# Human Language Technology

William Simoni and Marco Natali

February 2021

# Contents


**1 Introduction** **6**
- 1.1 History . . . 6
- 1.2 Statistical Machine Learning . . . 7
- 1.3 Deep Learning . . . 8

**2 Language Modelling** **10**
- 2.1 Language model . . . 10
- 2.2 Shannon's game . . . 14
- 2.3 Smoothing . . . 15
- 2.4 Zipf's law . . . 16
- 2.5 Evaluation and Perplexity . . . 17
- 2.6 Unknown words handling . . . 18

**3 Representation of Words** **20**
- 3.1 Word Meaning . . . 20
- 3.2 The vector space model . . . 21
  - 3.2.1 tf-idf weighting . . . 22
  - 3.2.2 Embedded similarity . . . 23
- 3.3 Word Embeddings . . . 26
  - 3.3.1 Word2Vec . . . 27
- 3.4 Matrix reduction . . . 33
- 3.5 Evaluation . . . 35
  - 3.5.1 Intrinsic Embeddings Evaluation . . . 35
  - 3.5.2 Extrinsic Vector Evaluation . . . 36
- 3.6 Laboratory . . . 36

3.6.1 Train a word2vec model . . . 36
3.6.2 Find analogies . . . 39
3.7 Other topics . . . 40
3.7.1 Embeddings in Neural Networks . . . 40
3.7.2 Limits of Word Embeddings . . . 41
3.7.3 Sentiment Specific word embeddings . . . 41
3.7.4 Context Aware word embeddings . . . 41

**4 Text Classification with Naïve Bayes Classifier 43**
4.1 Naïve Bayes Intuition . . . 44
4.1.1 Bag of Words . . . 46
4.1.2 Learning . . . 47
4.1.3 Classification . . . 48
4.2 Algorithm summary . . . 48
4.3 Evaluation measures . . . 50
4.4 Tips . . . 52

**5 Basic Text Processing 54**
5.1 Tokenization . . . 54
5.1.1 Terminology . . . 55
5.2 Word Normalization . . . 56
5.2.1 Lemmatization and Stemming . . . 56
5.3 Sentence Segmentation . . . 57

**6 Hidden Markov Model 58**
6.1 Markov Chain . . . 58
6.2 Hidden Markov model . . . 61
6.2.1 Evaluation - The forward algorithm . . . 64
6.2.2 Decoding - The Viterbi algorithm . . . 68
6.3 Speech tagging . . . 71

**7 Neural Network Classifier 75**
7.1 Towards non linearly separable problems . . . 75
7.2 Perceptron . . . 76
7.2.1 Perceptron Learning Rule . . . 78
7.3 Softmax classifier . . . 79
7.3.1 Cross entropy . . . 81
7.4 Again on POS tagging . . . 81

7.4.1 Maximum entropy Markov model (MEMM) . . . . . 83
7.5 Named Entity Recognition [TODO] . . . . . 84

**8 Tensorflow** **85**
8.1 Execution Models . . . . . 85
8.2 Automatic Differentiation . . . . . 86
8.3 Autograph . . . . . 88

**9 Parsing** **90**
9.1 Constituency Grammar approach . . . . . 91
9.2 Dependency Grammar . . . . . 93
9.3 Dependency parsing - Transitional based . . . . . 94
9.3.1 Non Projective Transitions . . . . . 98
9.4 Learning Phase . . . . . 100
9.4.1 Issues . . . . . 101
9.4.2 Dependency Parser using Neural Networks . . . . . 102
9.5 Graph-based Dependency Parsing . . . . . 103
9.6 Universal Dependencies . . . . . 104

**10 Convolutional Neural Networks for NLP** **109**
10.1 Convolution for text . . . . . 110
10.2 Sentiment analysis on Twitter . . . . . 113

**11 Machine Translation** **116**
11.1 Example of language similarities and divergences . . . . . 117
11.2 Classical MT methods . . . . . 118
11.2.1 Direct . . . . . 118
11.2.2 Transfer . . . . . 119
11.2.3 Interlingua . . . . . 120
11.3 Statistical MT . . . . . 122
11.3.1 Phrase Based Machine Translation . . . . . 123
11.3.2 Evaluation MT . . . . . 132
11.4 Neural MT [TO DO] . . . . . 133

**12 Transformer and Attention** **135**
12.1 From RNN to Self-Attention . . . . . 135
12.1.1 Attention . . . . . 136
12.2 Transformers . . . . . 139

