Natural Language Processing

Cristian Perez Jensen

January 21, 2024

Contents

SUMMARY OF NOTATION

$\doteq$	equality by definition
$\approx$	approximate equality
$\propto$	proportial to
$\mathbb{N}$	set of natural numbers
$\mathbb{R}$	set of real numbers
$[m]$	set of natural numbers from 1 to m, $\{1, 2, \ldots, m\}$
$i : j$	set of natural numbers between i and j, $\{i, i+1, \ldots, j\}$
$f : A \to B$	function f that maps elements of set A to elements of set B
$\mathbb{1}\{\text{predicate}\}$	indicator function (1 if predicate is true, otherwise 0)

LINEAR ALGEBRA

$\boldsymbol{v} \in \mathbb{R}^n$	vector
$\boldsymbol{M} \in \mathbb{R}^{m \times n}$	matrix
$\mathbf{T} \in \mathbb{R}^{d_1 \times \cdots \times d_n}$	tensor
$\boldsymbol{M}^\top$	transpose of matrix $\boldsymbol{M}$
$\boldsymbol{M}^{-1}$	inverse of matrix $\boldsymbol{M}$
$\det(\boldsymbol{M})$	determinant of $\boldsymbol{M}$

CALCULUS

$\frac{\mathrm{d}}{\mathrm{d}x} f(x)$	ordinary derivative of $f(x)$ w.r.t. x
$\frac{\partial}{\partial x} f(\boldsymbol{x})$	partial derivative of $f(\boldsymbol{x})$ w.r.t. x
$\boldsymbol{\nabla} f(\boldsymbol{x}) \in \mathbb{R}^n$	gradient of $f : \mathbb{R}^n \to \mathbb{R}$ at point $\boldsymbol{x} \in \mathbb{R}^n$

MACHINE LEARNING

$\boldsymbol{\theta} \in \Theta$	parameterization of a model
$\mathcal{X}$	input space
$\mathcal{Y}$	output space
$\mathcal{D} \subseteq \mathcal{X} \times \mathcal{Y}$	labeled training data

1 Backpropagation

Backpropagation is the single most important algorithm in modern machine learning, because it is used to compute gradients of composite functions efficiently. It is a linear-time dynamic program for computing derivatives, *i.e.*, it stores intermediate results to be as fast as forward propagation.

In machine learning, most of the time, we have inputs $\boldsymbol{x} \in \mathcal{X}$ and outputs $\boldsymbol{y} \in \mathcal{Y}$ from a dataset $\mathcal{D} \subseteq \mathcal{X} \times \mathcal{Y}$, and we want to fit some function $f_{\boldsymbol{\theta}} : \mathcal{X} \to \mathcal{Y}$ such that it minimizes some loss function,

$$\sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{D}} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y}).$$

We need this function's gradient $\frac{\partial}{\partial\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\boldsymbol{x})$ to be able to use an algorithm such as gradient descent to optimize it.[1] For a composite function, it is time consuming to derive the gradient by hand. Thus, we use backpropagation to automatically compute the gradients, as long as we have access to the derivatives of its primitive functions.

[1] Most of the time, this function cannot be solved in closed form.

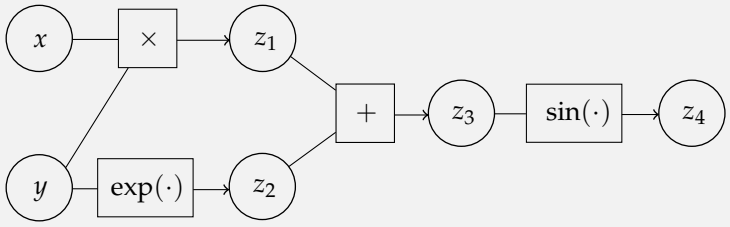
Example 1. Computation graph.

A composite function can be represented using intermediate variables such that each variable is computed by a single primitive function. Let's say we have the following function,

$$f(x, y) = \sin(xy + \exp(y)).$$

Then, we can represent the intermediate variables as the following,

$$\begin{aligned} z_1 &= xy \\ z_2 &= \exp(y) \\ z_3 &= z_1 + z_2 \\ z_4 &= \sin(z_3). \end{aligned}$$



We could also describe a function as a labeled, directed acyclic[2]

[2] The fact that our computation graph is acyclic makes it possible for backpropagation to be linear.

hypergraph[3], where each node is a variable and each hyperedge is labeled with a function. Given a function $f : \mathbb{R}^n \to \mathbb{R}^m$, $\boldsymbol{y} = f(\boldsymbol{x}) \in \mathbb{R}^m$, for input x_j and outputs y_i, Bauer's formula gives

$$\frac{\partial y_i}{\partial x_j} = \sum_{p \in P(j,i)} \prod_{(k,l) \in p} \frac{\partial z_l}{\partial z_k}, \tag{1}$$

where $P(j,i)$ is the set of paths from vertex j to vertex i and $p \in P(j,i)$ is the set of edges that make up the path p. *I.e.*, the partial derivative is a sum over all paths in the computation graph, where the derivative over each path is computed by the chain rule.[4] When computing the partial derivatives for functions with dense computation graphs naively, we are typically summing over an exponential number of paths, because many of these partial derivatives are recomputed many times. We can use dynamic programming to store these values and avoid computing them again. In this case, the amount of computation scales linearly with the number of edges.

[3] A hypergraph allows the edges to have multiple sources and targets. This is needed because functions can have multiple inputs and multiple outputs.

[4] $\frac{\mathrm{d}}{\mathrm{d}x} f(g(x)) = \frac{\mathrm{d}}{\mathrm{d}g(x)} f(g(x)) \cdot \frac{\mathrm{d}}{\mathrm{d}x} g(x)$.

function ForwardPropagation($f, \boldsymbol{x}$)
  $z_i \leftarrow \begin{cases} x_i & \textbf{if } i \leq m \\ 0 & \textbf{otherwise} \end{cases}$ ▷ *Initialize input variables*
  for $i = m+1, \ldots, n$ **do**
    $z_i \leftarrow g_i(\boldsymbol{z}_{\text{Parents}(i)})$ ▷ *Set intermediate variables*
  return $\boldsymbol{z}$

Algorithm 1. Forward propagation algorithm that assumes that the edges are topologically sorted so $i < j$ implies that z_i is computed before z_j.

function Backpropagation($f, \boldsymbol{x}$)
  $\boldsymbol{z} \leftarrow$ ForwardPropagation($f, \boldsymbol{x}$)
  $\frac{\partial f}{\partial z_i} \leftarrow \begin{cases} 1 & \textbf{if } i = n \\ 0 & \textbf{otherwise} \end{cases}$ ▷ *Base case*
  for $i = n-1, \ldots, 1$ **do** ▷ $O(n)$
    $\frac{\partial f}{\partial z_i} \leftarrow \sum_{z_p \in \text{Parents}(z_i)} \frac{\partial f}{\partial z_p} \frac{\partial z_p}{\partial z_i}$ ▷ *Chain rule*
  return $\nabla_{\boldsymbol{x}} f$

Algorithm 2. Backpropagation algorithm that assumes that the edges are topologically sorted so $i < j$ implies that z_i is computed before z_j.

The general framework that any backpropagation framework uses is the following:

1. Write down a composite function as a hypergraph with intermediate variables as nodes and hyperedges labeled with primitive functions;
2. Given a set of inputs, perform forward propagation through the graph to compute the function's value (Algorithm 1);
3. Run backpropagation on the graph using the stored forward values (Algorithm 2). Intuitively, we set up a dynamic programming table using Equation (1).

$$\frac{\partial f}{\partial z_4} = 1$$
$$\frac{\partial f}{\partial z_3} = \frac{\partial f}{\partial z_4}\frac{\partial z_4}{\partial z_3} = \cos(z_3)$$
$$\frac{\partial f}{\partial z_2} = \frac{\partial f}{\partial z_3}\frac{\partial z_3}{\partial z_2} = \cos(z_3)$$
$$\frac{\partial f}{\partial z_1} = \frac{\partial f}{\partial z_3}\frac{\partial z_3}{\partial z_1} = \cos(z_3)$$
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial x} = \cos(z_3)y$$
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial y} + \frac{\partial f}{\partial z_2}\frac{\partial z_2}{\partial z_1} = \cos(z_3)x + \cos(z_3)\exp(y).$$

Example 2. Backpropagation computation for the function in Example 1.

2 *Log-linear modelling*

Let's say we want to model the conditional probability $p(y \mid x)$. A naive way of doing this is the following,

$$p(y \mid x) \doteq \frac{\text{count}(x, y)}{\text{count}(x)}.$$

There are two main problems with this interpretation of discrete conditional probability,

- Suppose $\text{count}(x, y) = 0$, then the probability will be 0, *i.e.*, the model says that y is impossible in context x;
- There is no way to look at finer-grained aspects of x, *i.e.*, some values of x might be related.

Thus, we need a more general framework for modelling conditional distributions. One such general framework is to simply exponentiate some scoring function that we construct,[5] and let the conditional probability be proportional to it,

$$p(y \mid x) \propto \exp \text{score}(x, y) > 0.$$

[5] The exponentiation makes sure that it is non-negative.

The linear scoring function looks like the following,

$$\text{score}(x, y) = \boldsymbol{\theta}^\top \boldsymbol{f}(x, y) > 0$$

with feature weights $\boldsymbol{\theta} \in \mathbb{R}^K$ and $\boldsymbol{f}(x, y) \in \mathbb{R}^K$ as a vector describing y in context x. The conditional probability then looks like the following,

$$p_{\boldsymbol{\theta}}(y \mid x) = \frac{1}{Z_{\boldsymbol{\theta}}} \exp\Big(\boldsymbol{\theta}^\top \boldsymbol{f}(x, y)\Big)$$
$$Z_{\boldsymbol{\theta}}(x) = \sum_{y' \in \mathcal{Y}} \exp\Big(\boldsymbol{\theta}^\top \boldsymbol{f}(x, y')\Big).$$

This is called log-linear modelling, because if we take the logarithm of the conditional probability, we get a linear model,

$$\log p_{\boldsymbol{\theta}}(y \mid x) = \boldsymbol{\theta}^\top \boldsymbol{f}(x, y) - \log Z_{\boldsymbol{\theta}}(x).$$

The design of the *feature function* $\boldsymbol{f}(x, y)$ is a big portion of the work in log-linear modelling. It can be split into two parts: preprocessing and extracting features. The preprocessing simply consists of steps such as tokenization, lower-casing, stemming, stop-word removal, and reducing vocabulary. After the preprocessing, we can obtain features. Examples include one-hot encoding, bag-of-words, n-grams, and word embeddings.

