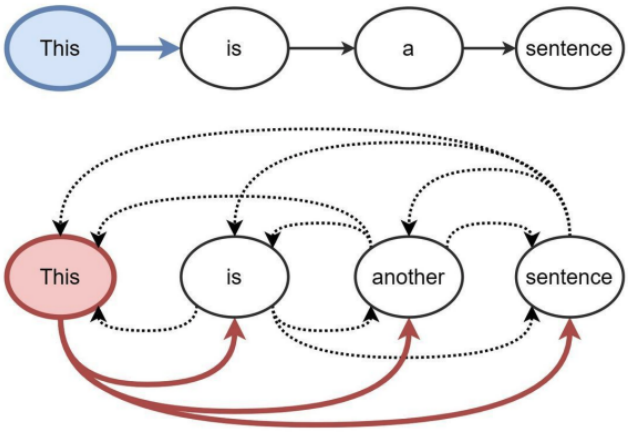


Figure 10.4: Information flow between RNN (top) vs. Transformer (bottom)

In the following paragraphs, we will introduce the different building blocks of the Transformer architecture and explain their purpose. To this end, we will refer to the letters in Fig. 10.5, to situate the different parts within the overall architecture.

A Just as in chapter 3, we map one-hot encoded words into dense representations, i.e., **word embeddings**, where semantically similar words are clustered together.

B The Transformer does not follow the recurrent structure seen in RNNs. Thus, it needs another way to express the order of words in a sequence. This is where **positional encodings** come into place. These are vectors of the same dimension as the input embeddings and encode a specific location within the sequence. This information about position is incorporated into the model pipeline simply by adding the positional encoding to the input embedding. Positional encodings can be absolute or relative and there are many choices for them, either learned or fixed. The original Transformer uses a fixed and absolute encoding through sine and cosine functions of different frequencies.

C **Self-attention** is one of the versions of attention within the Transformer architecture. It corresponds to the scaled dot-product attention described in Table 10.2, where queries Q, keys K and values V are all based on the hidden representations produced by the encoder. Refer to Table 10.1 for details about the matrix dimensions.

D **Masked self-attention** follows the same paradigm as standard self-attention. However, specifically in the decoder, positions after the current time step cannot be attended to when modeling $p(y_t \mid \mathbf{x}, \mathbf{y}_{<\mathbf{t}})$. Thus, a masked self-attention mechanism is used in the decoder, which additionally takes a set of positions that are masked in the attention computations (i.e. context weights are set to 0 at these positions).

E **Cross-attention** corresponds to the scaled dot-product attention as in Table 10.2. This version is used to allow the decoder to attend the encoder. Therefore queries Q are based on hidden representations of the decoder and keys K and queries Q on the hidden representation of the encoder.

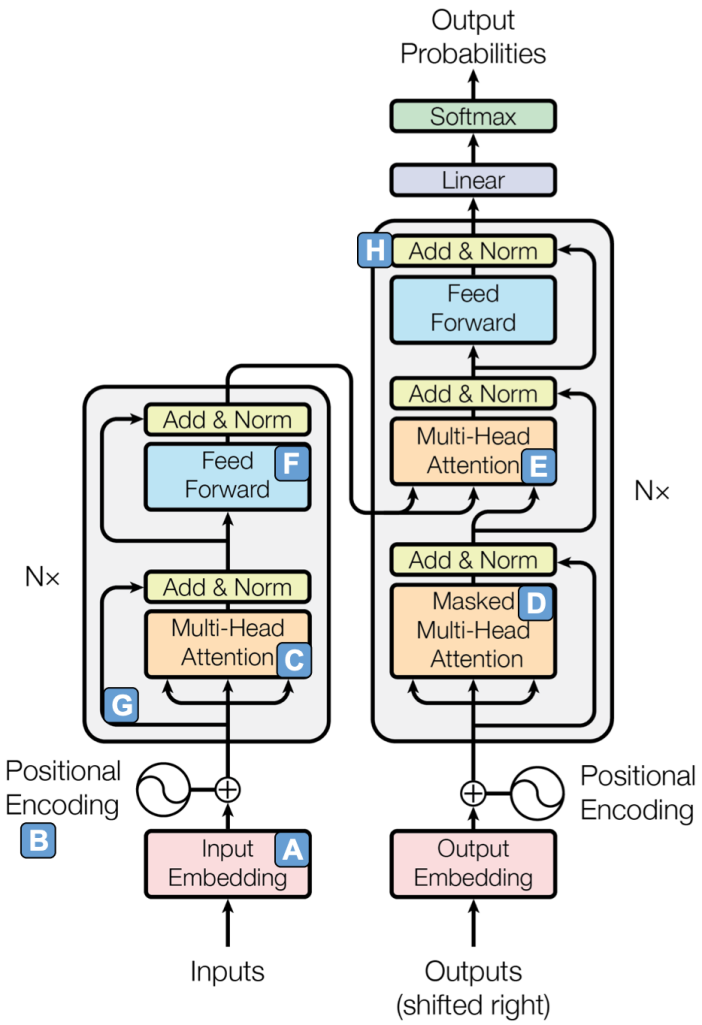


Figure 10.5: Transformer architecture

F The **feed forward** layers within the encoder and decoder are of the same form as discussed in chapter 3, i.e., linear projections followed by non-linearities.