12.2.1 Self-Attention . . . 141
12.3 Pretraining . . . 143
12.3.1 Pretraining decoders . . . 143
12.3.2 Pretraining encoders . . . 144
12.3.3 Pretraining encoders-decoders . . . 144
12.4 BERT . . . 144

**13 Analysis of Language models** **146**
13.0.1 LM Limits . . . 146

**14 Reading comprehension** **149**
14.1 SQuAD . . . 150
14.2 Neural Models . . . 150
14.2.1 BiDAF . . . 151
14.2.2 BERT . . . 151
14.2.3 Differences between BiDAF and BERT . . . 152

**15 Open Domain Question Answering** **154**
15.1 Alternative Approaches . . . 155

Chapter 1

# Introduction

**Motivation**: language is the most distinctive feature of human intelligence. Other mammals does certain things that we also do, but they lack language to express and communicate and someone think that language shapes thought. Emulating the language is a scientific challenge.

From a practical point of view, most of the relevant data, is expressed in an unstructured form, also information is mostly communicated by reading or writing e-mails, reports, or articles and the like, in conversations, on by listening/watching media. People had also tried to turn text into structured (HTML) or micro-format data, but this was complicated to do since requires universal set of annotation and also an additional effort for the user.

Entity linking attempts to provide a bridge between the two type of data.

## 1.1 History

NLP started to be explored in the early 50s. During WW2, Alan Turing built a computer that was able to crack the Enigma Machine. From here, there came the idea to create other machines able to "crack" other languages: people started to work on machine translation. Theoretical fundamental work was done on automata, formal languages, probabilities, and information theory. There was also created the first speech system.

NLP was mostly funded by military and a lot of it was just word substitution programs.

There was a little understanding of natural languages syntax, semantics,

pragmatics. Very soon, the problem appear to be **intractable**. And because of that, the discipline was neglected (AI winter). In 1978, Hans Moravec wrote that it was impossible to achieve AI because of insufficient computing power. According to him, we should have had a computer $10^9$ more powerfull than the 1978 computers. He expected that such computers would have been available in 2018.

Things changed in the 90s with the introduction of statistical methods. Also, people started to build data set that could be used in experiments. These data sets were used in specific challenges: NIST, Netflix, DARPA and so on.

In 2012, deep learning became very successful in recognizing images. in 2016 neural machine translation was introduced. In the last few years, things have changed drastically.

## 1.2 Statistical Machine Learning

The approach uses a supervised training and similar techniques for both speech and text. The paradigm of statistical machine learning is the following:

- Given a training set $\{x_i, y_i\}$.
- Choose a **set of features** to represent data: and item $x$ is turned into $\Phi(x) \in \mathbb{R}^D$.
- Choose an **hypothesis function** to compute $f(x) = \mathbb{F}_\theta(\Phi(x))$.
- Define a **cost function** on error with respect to examples:

$$J(\theta) = \sum_i (f(x_i) - y_i)^2$$

- **Objective**: find parameter $\theta$ that minimizes $J(\theta)$.

SML introduces a **standard approach** for each new problem. However, this imposed the tyranny of feature engineering because everything depended on what type of data we provided to the algorithm. To do this, we needed to spend hours hand engineering some features/doing feature selection, dimensionality reduction and so on.

## 1.3 Deep Learning

Deep learning was a breakthrough because allowed us to avoid this "tyranny": the system is able to learn itself the features, in this case the architecture is made of many layers of neurons. We don't need to make intermediate stage nor representation. We have just to plug in the data and the first layers take care of performing instructions of the features that are useful for the higher layers. This is called an end-to-end approach.

GPUs are another technological breakthrough and this is because they can run in parallel optimization stages. Besides, there is no need to protect shared memory access and low precision (half,single) is enough.

Complexity of deep learning system grows at non linear rate: AlexNet had 8 layers (in 2012), MS had 350 layers (in 2017). The more layers we have, the more data we need because we have to train all parameters in all the layers. That means also that the cost increases with more layers.

### Deep Learning and NLP

How we can represent words? One idea is **word embedding**: transform a word into a vector of hundreds of dimensions capturing many subtle aspects of its meaning (see figure 1.1). So, a vector where each entry captures an aspect of the word, so, in other words, we pass from a discrete representation (entry in dictionary) to a distributed representation (a vector), the word embedding can be competent by a **language model**, and we will se how to build a neural network that can build such language model.