Maximum likelihood estimation (MLE) is a way of finding the parameters $\boldsymbol{\theta} \in \Theta$ that minimize the *negative log-likelihood* of the training data, *i.e.*, we want to minimize the following,

$$\hat{\boldsymbol{\theta}} = \operatorname*{argmin}_{\boldsymbol{\theta} \in \Theta} \sum_{(x,y) \in \mathcal{D}} -\log p_{\boldsymbol{\theta}}(y \mid x),$$

where Θ is a compact (bounded and closed) subset of $\mathbb{R}^K$.[6] In log-linear modelling, this objective function is convex, thus any local minimum is a global minimum. We usually optimize the log-likelihood with gradient-based methods,

[6] The reason for not taking $\boldsymbol{\theta} \in \mathbb{R}^K$ is that the weights will likely go to infinity.

$$\begin{aligned}
\frac{\partial}{\partial \boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(y \mid x) &= \frac{\partial}{\partial \boldsymbol{\theta}} \boldsymbol{\theta}^\top \boldsymbol{f}(x, y) - \log Z_{\boldsymbol{\theta}}(x) \\
&= \boldsymbol{f}(x, y) - \frac{\partial}{\partial \boldsymbol{\theta}} \log Z_{\boldsymbol{\theta}}(x) \\
&= \boldsymbol{f}(x, y) - \frac{1}{Z_{\boldsymbol{\theta}}(x)} \sum_{y' \in \mathcal{Y}} \frac{\partial}{\partial \boldsymbol{\theta}} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(x, y')\right) \\
&= \boldsymbol{f}(x, y) - \sum_{y' \in \mathcal{Y}} \frac{1}{Z_{\boldsymbol{\theta}}(x)} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(x, y')\right) \frac{\partial}{\partial \boldsymbol{\theta}} \boldsymbol{\theta}^\top \boldsymbol{f}(x, y') \\
&= \boldsymbol{f}(x, y) - \sum_{y' \in \mathcal{Y}} p_{\boldsymbol{\theta}}(y' \mid x) \boldsymbol{f}(x, y') \\
&= \boldsymbol{f}(x, y) - \sum_{y' \in \mathcal{Y}} p_{\boldsymbol{\theta}}(y' \mid x) \boldsymbol{f}(x, y').
\end{aligned}$$

Due to convexity, the global minimum is the only point that has its gradient equal 0. Thus, at the optimal parameters, the following is the case,

$$\boldsymbol{f}(x, y) = \sum_{y' \in \mathcal{Y}} p_{\boldsymbol{\theta}}(y' \mid x) \boldsymbol{f}(x, y').$$

Therefore, the optimum is where the observed feature counts $\boldsymbol{f}(x, y)$ look like the expected feature counts $\sum_{y' \in \mathcal{Y}} p_{\boldsymbol{\theta}}(y' \mid x) \boldsymbol{f}(x, y')$. In other words, the training data looks exactly like what our model predicts through the eyes of our feature function $\boldsymbol{f}$. This is referred to as *expectation matching*.

2.1 Softmax

The softmax : $\mathbb{R}^K \to \Delta^{K-1}$ function is the default way of building probabilistic models using neural networks, because it maps vectors to categorical probability distributions. It is basically the same as log-linear modelling. It is defined as

$$\text{softmax}(\boldsymbol{h})_y \doteq \frac{\exp(h_y/T)}{\sum_{y' \in \mathcal{Y}} \exp(h_{y'}/T)}.$$

The probability simplex Δ^{K-1} is a subspace of $\mathbb{R}^K_{\geq 0}$ such that the sum of the components of its elements is 1. It is denoted as Δ^{K-1}, because it has $K-1$ free parameters, and looks like a triangle in three dimensions.

Usually, the temperature is set to $T = 1$.[7] This is a generalization of log-linear modelling, where, instead of $\boldsymbol{\theta}^\top \boldsymbol{f}(x, y)$, we can use any function of the input.

[7] As $T \to 0$, softmax becomes the argmax function and as $T \to \infty$, softmax becomes a uniform categorical distribution.

3 Multi-layer perceptrons

For log-linear models to find an appropriate model, the data has to be linearly separable.[8] The solution to this problem is the *multi-layer perceptron* (MLP) [Haykin, 1994]. They jointly learn a non-linear feature function with the model's parameters. MLPs consist of n alternating linear projections and non-linearities,[9]

$$\boldsymbol{h}_n = \sigma_n\Big(\boldsymbol{W}_n^\top \boldsymbol{h}_{n-1}\Big)$$
$$\boldsymbol{h}_1 = \sigma_1\Big(\boldsymbol{W}_1^\top \boldsymbol{e}(x)\Big),$$

where $\boldsymbol{e}(x) \in \mathbb{R}^d$ is a continuous vector encoding of the input x. Then, we can combine this non-linear feature representation of the input with the parameters to obtain a categorical probability distribution,

$$\text{softmax}\Big(\boldsymbol{\theta}^\top \boldsymbol{h}_n\Big).$$

Basically, the only difference is that we now learn the feature function.

[8] If the data is non-linear, we can use a feature function $\boldsymbol{f}$ that makes the feature vector linearly separable by hacking the non-linearity into the feature function (*e.g.*, if the data is ellipsoidal, we would choose an ellipsoidal feature function.) But, this requires us to know the decision boundary's shape a priori to set the feature function.

[9] The non-linearities are very important, because otherwise we would just get a linear model, since a stack of linear transformations are equal to some single linear transformation. Thus, if the model would not contain non-linearities, we would not gain any expressiveness.

3.1 Word embeddings

To be able to use MLPs, we would need a continuous vector encoding of words/sentences. A naive idea would be to encode one-hot word counts of sentences. However, this approach discards word ordering information. A naive solution would be to then encode n-grams to regain some word ordering information, however the vectors would explode in size to $\mathcal{O}(|V|^n)$, where $|V|$ is the vocabulary size.

A great idea is to use unsupervised learning to train word embeddings on large corpora.[10] The first model that made use of this idea is Skip-Gram [Mikolov et al., 2013]. Its high-level idea is that it predicts whether a word w' is in the context of a word w.[11] The weights of this model are the word embeddings and they are used to predict

[10] Easy to get due to the internet.

[11] "You shall know a word by the company it keeps."

the model's objective. The idea is then that the model needs a good representation of the words to be able to do this task successfully, thus we can use the embeddings that this model trains for other tasks.

The first step of Skip-Gram is to preprocess the corpus. This is done by collecting positive and negative samples.[12] To collect positive samples, it goes through all words w and collects all words w' in the context of w. Then, it randomly generates other words w' as negative samples, *i.e.*, they are not in the context of w, and adds them to the dataset.

[12] A reason for collecting negative samples is because then we do not need to compute the normalizing constant $Z(w)$. This is due to the fact that we are simply maximizing the output for positive samples and minimizing the output for negative samples while training. But, if we did not have negative samples, we would have to normalize the output to be a probability between 0 and 1, because otherwise we would not know whether the output is good or not.

Then, we use the embeddings to predict whether a context word is in a word's context,

$$p(c \mid w) = \frac{1}{Z(w)} \exp\Big(\boldsymbol{e}_w(w)^\top \boldsymbol{e}_c(c)\Big)$$
$$Z(w) = \sum_{c \in V} \exp\Big(\boldsymbol{e}_w(w)^\top \boldsymbol{e}_c(c)\Big).$$

We use two different embedding types, because if $w = c$, then $\boldsymbol{e}(w)^\top \boldsymbol{e}(c)$ would be positive and thus the probability very high, even though this case is very unlikely, because words do usually not appear in their own context. Thus, to mitigate this, we use two different context and word embeddings.

3.2 *Sentiment analysis*

An example application is sentiment analysis [Pang et al., 2008], which classifies sentences as positive or negative. MLPs and word embeddings can be used for this and work very well [Iyyer et al., 2015]. The model first embeds all the words in the sentence and pools them together into one vector representation of the sentence.[13] Then, this vector is passed into an MLP, which is used as input to a softmax with 2 outputs (positive or negative).

[13] Pooling together discards word order, but for this simple task, it will still work very well.

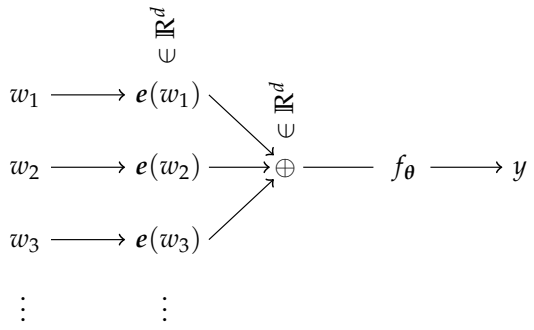


Figure 1. Architecture of a simple sentiment classifier.

4 *Structured prediction*

In NLP, we often have the case that the input and output of a model has some structure. *E.g.*, in context-free parsing, we have a sentence as input and want to output a parse tree. To be able to train a model and perform inference, we need to be able to compute the log-likelihood,

$$p_{\boldsymbol{\theta}}(y \mid x) \doteq \frac{1}{Z_{\boldsymbol{\theta}}(x)} \exp(\text{score}_{\boldsymbol{\theta}}(y, x))$$
$$Z_{\boldsymbol{\theta}}(x) \doteq \sum_{y' \in \mathcal{Y}} \exp(\text{score}_{\boldsymbol{\theta}}(y', x)).$$

However, often there are an exponential, or even infinite, amount of possible structures $y \in \mathcal{Y}$ for an input $x \in \mathcal{X}$. For training, this has the result that the normalizer $Z_{\boldsymbol{\theta}}$ is very inefficient to compute, since it is a sum over a very large amount of values. For inference, this has the result that we would need to search a very large space for the best output y.

The solution to this problem is that we need to make use of the structure to design algorithms for computing the normalizer and finding the most probable output. The next sections will include NLP problems, where we need to design algorithms to compute the normalizer efficiently.

5 *Language modelling*

In language modelling, Σ is a finite, non-empty set of symbols. In the context of natural language, this is usually set to be the vocabulary of words or tokens. A string over an alphabet Σ is any finite sequence of alphabet symbols. The output space $\mathcal{Y}$ is set to the set of all possible strings Σ^*, which is infinite, thus there is no way to sum over every possible structure, nor is there an easy way to output the maximum scoring structure.