G The **residual connection** (also termed identity shortcut connections) in the encoder and decoder passes an input to a subsequent layer without applying a transformation. In words, these connections are "routing information" directly into subsequent layers. They are a practical solution to mitigate the problem of vanishing gradients and may have a slight impact on performance gain.

H **Layer normalization** normalizes individual inputs to a layer such that collectively the elements in the input have mean 0 and variance 1. Normalization in neural networks has shown to help with numerical stability and convergence rates during training.

Note that each layer within the encoder and decoder contains multiple attentions mechanisms in parallel. They are also called heads and form together **Multi-Head Attention** mechanisms. In the case of the original Transformer the queries, keys and values are obtained via head-specific projections and the output is formed by a further projection of the concatenated outputs of the different heads.

10.5 Decoding

The previous sections focused on the modeling of probability distributions over strings for Machine Translation. The part of Machine Translation that we have not considered yet is **Decoding**.

Definition 10.4

Decoding *is the process of generating strings according to predictions from a model.* ♣

At inference time, we wish to find a translation $\mathbf{y}$ for our source sentence $\mathbf{x}$. Specifically, we want a $\mathbf{y}$ that is assigned high probability by our model p, as in Equation (10.1). When decoding, we are searching over all sentences in an exponential space. In previous chapters, we have approached this problem using dynamic programming. However for locally-normalized neural models this space is non-decomposeable, i.e., we generate words according to Equation (10.2) in a non-markovian structure. Thus our dynamic programming approaches would be no faster than brute force methods. This is why commonly used decoding strategies are heuristic search methods (i.e. **Greedy Search**, **Beam Search**, ...) or sampling schemes (i.e. **Ancestral Sampling**, **Nucleus/top-k Sampling**, ...). Note that maximizing y locally at each step (i.e. **Greedy Search**) will most likely not result in the global optimum. The most common deterministic heuristic approach is Beam Search.

Beam Search can be seen as a pruned breadth-first search where the breadth is limited to size k. Thus, we only keep the k most likely paths at every step of the search (see Fig. 10.6). Despite the lack of formal guarantees for finding an optimal solution, Beam Search works competitively well in practice.

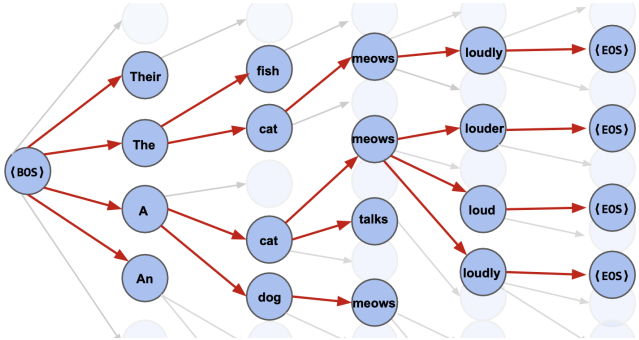


Figure 10.6: Example of beam search with $k = 4$ (at each step the 4 most likely paths are kept)

Another choice for decoding is **Sampling**, where words are sampled iteratively from the distribution $p(y_t \mid \mathbf{x}, \mathbf{y}_{<t})$. By the chain rule of probability, when $p(y_t \mid \mathbf{x}, \mathbf{y}_{<t})$ is unaltered, this is equivalent to directly sampling from the model $\mathbf{y} \sim p$. In **Nucleus Sampling**, we alter $p(y_t \mid \mathbf{x}, \mathbf{y}_{<t})$, restraining the words that can be sampled to only those within the top words that cover $\mathrm{p}\%$ of the probability mass.

Chapter 11 Axes of modeling

Most problems in NLP can be boiled down to building some function $f : \mathcal{X} \to \mathcal{Y}$ that maps an *input space* $\mathcal{X}$ to a *label space* $\mathcal{Y}$. For this purpose, we must understand at a high level what this mapping entails. Some questions that arise here are:

- What types of functions can f be for the given problem?
- What is our metric of success for f and how can we optimize for this metric in its parameterization?
- How should we choose a final f for our problem?

11.1 Modeling

In the context of classification, we can divide models into two classes. There are benefits and drawbacks to both approaches.

- **Probabilistic models.** These models represent conditional distributions over $\mathcal{Y}$. To train a classifier, one uses the dataset to compute $p(y \mid x)$, for $x, y \in \mathcal{X} \times \mathcal{Y}$. A classification rule based on these conditional probabilities is then used to choose a label for y.
- **Non-probabilistic models.** These models typically operate by learning rules to separate the feature space. The model then returns the class associated with the space where it believes a sample comes from. Examples of these approaches include perceptrons (Fig. 11.1) and support-vector machines.