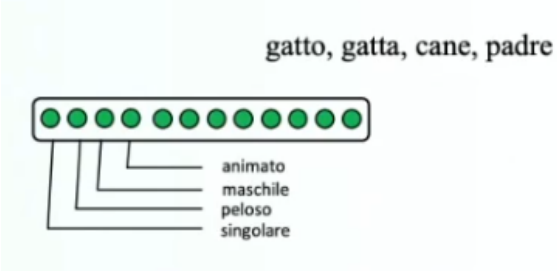



Figure 1.1: Each entry of the vector tells us some characteristics of the cat. For instance, the first element tells us if "cat" is a singular or plural word, the second element if "cat" is a male or female name, and so on.

But, how can we deal with sentences? A sentence is a sequence of words

and the meaning of a sentence is obtained composing the meaning of the words in the sentence, this idea get rise to sequence to sequence models.

### NMT (Neural Machine Translation)

We have a vector representation for each word. Then there is an encoder RNN that produces an encoding of the sentence we have to translate (combining the word linearly from left to right).

Then a decoder RNN, a language model, generates a translation based on the encoding previously generated. The translation is made linearly. The process is also explained by figure 1.2. This method worked reasonably well.

But just consider one word at a time could not be enough unless we consider the word in a context, also, an evolution of NMT added the notion of **attention**: when the encoder translates a word, it does not just consider the previous word in the output and the overall meaning of the sentence, but it considers also attention, that is a score given to every word and that indicate the importance of that word (if a word has a high score, then it is like to ask the encoder: "pay attention to this word").

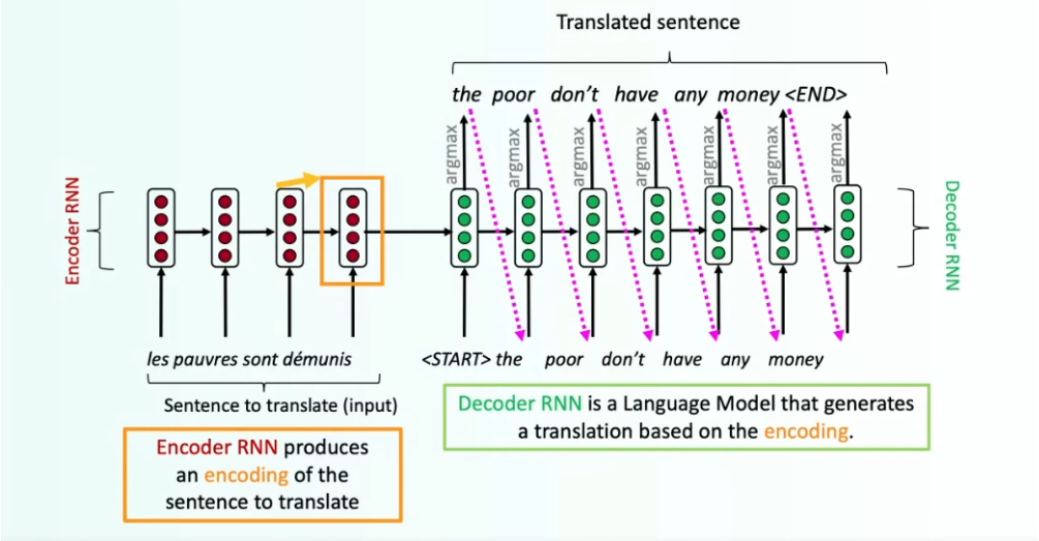



Figure 1.2: Each entry of the vector tells us some characteristics of the cat. For instance, the first element tells us if "cat" is a singular or plural word, the second element if "car" is a male or female name, and so on.

Chapter 2

# Language Modelling

## 2.1 Language model

A language model is a model that assigns a probability to a sentence. A language model can be applied to solve many problems, for instance:

**Machine translation**. Is "High winds tonite" more likely than "large winds tonite"?

**Spell correction**. Let us assume we wrote: "The office is about fifteen minuets from my". Of course, the sentence "about fifteen minutes from" is more likely than "about fifteen minutes from".