A *language model* (LM) is a probability distribution over Σ^*, *i.e.*, LMs assign probabilities to strings $\boldsymbol{y} \in \Sigma^*$. We can discriminate between two types of LMs,

- *Globally normalized* LMs, which define a single scoring function $\text{score}_{\boldsymbol{\theta}} : \Sigma^* \to \mathbb{R}$ and normalizes the scores across all $\boldsymbol{y} \in \Sigma^*$,[14]

$$p(\boldsymbol{y}) \doteq \frac{1}{Z_{\boldsymbol{\theta}}} \exp \text{score}_{\boldsymbol{\theta}}(\boldsymbol{y});$$

- *Locally normalized* LMs, which decompose string probabilities into conditional probabilities $p(y_i \mid \boldsymbol{y}_{<i})$ over symbols y_i given the pre-

[14] Global LMs are not used much, because their normalizer requires a sum over an infinite set. Thus, we will focus on local LMs.

vious symbols $\boldsymbol{y}_{<i}$,

$$p(\boldsymbol{y}) \doteq p(\text{EOS} \mid \boldsymbol{y}) \cdot \prod_{i=1}^{N} p(y_i \mid \boldsymbol{y}_{<i})$$
$$p(y_i \mid \boldsymbol{y}_{<i}) \doteq \frac{1}{Z_{\boldsymbol{\theta}}(\boldsymbol{y}_{<i})} \exp \text{score}_{\boldsymbol{\theta}}(y_i, \boldsymbol{y}_{<i}).$$

Local LMs are collections of conditional probability distributions $p(y_i \mid \boldsymbol{y}_{<i})$, which intuitively tells us how probable symbol y_i is to follow the already seen string $\boldsymbol{y}_{<i}$. In practice, we also need beginning-of-sentence (BOS) and end-of-sentence (EOS) symbols. We condition on BOS for the first token to model that the sequence starts with y_1 and we condition on the entire string with EOS to model the probability that no more symbols follow.

A well-defined LM always assigns probability to EOS given any history $\boldsymbol{y}_{<i}$, because otherwise we could get in the situation where we have a history that never ends in EOS, which will not result in a string. Models that have this problem are called non-tight. The probability of all sentences in a non-tight model do not sum to 1. To mitigate this problem, we ensure a model is tight by forcing $p(\text{EOS} \mid \boldsymbol{y}_{<i}) > \xi > 0$ for every history $\boldsymbol{y}_{<i}$.

The problem with local LMs is that there are infinitely many distributions $p(\cdot \mid \boldsymbol{y}_{<i})$ with $\boldsymbol{y}_{<i} \in \Sigma^*$. Thus, we need some way of being able to compute all these distributions.

We can also use local LMs to generate strings by continuously sampling from the probability distribution $p(y_i \mid \boldsymbol{y}_{<i})$ until we sample EOS, *e.g.*,

$$\text{HE WALKS THE} \begin{cases} p(\text{DOG} \mid \text{HE} \cdots) = 0.5 \\ p(\text{CAT} \mid \text{HE} \cdots) = 0.24 \\ p(\text{EOS} \mid \text{HE} \cdots) = 0.01. \end{cases}$$

5.1 *n-grams*

The n-gram solution is to limit histories $\boldsymbol{y}_{<i}$ to a length $n-1$. This assumption leads to the following probability distribution,

$$p(y_i \mid \boldsymbol{y}_{<i}) = p(y_i \mid y_{i-n+1}, \ldots, y_{i-1}).$$

This ensures a finite number $|\Sigma|^{n-1}$ of histories.

The naive implementation of n-gram would be to define a separate conditional probability distribution for every possible context of size $n-1$. Thus, we can assign each history a probability distribution based on the counts we observe in training data,

$$p(y_i \mid y_{i-n+1}, \ldots, y_{i-1}) = \frac{\text{count}(y_{i-n+1}, \ldots, y_{i-1}, y_i)}{\text{count}(y_{i-n+1}, \ldots, y_{i-1})}.$$

However, this does not allow us to share parameters between histories, which might be very similar and give much insight.[15] Furthermore, it is very memory inefficient, since we need to store $(|\bar{\Sigma}| - 1)\,|\Sigma|^{n-1}$

[15] *E.g.*, the distributions over

$$p(\cdot \mid \text{SHE WALKS}) \quad p(\cdot \mid \text{HE WALKS})$$

are parametrized independently, even though their distributions should be extremely similar.

parameters to specify the $|\Sigma|^{n-1}$ conditional distributions for each history, where $\bar{\Sigma} = \Sigma \cup \{\text{EOS}\}$.

The neural n-gram [Bengio et al., 2000] is a more efficient approach that does allow parameter sharing. It uses word embeddings to encode the words and histories,

$$p(y_i \mid \boldsymbol{y}_{<i}) = \frac{\exp\left(\boldsymbol{e}(y_i)^\top \boldsymbol{h}_i\right)}{\sum_{y' \in \bar{\Sigma}} \exp\left(\boldsymbol{e}(y')^\top \boldsymbol{h}_i\right)},$$

where $\boldsymbol{h}_i = \text{enc}(\boldsymbol{y}_{<i}) = \text{enc}(y_{i-n+1}, \ldots, y_{i-1})$.[16] The crucial idea of this approach is that if the word embedding is similar to the encoding of the history, it is more likely.

[16] Bengio et al. [2000] used a neural network to encode the history. In this architecture, the words in the history $\boldsymbol{y}_{i-n+1:i-1}$ are concatenated (preserve word-order) and used as input to a neural network that outputs a d-dimensional representation $\boldsymbol{h}_i$.

5.2 *Recurrent neural networks*

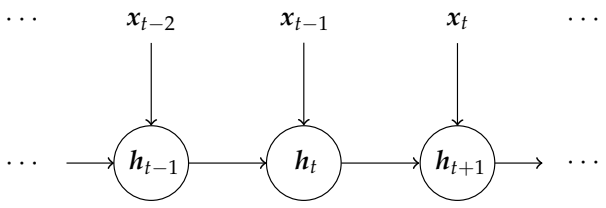


Figure 2. Diagram of the RNN architecture. Each hidden state $\boldsymbol{h}_i$ has "seen" all previous tokens $\boldsymbol{x}_{1:i-1}$. TODO: Update t to be i.

The fixed history size of the n-gram model is not realistic. We want to be able to encode the entire history, which is possible with *recurrent neural networks* (RNN). RNNs work by having two inputs at every timestep: the time-dependent hidden state $\boldsymbol{h}_{i-1}$, representing all words before the current one, and embedding of the current token $\boldsymbol{e}(y_{i-1})$. These are combined to represent the entire history. There are many variations, but the most simple one is the Elman RNN [Elman, 1990],

$$\boldsymbol{h}_i = \sigma(\boldsymbol{W}_h \boldsymbol{h}_{i-1} + \boldsymbol{W}_x \boldsymbol{e}(y_{i-1}) + \boldsymbol{b}),$$

where $\boldsymbol{W}_h \in \mathbb{R}^{d \times d}$, $\boldsymbol{W}_x \in \mathbb{R}^{d \times d}$, and $\boldsymbol{b} \in \mathbb{R}^d$ are learned parameters. At their core, RNNs are just a non-linear combination of the recurrent state and the inputs.

RNNs are trained by unrolling the timesteps and applying backpropagation. The problem with this is that RNNs become prone to the vanishing gradient problem, because computing the gradient of the parameters involves a lot of matrix operations that usually have values between -1 and 1. The LSTM [Hochreiter and Schmidhuber, 1997] and GRU [Cho et al., 2014] architectures seek to solve this problem.

6 Semirings

Definition 1 (Monoid)**.** A monoid is a 3-tuple $\langle \mathbb{K}, \odot, \boldsymbol{e} \rangle$, such that

1. $\odot$ is associative for all values in $\mathbb{K}$. $\forall x, y, z \in \mathbb{K}$,

$$(x \odot y) \odot z = x \odot (y \odot z).$$

2. $\boldsymbol{e} \in \mathbb{K}$ is the identity element. $\forall x \in \mathbb{K}$,

$$x \odot \boldsymbol{e} = x.$$

Definition 2 (Semiring)**.** A semiring is a 5-tuple $\langle \mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1} \rangle$, such that

1. $\langle \mathbb{K}, \oplus, \mathbf{0} \rangle$ is a commutative monoid.
2. $\langle \mathbb{K}, \otimes, \mathbf{1} \rangle$ is a monoid.
3. $\otimes$ distributes over $\oplus$. $\forall x, y, z \in \mathbb{K}$,

$$(x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z)$$
$$z \otimes (x \oplus y) = (z \otimes x) \oplus (z \otimes y).$$

4. $\mathbf{0}$ is an annihilator for $\otimes$. $\forall x \in \mathbb{K}$,

$$\mathbf{0} \otimes x = \mathbf{0}$$
$$x \otimes \mathbf{0} = \mathbf{0}.$$

Semirings are very useful for generalizing algorithms that only make use of associativity, commutativity, and distributivity. For example, if we have an efficient algorithm for computing the following,

$$Z = \sum_{\boldsymbol{y} \in \mathcal{Y}} \prod_{n=1}^{N} \exp \text{score}(y_n),$$

which uses the real semiring, we can semiringify it to compute the following,

$$\bigoplus_{\boldsymbol{y} \in \mathcal{Y}} \bigotimes_{n=1}^{N} \exp \text{score}(y_n).$$

Then, we can use any semiring with this algorithm. For inference, we would then want to use for example the Viterbi semiring $\langle [0, 1], \max, \times, 0, 1 \rangle$ to compute the following,

$$\max_{\boldsymbol{y} \in \mathcal{Y}} \prod_{n=1}^{N} \exp \text{score}(y_n).$$

6.1 Closed semirings

Definition 3 (Closed semiring)**.** A closed semiring is a semiring with an additional unary operation: the Kleene star,

$$x^* = \bigoplus_{n=0}^{\infty} x^{\otimes n}.$$

The Kleene star must obey the following two axioms,

$$x^* = \mathbf{1} \oplus x \otimes x^*$$
$$x^* = \mathbf{1} \oplus x^* \otimes x.$$

Closedness allows us to compute infinite sums within a semiring. For example, the real semiring is closed if we let its set be $(-1, 1)$, because then we can compute the Kleene star to be the following using the closed form of the geometric series,

$$x^* = \sum_{n=0}^{\infty} x^n = \frac{1}{1-x}.$$

7 Part-of-speech tagging

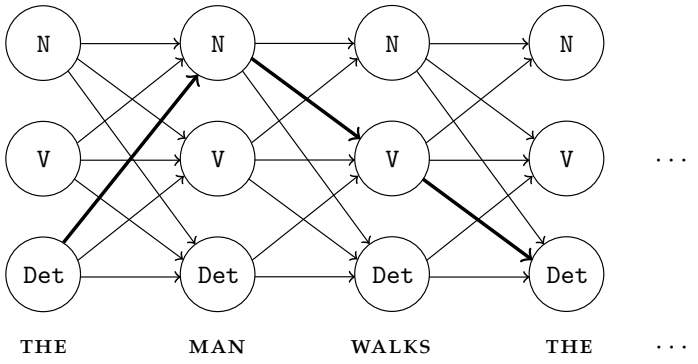


Figure 3. Example POS graph with $\mathcal{T} = \{\texttt{N}, \texttt{V}, \texttt{Det}\}$. For inference, we want to find the best N-length path within this graph. For training, we want to compute the sum over all N-length paths within this graph.