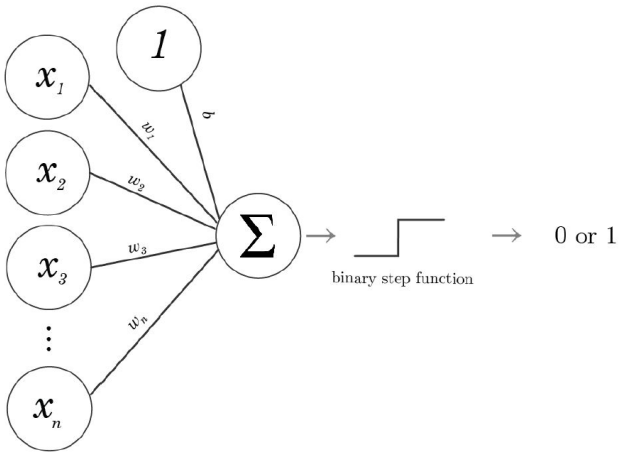


Figure 11.1: Perceptrons

One advantage of a probabilistic model is that it can produce estimates that indicate how confident we are of a prediction. One can also use the model to simulate new samples and even perform outlier detection. A drawback is that probabilistic models require the choice of an adequate parametric family of distributions. This choice induces bias in the model, which can be harmful when our assumptions are incorrect.

Non-probabilistic models can often be more interpretable than their probabilistic counterparts. The downside is that classifiers produced in this way give fixed predictions, without a metric that quantifies the uncertainty behind the prediction.

In these course, we have mostly considered probabilistic models. Probabilistic models can be further divided into two categories (Fig. 11.2):

- **Generative.** Such approaches fit a distribution p over $\mathcal{X} \times \mathcal{Y}$. For use in classification tasks, we then compute the conditional distribution $p(y \mid x)$, for $x, y \in \mathcal{X} \times \mathcal{Y}$ by Bayes' rule. Examples of these types of models include n-gram models, Markov random fields, and some recurrent neural networks.
- **Discriminative.** Such approaches skip the fitting of p and directly fit a distribution $p(y \mid x)$. Examples of these are logistic regression, conditional random fields, and some recurrent neural networks.

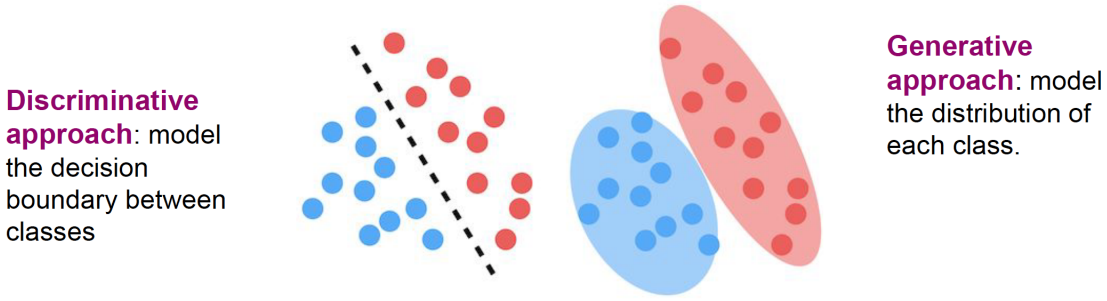


Figure 11.2

11.2 Loss functions and evaluation metrics

A loss function maps a model and a dataset to a real number. It should quantify how well the model performs the given task, as defined by the dataset. Model parameters are typically chosen by minimizing such a loss function, hence the choice of a loss function influences the quality of the trained classifier. Formally, let us consider a *parameter space* Θ that defines our possible models. For a fixed dataset $\mathcal{D} = \{(x_i, y_i)\}$ consisting of n training pairs, the loss function is a function $L : \Theta \to \mathbb{R}$. Usually, a loss function is defined via an intermediate function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$, where L is then defined as

$$L(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f_\theta(x_i)). \tag{11.1}$$

The parameters of a model $\theta \in \Theta$ can then be chosen to minimize L. Note that these loss functions are typically minimized using analytical or computational methods. Thus, there are some desired properties for these functions:

- Convexity: this ensures convergence guarantees by iterative methods.
- Differentiability: this facilitates the mathematical analysis.
- Robustness: the function should lead to models that are not sensitive to the noise in the dataset.
- Tractability: one should be able to minimize this function with a computationally efficient procedure.
- Reliability: the function should lead to meaningful models that match reality.