**Speech recognition**.

More formally, given a sequence "$w_1 w_2 w_3 \dots w_n$", a language model computes:

$$P(W) = P(w_1, w_2, w_3, \dots w_{n-1}, w_n)$$

This joint probability can be computed using the **chain rule**. Recall that the chain rule is:

$$\begin{aligned} P(w_1, w_2, w_3, \dots w_{n-1}, w_n) &= P(w_1)P(w_2|w_1)P(w_3|w_1, w_2) \dots P(w_n|w_1 \dots w_{n-1}) \\ &= \prod_{k=1}^{n} P(w_k \mid w_1^{k-1}) \end{aligned} \tag{2.1}$$

where $w_1^{k-1} = w_1, w_2, \dots, w_{k-1}$.

**Example 2.1.1**

$$P(\text{"}the\ big\ red\ dog\ was\text{"}) =$$
$$P(\text{"}the\text{"}) \times P(\text{"}big\text{"}|\text{"}the\text{"}) \times P(\text{"}red\text{"}|\text{"}the\ big\text{"})$$
$$\times P(\text{"}dog\text{"}|the\ big\ red\text{"}) \times P(\text{"}was\text{"}|\text{"}the\ big\ red\ dog\text{"})$$

We can estimate $P(w_n|w_1 \dots w_{n-1})$ counting how often the sequence $w_1 \dots w_{n-1}w_n$ appears in the corpus, and dividing this by the numbers of time $w_1 \dots w_{n-1}$ appears in the corpus:

$$P(w_n|w_1 \dots w_{n-1}) = \frac{C(w_1 \dots w_{n-1}w_n)}{C(w_1 \dots w_{n-1})} \tag{2.2}$$

We will indicate with $C$ the function that counts the number of occurrences of a given sentence.

**Example 2.1.2**

$$P(\text{"}was\text{"}|\text{"}the\ big\ red\ dog\text{"}) = \frac{C(\text{"}the\ big\ red\ dog\ was\text{"})}{C(\text{"}the\ big\ red\ dog\text{"})}$$

The practical problem is that there a lot of possible sentences. We will never be able to get enough data to compute the statistics for those long prefixes. So, we need to simplify the computations using the **Markov assumption**: we consider only a limited number of words rather than counting all the words that appear previously in the sentence.

**Example 2.1.3**

$$P(\text{"}was\text{"}|\text{"}the\ big\ red\ dog\text{"}) = P(\text{"}was\text{"}|\text{"}dog\text{"})$$

or maybe:

$$P(\text{"}was\text{"}|\text{"}the\ big\ red\ dog\text{"}) = P(\text{"}was\text{"}|\text{"}red\ dog\text{"})$$

So, for each component in the product 2.1:

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-N+1}^{n-1})$$

In particular, we are talking about a **bigram model** if N=2:

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-1}) \tag{2.3}$$

We can extend to 3-grams, 4-grams, 5-grams and so on. Note that this is, in general, an insufficient model of language because **language has long-distance dependencies**. We can often get away with N-gram models, though.

Equation 2.2 with bigrams becomes:

$$P(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i)}{C(w_{i-1})} \tag{2.4}$$

Example 2.1.4

Let us suppose that out corpus is composed by the following sentences:

- I am Sam
- Sam I am
- I do not like eggs and ham
- I like red eggs

We want to compute:

$$P(\text{"}I\ like\ eggs\text{"})$$

That, using the Markov assumption with N=2, is equal to:

$$P(\text{"}like\text{"}|\text{"}I\text{"})P(\text{"}eggs\text{"}|\text{"}like\text{"})$$

Using formula 2.4, we have that:

$$P(\text{"}like\text{"}|\text{"}I\text{"}) = \frac{C(\text{"}I\ like\text{"})}{C(\text{"}I\text{"})} = \frac{1}{4}$$

$$P(\text{"}eggs\text{"}|\text{"}like\text{"}) = \frac{C(\text{"}like\ eggs\text{"})}{C(\text{"}like\text{"})} = \frac{1}{2}$$

Therefore:

$$P(\text{"}like\text{"}|\text{"}I\text{"})P(\text{"}eggs\text{"}|\text{"}like\text{"}) = \frac{1}{4} \times \frac{1}{2} = \frac{1}{8}$$

This kind of estimation is called **Maximum Likelihood Estimate**(MLE) because it maximizes $P(TrainingSet|Model)$. In other words, the Maximum Likelihood Estimate of some parameter of a model $M$ from a training set $T$, is the estimate that maximizes the likelihood of the training set $T$ given the model $M$.