In *part-of-speech tagging* (POS tagging), we want to predict a POS tag $t \in \mathcal{T}$ for every word of the input sentence $\boldsymbol{w}$ of length N, *i.e.*, given an N-dimensional input sequence of words $\boldsymbol{w} \in \Sigma^N$, we want to output an N-dimensional sequence of tags $\boldsymbol{t} \in \mathcal{T}^N$. This can be seen as searching through a POS graph as in Figure 3. The output space $\mathcal{T}^N$ is exponential, so we need to design an algorithm to efficiently compute the normalizer $Z_{\boldsymbol{\theta}}$, and find the maximum scoring tagging.

7.1 Conditional random fields

Conditional random fields (CRF) are a conditional probabilistic model for structured labelling. Whereas a classifier predicts a label for a single sample without considering neighboring samples in the structure, a CRF does take context into account. In other words, CRFs are a model for computing the normalizer Z in a structured labelling case.

In the sequence labelling case of POS tagging, we will assume that tags only depend on their immediate neighbors,

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

which can be further decomposed into transition and emission scores,

$$\text{score}(\langle t_{n-1}, t_n \rangle, \boldsymbol{w}, n) = \text{transition}(\langle t_{n-1}, t_n \rangle) + \text{emission}(w_n, t_n).$$

This balances how likely t_n is to follow t_{n-1} and how likely word w_n is to be assigned tag t_n.

Note that our bigram assumption does not mean that the current tag only depends on the previous and current word, because the word representations can be anything. *E.g.*, if we use a bidirectional RNN for the word representations, the tags will still depend on the entire input sentence. However, a problem that this can cause is that it cannot correctly tag garden-path sentences, since we cannot change the start of the tagging after seeing the end of the tagging.[17]

[17] An example garden-path sentence is "The horse raced past the barn fell."

We can use the new decomposed scoring function to find an efficient algorithm for computing the normalizer under a semiring,

$$\begin{aligned}
&\bigoplus_{\boldsymbol{t} \in \mathcal{T}^N} \bigotimes_{n=1}^{N} \exp \text{score}(\langle t_{n-1}, t_n \rangle, \boldsymbol{w}, n) \\
&= \bigoplus_{t_1 \in \mathcal{T}} \cdots \bigoplus_{t_n \in \mathcal{T}} \exp \text{score}(\langle t_0, t_1 \rangle, \boldsymbol{w}, 1) \otimes \cdots \otimes \exp \text{score}(\langle t_{N-1}, t_N \rangle, \boldsymbol{w}, N) \\
&= \bigoplus_{t_1 \in \mathcal{T}} \exp \text{score}(\langle t_0, t_1 \rangle, \boldsymbol{w}, 1) \otimes \left(\cdots \otimes \left(\bigoplus_{t_n \in \mathcal{T}} \exp \text{score}(\langle t_{N-1}, t_N \rangle, \boldsymbol{w}, N) \right) \right),
\end{aligned}$$

where distributivity is used in the last step. The backward and forward algorithms (Algorithms 3 and 4) are direct results of this rederivation, which compute the normalizer in $\mathcal{O}(N \cdot |\mathcal{T}|^2)$.

Under the real semiring, we can thus use these algorithms to compute the normalizer Z during training. For inference, the Viterbi algorithm is a version of the backward algorithm under the Viterbi semiring. Furthermore, the Viterbi algorithm keeps track of backpoints to find the maximally scoring tagging.

```
function BackwardAlgorithm(w, score, ⟨A, ⊕, ⊗, 0, 1⟩)
    β[N] ← 1
    for n = N − 1, ..., 0 do
        for t_n ∈ T do
            β[n, t_n] ← ⊕_{t_{n+1} ∈ T} exp(score(⟨t_n, t_{n+1}⟩), w, n + 1) ⊗ β[n + 1, t_{n+1}]
    return β[0, BOT]
```

Algorithm 3. Backward algorithm that computes the semiring-sum over all taggings of a sentence $\boldsymbol{w}$.

```
function ForwardAlgorithm(w, score, ⟨A, ⊕, ⊗, 0, 1⟩)
    α[0] ← 1
    for n = 1, ... N do
        for t_n ∈ T do
            α[n, t_n] ← ⊕_{t_{n−1} ∈ T} exp(score(⟨t_{n−1}, t_n⟩), w, n) ⊗ α[n − 1, t_{n−1}]
    return α[N, EOT]
```

Algorithm 4. Forward algorithm that computes the same thing as Algorithm 3, but then in a different fashion. This version is more intuitive.

8 *Finite-state automata*

A *finite-state automaton* (FSA) is a computational device that determines whether a string $\boldsymbol{s} \in \Sigma^*$ is an element of a given language $L \subseteq \Sigma^*$. To check whether a string is part of a language defined by an FSA, the FSA reads in letters of an input string $\boldsymbol{s} \in \Sigma^*$. Then, it transitions from state to state according to the transition function δ and the letters $a \in \boldsymbol{s}$. If there is a path from an initial state to a final state while taking transitions as specified, the FSA accepts the string and is part of its language.

Definition 4 (Finite-state automaton)**.** A finite-state automaton is defined by a 5-tuple $\mathcal{A} = \langle \Sigma, Q, I, F, \delta \rangle$, where

- Σ is an alphabet;
- Q is a finite set of states;
- $I \subseteq Q$ is the set of initial states;
- $F \subseteq Q$ is the set of final states;
- $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is a finite multi-set that defines the transitions between states. Let $\langle q_0, a, q_1 \rangle \in \delta$ be a transition, then, if we make that transition, we go to state q_1 from q_0 and concatenate a to the current string.

A *weighted finite-state automation* (WFSA) adds weights from a semiring to the transitions ($\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \mathbb{K} \times Q$), initial states ($\lambda : I \to \mathbb{K}$), and final states ($\rho : F \to \mathbb{K}$). Weights are added together using the $\otimes$ operator.

An n-gram LM can be represented by a WFSA by setting the states to the n-length substrings. The initial states are all BOS. Every state is a final state, since we can end on any state. Two states map to each other if the two n-length substrings can follow each other, and the weight is the probability of the target vertex. The initial and final weights are all $\mathbf{1}$.

A CRF can also be represented by a WFSA, where each POS tag is a state. We can start and end on any state. The transitions then go from tag to tag, where the weight is the score of the target tag following the source tag. The initial weights are then the score of the tag following BOS, and the target weights are the score of EOSfollowing the tag.

The Kleene star of an alphabet Σ is defined as

$$\Sigma^* \doteq \bigcup_{n=0}^{\infty} \Sigma^n.$$

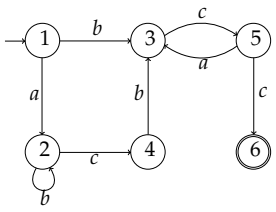
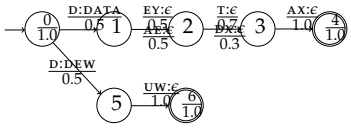


Figure 4. The graph of a simple FSA, where initial states are denoted by an incoming arrow and final states are denoted by a double circle. This defines the language $L = \{b(ca)^*cc, a(b)^*cb(ca)^*cc\}$.



A *weighted finite-state transducer* (WFST) further adds an output alphabet. The transitions then go from state to state while mapping input characters to output characters. Formally, $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Omega \cup \{\epsilon\}) \times \mathbb{K} \times Q$, where Ω is the output alphabet.

8.1 WFST composition

WFST composition of two transducers $\mathcal{T}_1$ and $\mathcal{T}_2$ is a common operation that involves mapping the inputs of $\mathcal{T}_1$ to the outputs of $\mathcal{T}_2$. This requires the output alphabet of $\mathcal{T}_1$ to be equal to the input alphabet of $\mathcal{T}_2$. Intuitively, it is the same as first running the input through $\mathcal{T}_1$ then that output through $\mathcal{T}_2$,

$$x \overset{\mathcal{T}_1}{\Longrightarrow} z \overset{\mathcal{T}_2}{\Longrightarrow} y.$$

The weight of mapping x to y using the composition of two WFSTs is the semiring-sum of the weights of all possible transformations of the above form.

Definition 5 (WFST composition)**.** Formally, the composition $\mathcal{T} = \mathcal{T}_1 \circ \mathcal{T}_2$ of two WFSTs,

$$\mathcal{T}_1 = \langle \Sigma, \Omega, Q_1, I_1, F_1, \delta_1, \lambda_1, \rho_1 \rangle$$
$$\mathcal{T}_2 = \langle \Omega, \Xi, Q_2, I_2, F_2, \delta_2, \lambda_2, \rho_2 \rangle,$$

is the WFST $\mathcal{T} = \langle \Sigma, \Xi, Q, I, F, \delta, \lambda, \rho \rangle$, such that

$$\mathcal{T}(x, y) = \bigoplus_{z \in \Omega^*} \mathcal{T}_1(x, z) \otimes \mathcal{T}_2(z, y).$$

```
1: function NaiveComposition(T1, T2)
2:     T ← ⟨Σ, Ω, Q, I, F, δ, λ, ρ⟩                    ▷ Create a new WFST
3:     for q1, q2 ∈ Q1 × Q2 do
4:         for q1 --a:b/w1--> q1', q2 --c:d/w2--> q2' ∈ E1(q1) × E2(q2) do
5:             if b = c then
6:                 Q ← Q ∪ {(q1, q2), (q1', q2')}          ▷ Add states
7:                 δ ← δ ∪ {(q1, q2) --a:d/w1⊗w2--> (q1', q2')}   ▷ Add arcs
8:     for (q1, q2) ∈ Q do                              ▷ Initial and final weights
9:         λ((q1, q2)) ← λ1(q1) ⊗ λ2(q2)
10:        ρ((q1, q2)) ← ρ1(q1) ⊗ ρ2(q2)
11:    return T
```

Algorithm 5. Naive version of the algorithm for computing the composition of two WFSTs.