11.2.1 Maximum likelihood estimation

When fitting probabilistic models, a common loss function is the negative log-likelihood of the data:

$$L(\theta) = -\sum_{i \leq n} \log p(y_i \mid x_i, \theta). \tag{11.2}$$

Observe that minimizer of this loss is a maximum likelihood estimator (MLE). Such estimators have attractive properties:

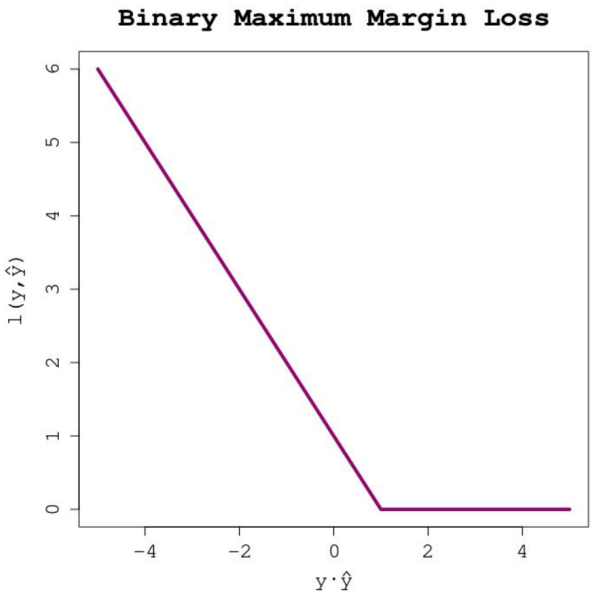


Figure 11.3: Hinge loss

- **Consistency.** Let $\mathcal{F} = \{p_\theta \mid \theta \in \Theta\}$ be a parametric family of distributions. Assume that $\mathcal{D}$ is drawn from p_{θ^*}. Then the MLE $\hat{\theta}$ converges in probability to θ^* as $n \to \infty$.
- **Asymptotic efficiency.** The MLE minimizes $\mathbb{E}\left[(\theta - \theta^*)^2\right]$ as $n \to \infty$.
- **Lowest KL-divergence with respect to the true distribution.** Even if $\mathcal{D}$ is a sample from a distribution $\tilde{p}$ outside the model family $\mathcal{F}$, the MLE yields the distribution in $\mathcal{F}$ that minimizes the KL-divergence with respect to $\tilde{p}$.

MLEs have their downsides as well. They can only be computed for probabilistic models. Also, if N is not sufficiently large, they show high variance and risk overfitting.

11.2.2 Other losses

Other loss functions can be employed as the objective when choosing model parameters, which may be more suitable the MLE in certain situations. To ensure robustness, some losses require not only a correct prediction, but also that the score given by a classifier to a particular class is above a certain margin. One example of such a loss is the *hinge loss* (Fig. 11.3):

$$\ell(y, \hat{y}) = \max\{0, 1 - y\hat{y}\}\,. \tag{11.3}$$

Some losses, like the logistic loss (Fig. 11.4), are designed to avoid the influence of outliers:

$$\ell(y, \hat{y}) = \log\left(1 + e^{-y\hat{y}}\right). \tag{11.4}$$

11.3 Regularization

It is important that any fitted model generalizes well, i.e., correctly predicts labels on unseen data. To this end, we must avoid overfitting the model to our training data, which we measure as the model's **generalization error**.

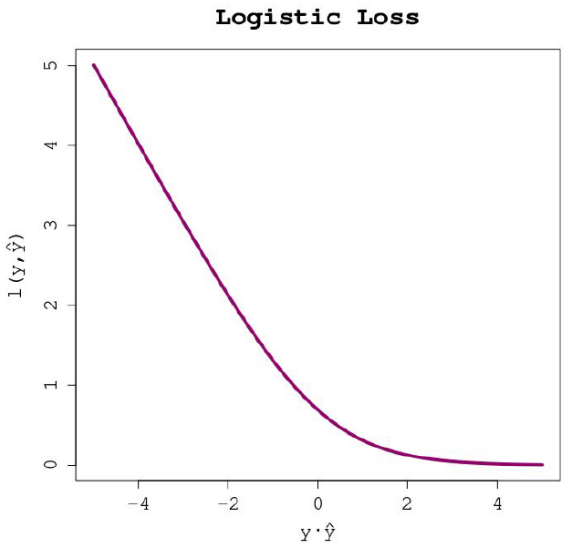


Figure 11.4

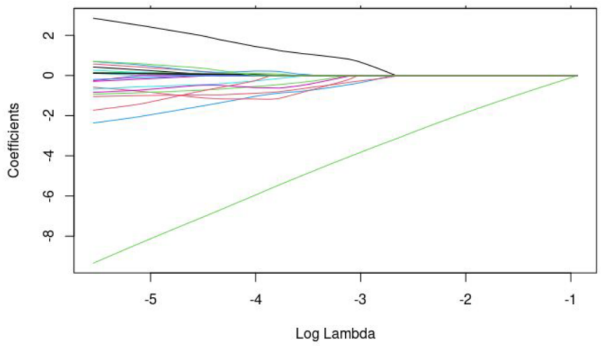


Figure 11.5: LASSO regularization

Definition 11.1

Consider a learning algorithm $A(\mathcal{D})$ that takes as input a dataset and outputs a parameter in Θ. The ***generalization error*** *of a learning algorithm A is given by $\mathbb{E}_{X,Y,\mathcal{D}}\left[\ell(Y, f_{A(\mathcal{D})}(X))\right]$. Overfitting occurs when there is an unacceptable difference between the generalization error and the empirical loss:*

$$\frac{1}{n}\sum_{i\leq n}\ell\left(y_i, f_{A(\mathcal{D})}\left(x_i\right)\right). \tag{11.5}$$