**Example 2.1.5** Maximum Likelihood Estimate

Let us suppose we tossed a coin ten times and that we get "head" six times and "cross" four times. We can use a simple binomial model assigning probability $p$ to the "head" case and $1-p$ to the "cross" case. Then we can write down the probability of all the data:

$$P(Data|p) = p^6(1-p)^4$$

Finally, we have to find the value parameter(s) that maximize this probability. In this case, we have to maximize only p. So we derive with respect to p, looking for the points where the derivative is 0:

$$\frac{\partial}{\partial p}p^6(1-p)^4 = 6p^5(1-p^4) - p^6 4(q-p)^3 =$$
$$= p^5(1-p)^3[6(1-p) - 4p] = p^5(1-p)^3(6-10p)$$

The derivative is 0 when:

- $p = 0$ (minimum)
- $p = 1$ (minimum)
- $p = 0.6$ (maximum)

So, the p we were looking for is 0.6.

We will usually deal with very small numbers, therefore, in order to avoid underflow problems, we will often use the logarithm:

$$log(p_1 \times p_2 \times p_3 \times p_4) = log(p_1) + log(p_2) + log(p_3) + log(p_4) \tag{2.5}$$

This change will also improve the speed of the code since adding is faster than multiplying.

## 2.2 Shannon's game

The Shannon's game consists in generating random sentences that are like sentences from which the model was derived. Algorithm 2.1 shows the pseudo-code to do this using bi-grams.

Listing 2.1: Shannon's game algorithm

```
1. Choose a random bigram <s>, w according to its
   probability
2. Choose a random bigram (w, x) according to its
   probability
3. repeat until we choose </s>
4. string the words together
```

The more are the words we consider, the more the sentence will be good. For instance, in picture 2.1 we can notice that the quality of the generation increases as we pass from uni-gram to bi-gram, from bi-gram to 3-gram and so on.

**Unigram**

To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have
Every enter now severally so, let
Hill he late speaks; or! a more to leg less first you enter
Are where exeunt and sighs have rise excellency took of.. Sleep knave we. near; vile like

**Bigram**

What means, sir. I confess she? then all sorts, he is trim, captain.
Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow.
What we, hath got so she that I rest and sent to scold and nature bankrupt, nor the first gentleman?

**Trigram**

Sweet prince, Falstaff shall die. Harry of Monmouth's grave.
This shall forbid it should be branded, if renown made it empty.
Indeed the duke; and had a very good friend.
Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done.

**Quadrigram**

King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in;
Will you not tell me who I am?
It cannot be but so.
Indeed the short and the long. Marry, "tis a noble Lepidus.

Figure 2.1: The corpus contains all the works written by Shakespeare.

N-grams only work well for word prediction if the test corpus looks like the training corpus. Although, it often doesn't happen in real life.

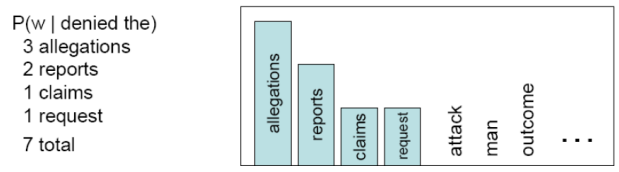
A language model must be trained on a large corpus of text to estimate good parameter values. A model can be evaluated based on its ability to predict a high probability for a disjoint (hold-out) test corpus (testing on the training corpus would give an optimistically biased estimate). Ideally, the training (and test) corpus should be **representative** of the actual application data.

in some cases, we may need to **adapt** a general model to a small amount of new (**in-domain**) data by adding highly weighted small corpus to original training data (domain adaptation).

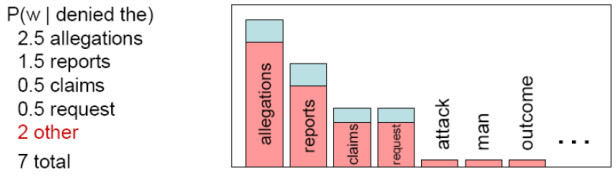
## 2.3 Smoothing

Since there are a combinatorial number of possible word sequences, many rare (but not impossible) combinations never occur in training, so MLE incorrectly assigns 0, so MLE incorrectly assigns zero to many parameters (aka sparse data). If a new combination occurs during testing, it is given a probability of zero and the entire sequence gets a probability of zero.

Therefore, parameters are **smoothed** (aka regularized) to reassign some probability mass to unseen events. So, we remove probability mass from seen events (discounting), and we add probability mass to the unseen ones. That is done to maintain a joint distribution that sums to 1. Figure 2.1 shows the effects of smoothing the probabilities.