8.2 Pathsum

A path $\pi \in \delta^*$ is an ordered set of *consecutive* transitions,

$$\left(q_1 \xrightarrow{a_1:b_1/w_1} q_2, q_2 \xrightarrow{a_2:b_2/w_2} q_3, \ldots, q_{N-1} \xrightarrow{a_N:b_N/w_N} q_N \right).$$

The weight of this path is defined as

$$w(\boldsymbol{\pi}) \doteq \lambda(q_1) \otimes \bigotimes_{n=1}^{N} w_n \otimes \rho(q_N).$$

The pathsum is then the semiring-sum over all paths in a WFST,

$$\begin{aligned} Z(\mathcal{T}) &\doteq \bigoplus_{\boldsymbol{\pi} \in \Pi(\mathcal{A})} w(\boldsymbol{\pi}) \\ &= \bigoplus_{\boldsymbol{\pi} \in \Pi(\mathcal{A})} \left(\lambda(q_1) \otimes \bigotimes_{n=1}^{N} w_n \otimes \rho(q_N) \right). \end{aligned}$$

Under the real semiring, the pathsum computes

$$\sum_{\boldsymbol{\pi} \in \Pi(\mathcal{A})} \left(\lambda(q_1) \times \prod_{n=1}^{N} w_n \times \rho(q_N) \right),$$

which is the normalizer, while under the Viterbi semiring, the pathsum computes

$$\max_{\boldsymbol{\pi} \in \Pi(\mathcal{A})} \left(\lambda(q_1) \times \prod_{n=1}^{N} w_n \times \rho(q_N) \right),$$

which is the maximum score of a path.

8.3 Lehmann's algorithm

Generally, there are an infinite amount of paths in a WFST, because of possible cycles. Thus, we need to design an algorithm that can compute this potentially infinite semiring-sum. Lehmann's algorithm [Lehmann, 1977] computes the semiring-sum matrix $\boldsymbol{R}$ over all inner paths between all pairs of nodes in $\mathcal{O}(|Q|^3)$ under a closed semiring. Then, we can use these to compute the normalizer with the following matrix multiplication,

$$Z(\mathcal{T}) \doteq \bigoplus_{i,k \in Q} \lambda(q_i) \otimes \boldsymbol{R}_{ik} \otimes \rho(q_k).$$

function LEHMANN($\boldsymbol{W}$)
  ▷ $\boldsymbol{W} \in \mathbb{R}^{|Q| \times |Q|}$ *is a matrix over a closed semiring* ◁
  $\boldsymbol{R}^{(0)} \leftarrow \boldsymbol{W}$
  for $j \leftarrow 1, \ldots, |Q|$ **do**
    for $i \leftarrow 1, \ldots, |Q|$ **do**
      for $k \leftarrow 1, \ldots, |Q|$ **do** ▷ *Loop through pairs of states*
        $\boldsymbol{R}_{ik}^{(j)} \leftarrow \boldsymbol{R}_{ik}^{(j-1)} \oplus \boldsymbol{R}_{ij}^{(j-1)} \otimes \left(\boldsymbol{R}_{jj}^{(j-1)} \right)^* \otimes \boldsymbol{R}_{jk}^{(j-1)}$
  return $\boldsymbol{I} \oplus \boldsymbol{R}^{(|Q|)}$

Algorithm 6. Lehmann's algorithm to compute the inner path semiring-sums.

TODO: Intuitition of Lehmann's algorithm.

9 *Transliteration*

Transliteration is the mapping of strings in one character set to strings in another character set. An example of this is the phonetic translation of English words. Formally, we want to develop a probabilistic model that can map strings from input vocabulary Σ to an output vocabulary Ω, *i.e.*, we want to compute $p(y \mid x)$ for all $x \in \Sigma^*, y \in \Omega^*$. We can use a WFST to specify the transliteration of Σ^* to Ω^* as a globally normalized model. The scoring function is then the semiring-sum over all paths that aligns x with y,

$$\text{score}(y, x) \doteq \sum_{\boldsymbol{\pi} \in \Pi(y)} w(\boldsymbol{\pi}).$$

To compute the normalizer, we need to design a WFST such that the input can only be x and that the output can be any element of Ω^*. To compute $\text{score}(y, x)$, we need a WFST such that the input can only be x and the output only y. We can guarantee such behavior by defining three transducers,

- $\mathcal{T}_x$ is the transducer that maps x to x;
- $\mathcal{T}_\theta$ is the transducer that maps any source string in Σ^* to any target string Ω^* (Figure 6);
- $\mathcal{T}_y$ is the transducer that maps y to y.

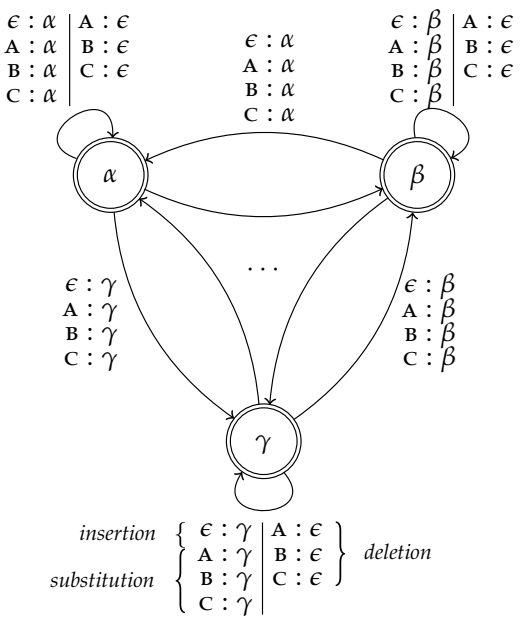
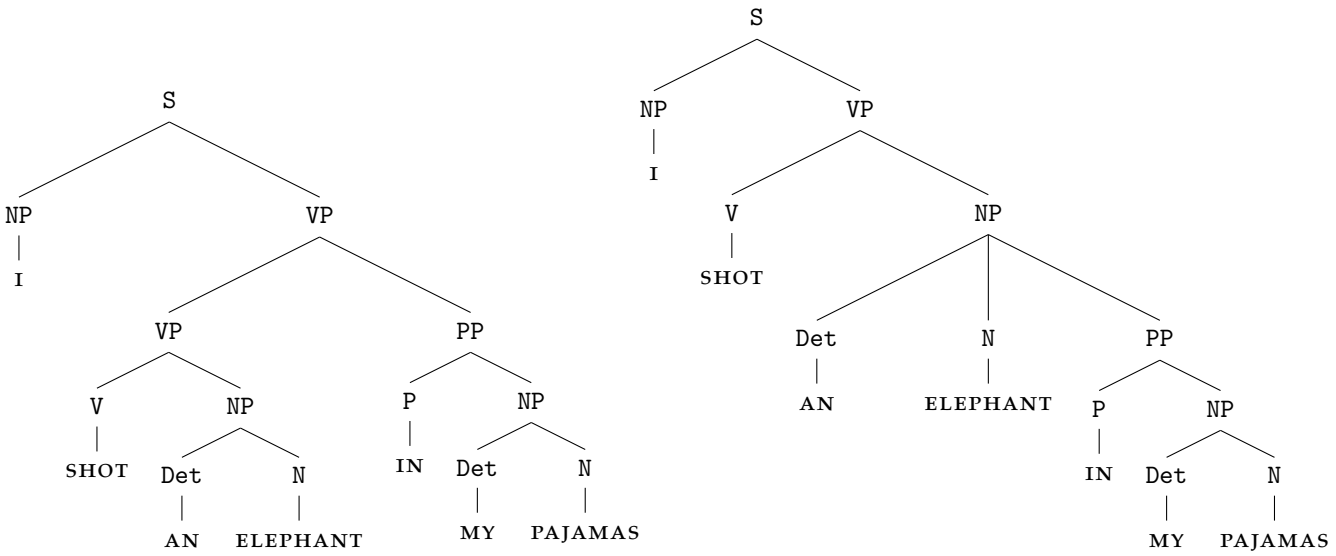


Figure 6. Mapping WFST between the alphabets $\{\text{A}, \text{B}, \text{C}\}$ and $\{\alpha, \beta, \gamma\}$. The vertex label denotes the last added symbol to the output string.

We can compose $\mathcal{T}_x \circ \mathcal{T}_\theta$ to get a transducer that has as input only x and output any target string in Ω^*. We can use Lehmann's algorithm to compute the normalizer $Z_\theta(x)$ using the real semiring, and the maximally scoring output using the Viterbi semiring. We can then use the transducer composition $\mathcal{T}_x \circ \mathcal{T}_\theta \circ \mathcal{T}_y$ to compute $\text{score}_\theta(y, x)$ by using the real semiring (or the log semiring to keep it in log-space). Thus, we have all the components we need for training and inference.

10 Context-free parsing



(a) The shooter is wearing the pajamas.

(b) The elephant is wearing the pajamas.

Figure 7. Two possible constituency trees of the ambiguous sentence "I shot an elephant in my pajamas."

A parse tree is a hierarchy of constituents, where a constituent is a multi-word unit that functions as a single unit. Each constituent encapsulates all of its leaf descendants, which are the words of the sentence. We say that a tree yields the sentence that can be found on its leaves. However, language is ambiguous, so some sentences have multiple trees that yield it. The goal is to compute the best parse tree that yields a given natural language input sentence.

10.1 Context-free grammars

Intuitively, a grammar defines a set of sentences that are deemed grammatical. Any sentence that can be yielded by a tree that consists of rules defined by the grammar is deemed grammatical.