♣

One mechanism for reducing generalization error comes from finding a balance between model bias and variance, otherwise known as the bias–variance trade-off. This can be done by, e.g., adding a **regularization term** to the loss function. Perhaps the most well-known class of regularization terms penalize the norm of the trained parameter, where the choice of norm can have a large effect on the resulting parameters. For example, adding the L_1-norm of model parameters to the training objective—which in the context of regression is known as LASSO regression—encourages many coefficients to be zero. Using the L_2-norm disproportionately penalizes large parameters. See Fig. 11.5 and 11.6.

Interestingly, it can be demonstrated that minimizers of loss functions that consist of the negative log-likelihood and specific regularization terms are also the resulting *maximum a posteriori* (MAP) estimates in the Bayesian inference paradigm for specific prior distributions. More formally, for certain prior distributions $p_0(\theta)$ over the parameter space Θ, the MAP estimate computed from the posterior distribution of $\theta \in \Theta$ is equivalent

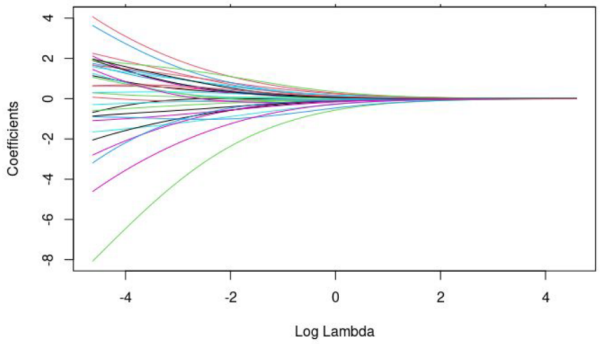


Figure 11.6: Ridge regularization

to a regularized MLE estimate.

11.4 Model evaluation in NLP

Once a model has been fitted, we quantify its performance using some **evaluation metric**. Ideally, the evaluation metric should be the loss function used to fit the model. Unfortunately, evaluation metrics are often non-convex or even non-differentiable. They can also be computationally expensive and do not include regularization terms. Even for cases when it is possible to fit a model using the evaluation metric, the gains are often marginal with respect to simpler loss functions.

11.4.1 Intrinsic Evaluation

We can evaluate models based on some criteria that we think are appropriate, even when these criteria are not directly related to the end use of the model. Such evaluation metrics are called *intrinsic*. One example of such a metric is the log-likelihood of some held-out test set. In the case of word embeddings, an intrinsic evaluation metric is the *cosine similarity*. The cosine similarities of pairs of embeddings is compared to similarity scores given by humans for the corresponding pairs of words.

In the context of classification, some examples of such metrics are:

- Accuracy: the ratio of correctly classified examples to the total of examples in a testing set.
- Precision: the ratio of correctly classified positive examples to the number of examples classified as positive.
- Recall: the ratio of correctly classified positive examples to the number of positive examples.
- F1 score: the harmonic mean between precision and recall.

Depending on the use-case of the model, some metrics may be more important that others. For example, it can be more dangerous to fail to diagnose a life-threatening disease than to misdiagnose it.

11.4.2 Extrinsic Evaluation

Other type of metrics are *extrinsic validation metrics*. They measure how well the fitted model performs on a downstream task. For example, for a language model, an extrinsic validation metric quantifies how well it performs in a task like translation. In the case of word embeddings, an extrinsic evaluation metric measures how the bias present in the text propagates in the embeddings to downstream tasks.

11.5 Model selection

Fitting a model is only one part of the learning process. Another important part is the process of choosing a final model from a set of candidate models, conditional on our data. Model selection typically has one of two goals: choosing a model that best explains the data-generating process or choosing a model that performs well by some performance metric. The two are complementary, but models which are best for one aren't necessarily best for the other (e.g., neural nets often perform very well, but generally not very helpful for inference due to interpretability).

11.5.1 Common Mistakes

There are several common mistakes made when performing model selection. If you perform multiple statistical tests (otherwise known as **multiple testing**) without adjustment, the type 1 error is no longer bounded by the significance level of the tests Think: If you try 20 different models and one of them ends up being better than your benchmark, the probability of this being due to random chance increases. Misleading results can also be cause by **data leakage**—giving your model access to the data on which it will ultimately be evaluated.