Definition 6 (Context-free grammar)**.** A context-free grammar (CFG) is defined as $G = \langle \mathcal{N}, \mathrm{S}, \Sigma, \mathcal{R} \rangle$, where

1. $\mathcal{N}$: A set of non-terminal symbols, written as uppercase letters $\mathtt{N}_1, \mathtt{N}_2, \ldots$.
2. $\mathtt{S}$: A distinguished start non-terminal. Every complete parse tree must have this symbol at its root.
3. Σ: An alphabet of terminal symbols, written as lowercase letters $a_1, a_2, \ldots$.
4. $\mathcal{R}$: A set of production rules of the following form,

$$\mathtt{N} \to \boldsymbol{\alpha},$$

where $\mathtt{N} \in \mathcal{N}$ and $\boldsymbol{\alpha} \in (\mathcal{N} \cup \Sigma)^*$.

A *context-free grammar* (CFG) encodes a subset of Σ^*, where a sentence is only part of the subset if we can construct a tree from $\mathcal{R}$ that yields the sentence, starting from $\mathtt{S}$.

10.2 Parsing

We might be able to assign multiple trees to a single sentence. To be able to pick the best tree, we can assign a probability to each rule, and pick the tree with the highest probability.

Definition 7 (Probabilistic context-free grammar)**.** A probabilistic CFG is defined as $G = \langle \mathcal{N}, S, \Sigma, \mathcal{R}, p \rangle$, where $p : \mathcal{R} \to [0, 1]$ is a locally normalized probability distribution over rules. The probability of a tree under a PCFG is defined as follows,

$$p(\boldsymbol{t} \mid s) = \prod_{r \in \boldsymbol{t}} p(r).$$

Instead of probabilities, we could also, more generally, assign weights to each production rule. The score of assigning a tree $\boldsymbol{t}$ to a sentence $\boldsymbol{w}$ then decomposes over the production rules,

$$\text{score}(\boldsymbol{t}, \boldsymbol{w}) \doteq \sum_{r \in \boldsymbol{t}} \text{score}(r),$$

where a tree $\boldsymbol{t}$ is simply a multiset of rules. However, we run into the problem that the normalizer $Z(\boldsymbol{w})$ will diverge if the ruleset contains a cycle rule, *e.g.*, $\mathtt{N} \to \mathtt{N}$.

Definition 8 (Chomsky normal form)**.** A grammar is in Chomsky normal form if all rules are of the following form,

$$\mathtt{N}_1 \to \mathtt{N}_2 \mathtt{N}_3$$
$$\mathtt{N} \to a.$$

Theorem 1 (CNF theorem)**.** *The CNF theorem states that for any grammar G, we can find another grammar G' that accepts the same set of strings and probabilities as G and is in CNF.*

A CFG in *Chomsky normal form* (CNF) does not contain any cyclic rules, since they are not allowed by the permitted rule forms. Thus, the normalizer can no longer diverge, since there are not an infinite amount of trees that yield the same string $\boldsymbol{w}$. Furthermore, the CNF theorem guarantees that we can create all the same CFGs in CNF as if we did not constrain them to be in CNF.

10.3 Cocke-Kasami-Younger algorithm

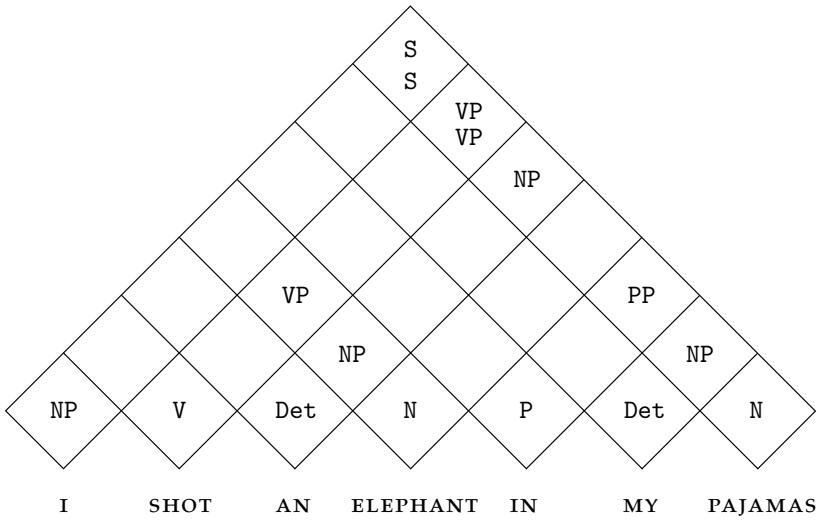


Figure 8. The CKY chart of the ambiguous sentence "I shot an elephant in my pajamas." See Figure 7 for the resulting trees.

Despite there not being an infinite amount of trees in CNF form, there are still an exponential amount of trees. Thus, we need to design an algorithm to efficiently compute the normalizer. The Cocke-Kasami-Younger (CKY) [Cocke, 1969, Kasami, 1966, Younger, 1967] algorithm provides an efficient dynamic program to compute the normalizer $Z(\boldsymbol{w})$ of CFGs in CNF. It works by looking at iteratively larger spans, and the subtrees that make up these spans, since a span from i to j can only be made up of smaller spans within this span, *e.g.*, there might be a subtrees that cover i to $i+1$ and $i+1$ to j. See Figure 8 for an illustration of how the algorithm works. It starts from the bottom of the chart and works its way up using dynamic programming.

This algorithm runs in $\mathcal{O}\big(N^3 \cdot |\mathcal{R}|\big)$, where N is the length of the sentence. We could also semiringify this algorithm to compute the best parse, where the Viterbi semiring finds the maximally scoring tree, and the real semiring computes the normalizer.

```
function WeightedCKY(w, ⟨N, S, Σ, R⟩, score)
    N ← |w|
    C ← 0                                                    ▷ Chart
    for i = 1, ..., N do
        for X → w_i ∈ R do
            C[i, i+1, X] ← C[i, i+1, X] ⊕ exp score(X → w_i)
    ▷ Compute score of every span in O(N^3 · |R|)                ◁
    for ℓ = 2, ..., N do
        for i = 1, ..., N − ℓ + 1 do
            k ← i + ℓ
            for j = i + 1, ..., k − 1 do
                for X → YZ ∈ R do
                    C[i, k, X] ← C[i, k, X] ⊕ exp score(X → YZ) ⊗
                    C[i, j, Y] ⊗ C[j, k, Z]
    return C[1, N + 1, S]
```

Algorithm 7. Semiringified CKY algorithm.

11 *Dependency parsing*

Dependency parsing is an alternative to constituency parsing. The basic idea is to link every word with its *syntactic head*.[18]

[18] We encode this as an arc in a graph, see Figures 9 and 10.

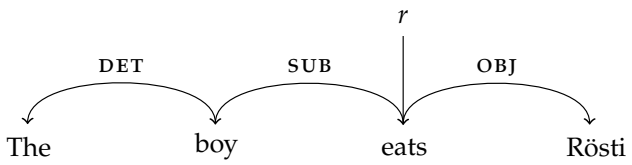


Figure 9. Projective dependency tree of "The boy eats Rösti."

In a dependency tree, only one word gets to be the root, and each word has a single parent, called its *syntactic head*. This allows for words to be linked that have other words in between not part of this structure, in contrast to constituency parsings.

There are two types of dependency trees, projective and non-projective. Projective dependency trees do not allow for crossing arcs, which make them closely related to constituency trees. Non-projective dependency trees do allow for crossing arcs, which will be the focus of this text. An example of a non-projective dependency tree can be found in Figure 10.

As always, we want to be able to parameterize a probability distribution over non-projective spanning trees, given a sentence $\boldsymbol{w}$. So, we need to be able to compute the normalizer Z. However, there are $(N-1)^{N-2}$ spanning trees with the single root constraint, thus we

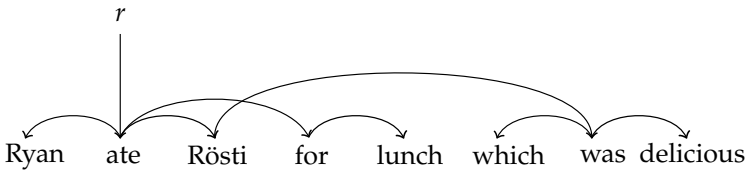


Figure 10. Non-projective dependency tree, without labels, of "Ryan ate Rösti, which was delicious."

need to design a more efficient algorithm that makes use of its structure. We do this by decomposing the scoring function over the edges,

$$\operatorname{score}(\boldsymbol{t}, \boldsymbol{w}) \doteq \operatorname{score}(r, \boldsymbol{w}) \sum_{(i \to j) \in \boldsymbol{t}} \operatorname{score}(i, j, \boldsymbol{w}).$$

Thus, we only need a matrix $\boldsymbol{A}$, containing $\exp \operatorname{score}(i, j, \boldsymbol{w})$ and a vector $\boldsymbol{\rho}$, containing $\exp \operatorname{score}(r, \boldsymbol{w})$.

Theorem 2 (Kirchhoff's matrix-tree theorem [Kirchhoff, 1847])**.** *For an undirected unweighted graph $\mathcal{G}$ with N vertices, let $\boldsymbol{L}$ be the graph Laplacian,*

$$L_{ij} = \begin{cases} -A_{ij} & i \neq j \\ \sum_{k \neq i} A_{kj} & \text{otherwise,} \end{cases}$$

where $\boldsymbol{A}$ is the adjacency matrix, i.e., *$A_{ij} = 1$ if $i \sim j$, otherwise $A_{ij} = 0$. Let $\hat{\boldsymbol{L}}_i \in \mathbb{R}^{(N-1)\times(N-1)}$ be the matrix created by removing the i-th row and column of $\boldsymbol{L}$. Then, we have*

$$N_T(\mathcal{G}) = \det\big(\hat{\boldsymbol{L}}_i\big),$$

where $N_T(\mathcal{G})$ is the number of trees in $\mathcal{G}$.

Tutte [1948] generalized Kirchhoff's MTT to directed trees, which allows us to compute the normalizer $Z(\boldsymbol{w})$ in $\mathcal{O}\big(N^3\big)$ as follows,[19]

$$Z(\boldsymbol{w}) = \det(\boldsymbol{L}).$$

[19] Since computing the determinant can be done in $\mathcal{O}\big(N^3\big)$.

However, this does not account for the single-root constraint. Koo et al. [2007] further generalized Tutte's MTT by modifying the graph Laplacian,

$$L_{ij} = \begin{cases} \rho_j & i = 1 \\ -A_{ij} & i \neq j \\ \sum_{k \neq i} A_{kj} & \text{otherwise.} \end{cases}$$

So, we can now compute $Z(\boldsymbol{w}) = \det(\boldsymbol{L})$ in $\mathcal{O}\big(N^3\big)$.

However, we are not able to semiringify this algorithm. Thus, we must design a new algorithm for inference.

11.1 Chu-Liu-Edmonds algorithm

A valid dependency tree must adhere to the following three constraints,

- All non-root nodes have exactly one incoming edge;
- No cycles;
- Only one outgoing edge from the root.

This is called an *arborescence*,[20] and we can find the maximum-weight arborescence with the Chu-Liu-Edmonds algorithm [Chu, 1965, Edmonds et al., 1967], sped up to $\mathcal{O}(N^2)$ by Tarjan [1977].

[20] The first two constraints are satisfied by the maximum-weight spanning tree, which we could compute with Kruskal's algorithm. However, it does not adhere to the third constraint, because Kruskal's algorithm only works for undirected graphs.

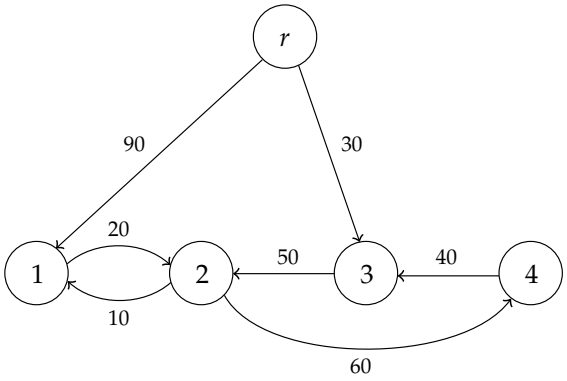


Figure 11. Example graph on which the Chu-Liu-Edmonds algorithm will be illustrated.

The algorithm starts by constructing the *greedy graph*, which is the graph that takes the best incoming edge to each node, except the root. See Figure 12.

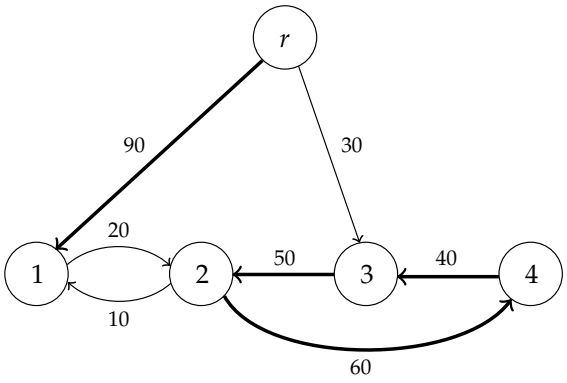


Figure 12. Greedy graph. This graph contains a cycle.

If the greedy graph contains a cycle, we *contract* the cycle into a single node c, and break the cycle by reweighting the enter edges, which are the edges that go into c. See Figure 13.

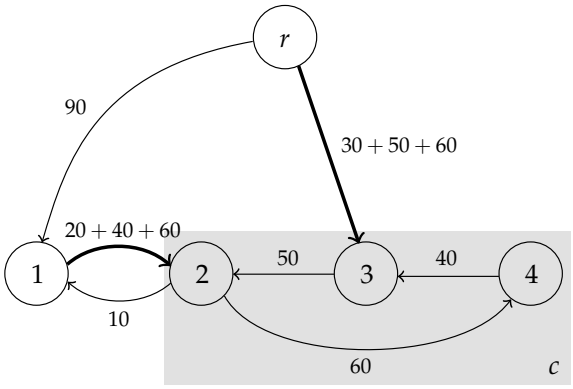


Figure 13. Contract the cycle by reweighting the **enter** edges.

Then, we pick the greedy graph from the contracted graph. If there is more than one edge emanating from the root, we need to delete edges outgoing from the root to satisfy the single-root constraint. We remove the edge with the lowest *swap score*, which is the difference between the next-best incoming edge and the current incoming edge of each node in the graph. After this, we expand the graph by selecting edges from c that break the cycle and ensure the single-parent constraint. See Figure 14.

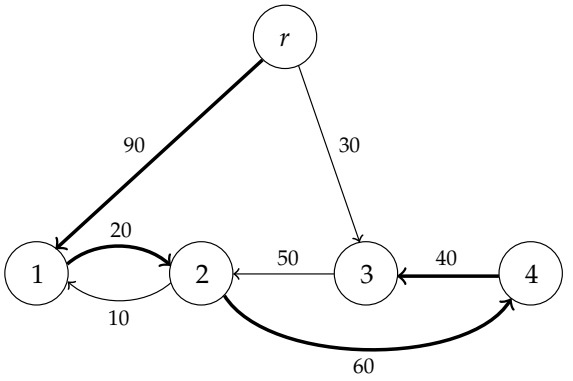


Figure 14. Expand by picking the greedy graph from the contracted graph. Then, if there are more than one edges emanating from r, remove the edge with the lowest swap score. Then pick the edges that break the cycle within c.

Do these steps recursively until a graph results that satisfies all three constraints. This tree is the best scoring dependency tree.

12 *Semantic parsing*

Language cannot only be syntactically ambiguous, but also semantically ambiguous. Also, syntactically valid sentences do not necessarily mean anything. A good example of this is Chomsky's famous sen-

tence "Colorless green dreams sleep furiously." In semantic analysis, we want to be able to reduce a natural language sentence, such as "Everyone loves someone else", to its logical form,

$$\forall p[\text{Person}(p) \to \exists q[\text{Person}(q) \wedge p \neq q \wedge \text{Loves}(p, q)]].$$

Note that this sentence is ambiguous, because we can swap the order of the quantifiers to a different logical form,

$$\exists p[\text{Person}(p) \to \forall q[\text{Person}(q) \wedge p \neq q \wedge \text{Loves}(q, p)]].$$

The challenge of semantic analysis, as always, is parsing a natural language sentence to its logical form. For this, we must use the *principle of compositionality*, which states that the meaning of a complex expression is a function of the meanings of that expression's constituent parts.[21]

[21] This must hold, because otherwise we would not know the meaning of most sentences, since we have not heard most possible order of words that form a sentence.

Proverbs are edge cases to this principle, since those words have meaning together that are independent of its parts.

12.1 *Lambda calculus*

Lambda calculus [Church, 1932] is a model for semantic analysis, based on the principle of compositionality. On a high level, lambda calculus first parses a sentence into simpler constituents, and then constructs the semantic representations using a bottom-up approach.

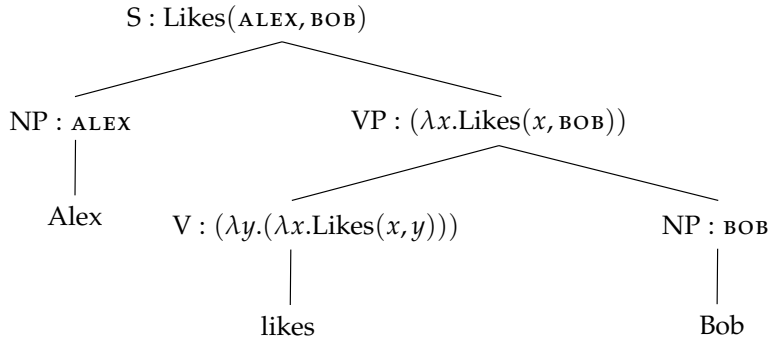


Figure 15. Lambda calculus on "Alex likes Bob."

Lambda calculus has only the following rules,

- $x, y, z, \ldots$ are variables;
- $(\lambda x.f(x))$, which is the *lambda operator*, where x is a variable and $f(x)$ an expression;
- $(M\,N)$, which is an application of function M to an argument N, where they are both lambda terms.

A variable is bound if it belongs to a scope of abstraction holding its name. Otherwise, a variable is free. *E.g.*, x is bound to λx and z is free in $((\lambda x.\lambda y.\text{Likes}(x, y))z)$.

To be able to represent natural language sentences, we also need the following,

- Constants that represent objects, denoted by *e.g.* ALEX, BOB, ...;
- Predicates that represent relations between objects, denoted by *e.g.* $\text{Teacher}(\cdot), \text{Likes}(\cdot, \cdot), \ldots$;
- Quantifiers $\exists$ and $\forall$.

Furthermore, lambda calculus has two operations,

- α-conversion is the process of renaming the variable of a lambda operator and all of its bound occurrences. *E.g.*,

$$(\lambda x.x) \to (\lambda y.y);$$

- β-reduction is the process of applying one lambda term to another, *i.e.*, in $(\lambda x.M\, N)$, we replace all occurrences of x in M by N, and remove the lambda operator. *E.g.*,

$$(\lambda x.(\lambda y.xy)\, z) \to (\lambda y.zy).$$

However, sometimes a β-reduction would result in a free variable becoming bound. Then, we would first need to apply an α-conversion. Two lambda terms are equivalent if one can be obtained from the other after a series of α-conversions and β-reductions.

12.2 Combinatory logic

Combinatory logic [Curry et al., 1958] 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, it uses complex functions using a few primitive higher order functions.

The basic terms of combinatory logic are

- $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}, \ldots$ are variables;
- $\mathbf{I}, \mathbf{S}, \mathbf{K}$ are the primitive combinators, which are functions that map functions to functions.

Terms are then recursively constructed using the rule of application, where $(\mathbf{A}\,\mathbf{B})$ denotes applying $\mathbf{A}$ to $\mathbf{B}$.

The following are the primitive combinators,[22]

[22] In combinatory logic, parentheses are left-associative, *e.g.*, $(\mathbf{K}\,\boldsymbol{x}\,\boldsymbol{y})$ means $((\mathbf{K}\,\boldsymbol{x})\,\boldsymbol{y})$ and $(\mathbf{S}\,\boldsymbol{x}\,\boldsymbol{y}\,\boldsymbol{z})$ means $((\mathbf{S}\,\boldsymbol{x})\,\boldsymbol{y}\,\boldsymbol{z})$.

- $(\mathbf{K}\, x\, y) = x$ manufactures constant functions;
- $(\mathbf{S}\, x\, y\, z) = (x\, z\, (y\, z))$, which applies x to y after first substituting z into each of them;
- $(\mathbf{I}\, x) = x$ works as the identity function.[23]

Any lambda term is equivalent to a combinatory term that only uses the **S** and **K** combinators.