11.5.2 Cross-validation

Cross-validation can be used to more robustly estimate the generalization error of a model. We split the data into k-folds. For each fold, we train the model on the other folds and then we evaluate it on the chosen fold. We then average the resulting evaluation across folds and use this as our metric estimate. When one is using cross-validation to also select hyper-parameters, then we should do nested cross-validation (Fig. 11.7). Nested cross-validation works as follows.

Algorithm 11.1

def *CROSS-VALIDATION:*
  Divide your data $\mathcal{D}$ into K folds $\mathcal{D}_1, \ldots, \mathcal{D}_K$.
  for *$i = 1, 2, \ldots, K$:*
    Let $\mathcal{D}'_1, \ldots, \mathcal{D}'_{K-1}$ be all folds different from $\mathcal{D}_i$.
    for *$j = 1, 2, \ldots, K-1$:*
      Train a model with different parameter configurations using $\{\mathcal{D}'_\ell : \ell \neq j\}$.
      Evaluate the model on $\mathcal{D}'_j$ for each parameter configuration.
    end for
    Choose the parameter configuration that achieved best results in average.
    Train on $\mathcal{D}'_1, \ldots, \mathcal{D}'_{K-1}$ using the chosen configuration and evaluate the trained model on $\mathcal{D}_i$.
  end for

Observe that the inner train set is different across outer CV models, so the best performing parameters in one fold may be different from those in another fold. This enables the evaluation of the stability of the model. If your model is very stable with respect to slight changes or noise in the data, then you can expect similar parameters across rounds.

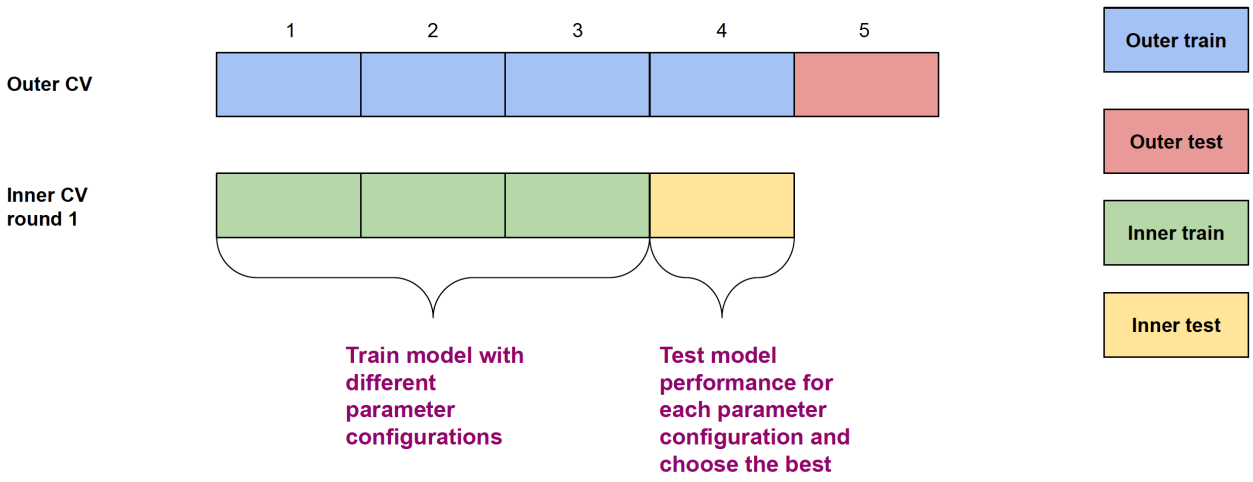


Figure 11.7: Nested cross validation

11.5.3 Statistical tests

Statistical tests can be used to determine whether the results we observe are statistically significant, e.g., whether the difference in the performance of two models is due to more than just chance. When performing statistical tests, it is important to understand both the probability of a type-I error and the statistical power under the alternative hypothesis (the probability of rejecting the null hypothesis when the alternative is true). A test may not have enough power to detect significant differences in performance.

McNemar's test

One such test for comparing classifiers is **McNemar's test**. Given two different classifiers A and B, the null hypothesis assumes that A and B have the same generalization error. Given some dataset on which we can evaluate our classifiers, we count the number n_0 of examples incorrectly classified by A but correctly classified by B and the number n_1 of examples incorrectly classified by B but correctly classified by A. The test statistic is then

$$\frac{(n_0 - n_1)^2}{n_0 + n_1} \tag{11.6}$$

and has a χ^2 distribution with 1 degree of freedom under the null hypothesis.

Permutation test

Permutation testing is a powerful tool that can be used to test the significance of observing almost any statistic, regardless of its underlying distribution. In a nutshell, a permutation test provides a simple method for constructing the sampling distribution of a test statistic through empirical observations. The method uses the value of this statistic over all possible rearrangements of the observed data points to represent the distribution of the test statistic under the null hypothesis. Using this distribution, we can then determine the probability of observing a value of the test statistic. For example, we can use a permutation test to assess whether a trained classifier performs as well as random chance. The p-value of this test is computed as follows:

Algorithm 11.2

def *PERMUTATIONTEST:*

Train a model on the dataset and evaluate its error p^* *on a holdout dataset.*

for $i = 1, 2, \ldots, K$*:*

Permute the labels in the dataset at random.

Retrain the model using the permuted labels.

Compute the error of the trained model p_k *on the permuted dataset.*

end for

The p-value of the test is given by $\frac{|\{i : p_i \leq p^*\}+1|}{K+1}$.

5×2cv paired t-test

This test compares the performance of two classifiers A and B based on some metric p. The test statistic is computed as follows:

Algorithm 11.3

def 5×2*CV-PAIRED T-TEST:*

for $i = 1, \ldots, 5$*:*

Split the dataset into 2 folds. Then for each fold j*, fit both classifiers on the train fold and calculate the difference in performances between* A *and* B *on the test fold* $p_i^{(j)} = p_A^{(j)} - p_B^{(j)}$.

Estimate the mean and variance of the performance difference across i*:*

$\bar{p}_i = \frac{p_i^{(1)}+p_i^{(2)}}{2}$.

$s_i^2 = \left(p_i^{(1)} - \bar{p}_i\right)^2 + \left(p_i^{(2)} - \bar{p}_i\right)^2$.

end for

$\tilde{t} = \frac{p_1^{(1)}}{\sqrt{\frac{1}{5}\sum_{i \leq 5} s_i^2}}$.

The null hypothesis assumes that A and B have the same performance. Under this hypothesis and some assumptions, the test has a t distribution with 5 degrees of freedom.

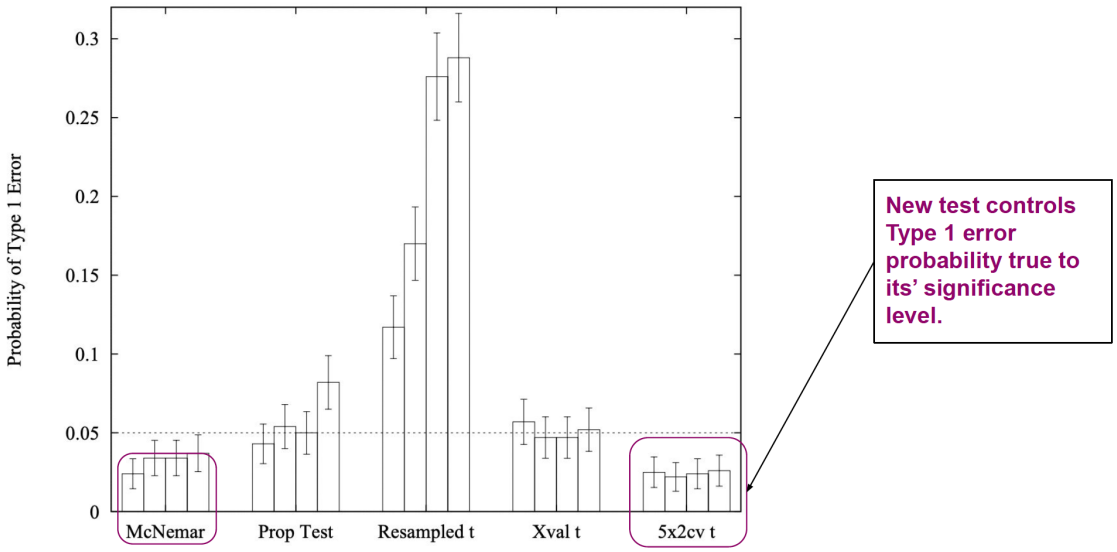


Figure 5 from [2]: McNemar's and 5x2cv t-test both have suitable Type 1 control, but 5x2 cv t-test has higher power (not shown).

Figure 11.8: Empirical comparison of the probability of a type I error for different statistical tests. McNemar's and 5x2cv t-test both have suitable Type 1 error, but 5x2 cv t-test has higher power (not shown).

Bibliography

[1] S. Kamal Abdali and B. David Saunders. "Transitive Closure and Related Semiring Properties via Eliminants". In: *Theor. Comput. Sci.* 40.2–3 (1985), pp. 257–274. ISSN: 0304-3975. URL: https://dl.acm.org/doi/10.5555/6400.6411.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL].

[3] F. Bauer. "Computational Graphs and Rounding Error". In: *SIAM Journal on Numerical Analysis* 11.1 (1974), pp. 87–96. DOI: 10.1137/0711010.

[4] Yoshua Bengio et al. "A Neural Probabilistic Language Model". In: *J. Mach. Learn. Res.* 3.null (Mar. 2003), pp. 1137–1155. ISSN: 1532-4435.

[5] S Chaiken and D.J Kleitman. "Matrix Tree Theorems". In: *Journal of Combinatorial Theory, Series A* 24.3 (1978), pp. 377–381. ISSN: 0097-3165. DOI: https://doi.org/10.1016/0097-3165(78)90067-5. URL: https://www.sciencedirect.com/science/article/pii/0097316578900675.