Further, there are the following combinators that are introduced for convenience,

$$\begin{aligned}(\mathbf{C}\, x\, y\, z) &= ((x\, z)\, y) && \text{cross}\\ (\mathbf{B}\, x\, y\, z) &= (x\, (y\, z)) && \text{composition}\\ (\mathbf{T}\, x\, y) &= (y\, z) && \text{type-raising.}\end{aligned}$$

The **B** and **T** combinators will be used in combinatory categorial grammars.

12.3 Combinatory categorial grammars

Combinatory categorical grammars (CCG) are an efficiently parsable group of grammars that are mildly context-sensitive, which means that they have more expressive power than context-free grammars. CCGs allow us to model *coordination* and *cross-serial dependencies* in language, which CFGs cannot.

Definition 9 (Combinatory categorial grammar)**.** A combinatory categorial grammar is a 5-tuple

$$\langle V_T, V_N, S, f, R\rangle,$$

where

- V_T is a finite set of terminals;
- V_N is the finite set of atomic categories;
- $S \in V_N$ is the distinguished start category;
- f is a function mapping terminals $V_T \cup \{\epsilon\}$ to finite subsets of $C(V_N)$;
- R is a finite set of combinatory rules.

$C(V_N)$ is the infinite set of categories that contains all elements of V_N and recursively contains all elements such that if $c_1, c_2 \in C(V_N)$, then $c_1/c_2, c_1\backslash c_2 \in C(V_N)$.

[23] **I** as a primitive combinator is not necessary, since it can be constructed from **S** and **K**,

$$\begin{aligned}((\mathbf{S}\,\mathbf{K}\,\mathbf{K})\, x) &= (\mathbf{S}\,\mathbf{K}\,\mathbf{K}\, x)\\ &= (\mathbf{K}\, x\, (\mathbf{K}\, x))\\ &= x.\end{aligned}$$

CCGs have two main parts: a lexicon that associates words with categories and rules that specify how categories can be combined into other categories. The lexicon contains all information specific to a given language, *i.e.*, valency, word order, and semantics. The structure information is encoded in the categories.[24]

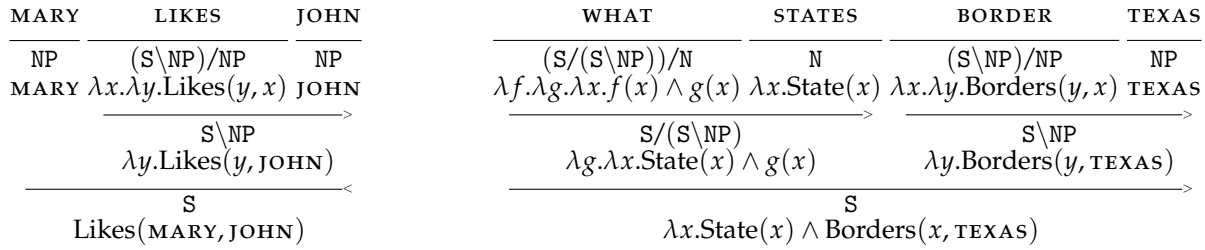
[24] Unlike CFGs, which encode structure in their rules.

The rules R are the following (inspired by combinatory logic),

$$\begin{aligned}
\mathtt{X/Y}\quad \mathtt{Y} &\implies \mathtt{x} && (>)\\
\mathtt{Y}\quad \mathtt{X\backslash Y} &\implies \mathtt{X} && (<)\\
\mathtt{X/Y}\quad \mathtt{Y/Z} &\implies \mathtt{X/Z} && (\mathbf{B}_>)\\
\mathtt{Y\backslash Z}\quad \mathtt{X\backslash Z} &\implies \mathtt{X\backslash Z} && (\mathbf{B}_<)\\
\forall \mathtt{T} \in V_N : \mathtt{X} &\implies \mathtt{T/(T\backslash X)} && (\mathbf{T}_>)\\
\forall \mathtt{T} \in V_N : \mathtt{X} &\implies \mathtt{T\backslash(T/X)} && (\mathbf{T}_<),
\end{aligned}$$

of which there are also generalized versions.

TODO: CCG parsing?



(a) CCG derivation of "Mary likes John."

(b) CCG derivation of "What states border Texas?"

Figure 16. Simple example CCG derivations.

13 *Transformers*

Attention is a mechanism in neural networks that a model can learn to make predictions by selectively attending to a given set of data by using query $\boldsymbol{q}$, key $\boldsymbol{k}$, and value $\boldsymbol{v}$ vector representations. The query and key vectors are used to determine how much weight should be given to the value vector.[25] The weights are computed by as $\boldsymbol{\alpha}_i = \text{softmax}(\boldsymbol{q}_i^\top \boldsymbol{k}_i)$, so the values after the attention block can be computed as

$$\text{att}(\boldsymbol{x}_i) = \sum_j \alpha_{ij} \boldsymbol{v}_i.$$

Self-attention blocks learn the query, key, and value representations from data. More specifically, it learns matrices $\boldsymbol{W}_Q$, $\boldsymbol{W}_K$, and $\boldsymbol{W}_V$ and

[25] Note the parallel with dictionaries/hashmaps in programming languages, but, in the attention mechanism, we do a "soft-lookup".

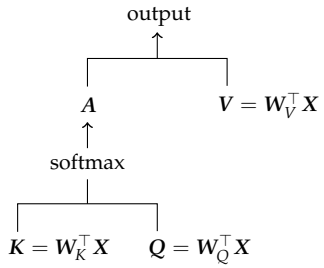


Figure 17. Self-attention mechanism.

computes the vectors from these matrices:

$$\begin{aligned}\boldsymbol{q}_i &= \boldsymbol{W}_Q^\top \boldsymbol{x}_i \\ \boldsymbol{k}_i &= \boldsymbol{W}_K^\top \boldsymbol{x}_i \\ \boldsymbol{v}_i &= \boldsymbol{W}_V^\top \boldsymbol{x}_i.\end{aligned}$$

Then, we can use these to compute the output of the self-attention block:

$$\text{self-att}(\boldsymbol{X}) = \text{softmax}\left(\frac{(\boldsymbol{W}_Q^\top \boldsymbol{X})^\top (\boldsymbol{W}_K^\top \boldsymbol{X})}{\sqrt{d_q}}\right) \boldsymbol{W}_V^\top \boldsymbol{X},$$

where d_q is the square root of the dimensionality of the query and key vectors. Furthermore, we need to add a positional encoding to provide ordering information to the model.[26] This is done by a sinusoidal positional encoding and are simply combined with $\boldsymbol{x}_i$ by addition.

[26] The self-attention operation is permutation invariant.

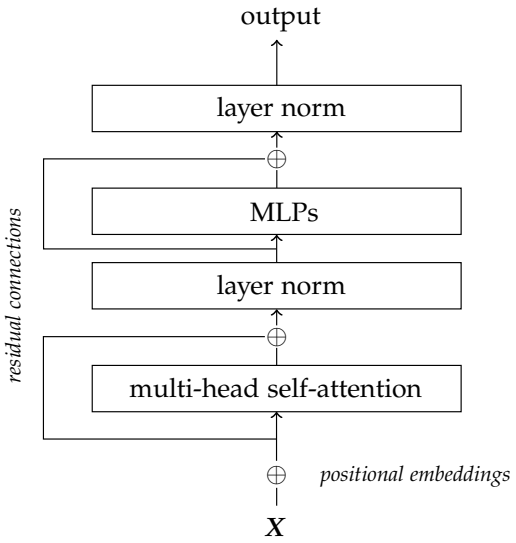


Figure 18. Transformer encoder/decoder architecture.

Transformers [Vaswani et al., 2017] use multi-headed self-attention, which is a module where self-attention is applied M times independently to the data. Thus, this module learns M different ways of looking at the same dataset. The outputs of each self-attention block is concatenated and linearly transformed to the expected dimensionality. Transformers follow this by normalization and MLP layers, as can be seen in Figure 18.

13.1 Translation

Translation is a sequence-to-sequence problem, where we want to compute the probability that $\boldsymbol{y}$ is the translation of $\boldsymbol{x}$. We can do this with

transformers by encoding the input sequence $\boldsymbol{x}$ using encoders, and feeding this representation of the input to a decoder. The decoder takes as input the input sequence and the already generated (incomplete) sequence $\boldsymbol{y}_{<i}$. It then runs the decoder as in Figure 18 and projects the output to a probability distribution over tokens using a linear layer, followed by softmax.

This will not give us the globally optimal translation $\boldsymbol{y}$, but performs very well in practice.

TODO: Figure of translation architecture with encoders and decoders.

References

Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. A neural probabilistic language model. *Advances in neural information processing systems*, 13, 2000.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014.

Yoeng-Jin Chu. On the shortest arborescence of a directed graph. *Scientia Sinica*, 14:1396–1400, 1965.

Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics*, pages 346–366, 1932.

John Cocke. *Programming languages and their compilers: Preliminary notes*. New York University, 1969.

Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. *Combinatory logic*, volume 1. North-Holland Amsterdam, 1958.

Jack Edmonds et al. Optimum branchings. *Journal of Research of the national Bureau of Standards B*, 71(4):233–240, 1967.

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

Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

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

Mohit Iyyer, Varun Manjunatha, Jordan Boyd-Graber, and Hal Daumé III. 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)*, pages 1681–1691, 2015.

Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. *Coordinated Science Laboratory Report no. R-257*, 1966.

Gustav Kirchhoff. Ueber die auflösung der gleichungen, auf welche man bei der untersuchung der linearen vertheilung galvanischer ströme geführt wird. *Annalen der Physik*, 148(12):497–508, 1847.

Terry Koo, Amir Globerson, Xavier Carreras Pérez, and Michael Collins. Structured prediction models via the matrix-tree theorem. In *Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 141–150, 2007.

Daniel J Lehmann. Algebraic structures for transitive closure. *Theoretical Computer Science*, 4(1):59–76, 1977.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

Bo Pang, Lillian Lee, et al. Opinion mining and sentiment analysis. *Foundations and Trends® in information retrieval*, 2(1–2):1–135, 2008.

Robert Endre Tarjan. Finding optimum branchings. *Networks*, 7(1): 25–35, 1977.

William T Tutte. The dissection of equilateral triangles into equilateral triangles. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 44, pages 463–482. Cambridge University Press, 1948.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Daniel H Younger. Recognition and parsing of context-free languages in time n3. *Information and control*, 10(2):189–208, 1967.