[6] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1724–1734. DOI: 10.3115/v1/D14-1179. URL: https://aclanthology.org/D14-1179.

[7] N. Chomsky. "Three models for the description of language". In: *IRE Transactions on Information Theory* 2.3 (1956), pp. 113–124. DOI: 10.1109/TIT.1956.1056813.

[8] Michael Collins. "Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms". In: *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing (EMNLP 2002)*. Association for Computational Linguistics, July 2002, pp. 1–8. DOI: 10.3115/1118693.1118694. URL: https://aclanthology.org/W02-1001.

[9] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[10] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals, and Systems (MCSS)* 2.4 (Dec. 1989), pp. 303–314. ISSN: 0932-4194. DOI: 10.1007/BF02551274. URL: http://dx.doi.org/10.1007/BF02551274.

[11] Jack Edmonds. "Optimum branchings". In: *Mathematics and the Decision Sciences, Part* 1.335-345 (1968), p. 25.

[12] Jacob Eisenstein. *Introduction to Natural Language Processing*. 2019. URL: https://mitpress.mit.edu/books/introduction-natural-language-processing.

[13] Jeffrey L. Elman. "Finding structure in time". In: *Cognitive Science* 14.2 (1990), pp. 179–211. ISSN: 0364-0213. DOI: https://doi.org/10.1016/0364-0213(90)90002-E. URL: https://www.sciencedirect.com/science/article/pii/036402139090002E.

[14] Joshua Goodman. "Semiring Parsing". In: *Computational Linguistics* 25.4 (1999), pp. 573–606. URL: https://aclanthology.org/J99-4004.

[15] Alex Graves, Greg Wayne, and Ivo Danihelka. *Neural Turing Machines*. 2014. arXiv: 1410.5401 [cs.NE].

[16] Andreas Griewank and Andrea Walther. *Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008.

[17] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[18] Liang Huang. "Advanced Dynamic Programming in Semiring and Hypergraph Frameworks". In: *Coling 2008: Advanced Dynamic Programming in Computational Linguistics: Theory, Algorithms and Applications - Tutorial notes*. Manchester, UK: Coling 2008 Organizing Committee, Aug. 2008, pp. 1–18. URL: https://aclanthology.org/C08-5001.

[19] Liang Huang. "Advanced Dynamic Programming in Semiring and Hypergraph Frameworks". In: *Coling 2008: Advanced Dynamic Programming in Computational Linguistics: Theory, Algorithms and Applications - Tutorial notes*. 2008, pp. 1–18. URL: https://aclanthology.org/C08-5001.

[20] Mohit Iyyer et al. "Deep Unordered Composition Rivals Syntactic Methods for Text Classification". In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Beijing, China: Association for Computational Linguistics, July 2015, pp. 1681–1691. DOI: 10.3115/v1/P15-1162. URL: https://aclanthology.org/P15-1162.

[21] S. Katz. "Estimation of probabilities from sparse data for the language model component of a speech recognizer". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 35.3 (1987), pp. 400–401. DOI: 10.1109/TASSP.1987.1165125.

[22] Terry Koo et al. "Structured Prediction Models via the Matrix-Tree Theorem". In: *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. Prague, Czech Republic: Association for Computational Linguistics, June 2007, pp. 141–150. URL: https://aclanthology.org/D07-1015.

[23] Daniel J. Lehmann. "Algebraic structures for transitive closure". In: *Theoretical Computer Science* 4.1 (1977), pp. 59–76. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(77)90056-1. URL: https://www.sciencedirect.com/science/article/pii/0304397577900561.

[24] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. *Effective Approaches to Attention-based Neural Machine Translation*. 2015. arXiv: 1508.04025 [cs.CL].

[25] Mehryar Mohri, Fernando Pereira, and Michael Riley. *Speech Recognition with Weighted Finite-State Transducers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 559–584. URL: https://doi.org/10.1007/978-3-540-49127-9_28.

[26] Bo Pang and Lillian Lee. "Opinion Mining and Sentiment Analysis". In: *Foundations and Trends® in Information Retrieval* 2.1–2 (2008), pp. 1–135. ISSN: 1554-0669. DOI: 10.1561/1500000011. URL: http://dx.doi.org/10.1561/1500000011.

[27] Robert Endre Tarjan. "Finding optimum branchings". In: *Networks* 7.1 (1977), pp. 25–35.

[28] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[29] P.J. Werbos. "Backpropagation through time: what it does and how to do it". In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560. DOI: 10.1109/5.58337.

[30] Ran Zmigrod, Tim Vieira, and Ryan Cotterell. “Please Mind the Root: Decoding Arborescences for Dependency Parsing”. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Online: Association for Computational Linguistics, Nov. 2020, pp. 4809–4819. DOI: 10.18653/v1/2020.emnlp-main.390. URL: https://aclanthology.org/2020.emnlp-main.390.