(a) No Smoothing




(b) Smoothing

Figure 2.1: Example of smoothing.

There are several types of smoothing:

- Laplace smoothing:

$$P_{Laplace}(w_i|w_{i-1}) = \frac{C(w_i, w_i) + 1}{C(w_{i-1}) + |vocabulary|}$$

- Bayesian Prior Probability:

$$P_{prior}(w_i|w_{i-1}) = \frac{C(w_i, w_i) + P(w_i)}{C(w_{i-1}) + 1}$$

The probabilities of existing combinations may change a lot. In particular, using the Laplace smoothing, we usually add a smaller number than 1.

## 2.4 Zipf's law

The Zipf's law is a natural phenomenon that states that:

- A small number of events occur with **high frequency**.
- A large number of events occur with **low frequency**.

So, we can quickly collect statistics on the high frequency events. But, we might have to wait an arbitrarily long time to get valid statistics on low frequencies events. In particular, we have that:

$$frequency(event) \propto \frac{1}{rank(event)} \quad \propto = proportional\ to \tag{2.6}$$

Where the rank of an event indicates its rarity. The event with rank 1 is the least rare. There exists a constant $k$ such that $frequency \times rank = k$.

We can interpret the Zipf's law using the principle of **least effort**: both the speaker and the listener in communication try to minimize the effort:

- Speakers tend to use a small vocabulary of common (shorter) words.
- Listeners prefer a large vocabulary of rarer less ambiguous words.

Zipf's law is the result of this compromise.

For instance, taking the Brown Corpus, the most common words are: [(**'the'**, 62713), (**','**, 58334), (**'.'**, 49346), (**'of'**, 36080), (**'and'**, 27915), (**'to'**, 25732), (**'a'**, 21881), (**'in'**, 19536), (**'that'**, 10237), (**'is'**, 10011), (**'was'**, 9777), (**'for'**, 8841), (**'"'**, 8837), (**'"'"'**, 8789), (**'The'**, 7258), (**'with'**, 7012), (**'it'**, 6723), (**'as'**, 6706), (**'he'**, 6566), (**'his'**, 6466)]. The number after the word indicates the number of times the word occurs in the brown corpus.

## 2.5 Evaluation and Perplexity

To evaluate a language model, as always, we need a **test set** that is a data-set which is different than our training set. Then, we need also an **evaluation metric** to tell us how well our model is doing on the test set. In general, there are two kinds of evaluation: **extrinsic** evaluation and **intrinsic** evaluation.

In the extrinsic evaluation case, we define a task, and we see how our language model A is accurate with that given task. We do the same with another model B. Eventually, we compare the accuracy of the two models. This gives us an indirect measure of how good is our model. For instance, we may compare the two language models using the *language identification task*. So, we create an N-gram model for each language. We compute how likely a text comes from a language. And finally, we return the language with the highest probability. This method could require much time, in particular, because we need to define a task. Running experiments could require much time, and in the past, this method was pretty unfeasible. Now, thanks to new technologies, also this method is used to evaluate a language model (see for instance the GLUE benchmark: gluebenchmark.com).

The intrinsic evaluation uses an approximation called **perplexity**. The intuition behind perplexity as a measure, is the notion of surprise: how surprised is the language model when it sees the test set?:

- The **more surprised** the model is, the **lower the probability** it assigned to the test set.
- The **higher the probability**, the **less surprised** it was.

So, perplexity on a sentence is the inverse of the probability given to that sentence by the language model:

$$PP(W) = \sqrt[N]{\frac{1}{P(w_1\ w_2\ w_3\ \ldots w_N)}} \tag{2.7}$$

Note that we normalize the inverse by squaring it by N (the length of the sentence). This normalization is fundamental because we may have sequences of different lengths, and the more a sentence is long, the more is likely that its probability will be low. So, we need to normalize to compare the perplexity of both long and short sentences. Using the chain rule (2.1) over 2.7, we

obtain:

$$PP(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1\ w_2\ w_3\ \ldots w_{i-1})}} \tag{2.8}$$

That for bi-grams becomes:

$$PP(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}} \tag{2.9}$$

**Example 2.5.1** perplexity

Suppose we have to recognize sentences made by digits: "0 1 2 3 4 5 6 7 8 9". And that our language model chooses a number at random for every element of the sentence. Then, the perplexity is 10:

$$\begin{aligned} PP(W) &= P(w_1\ w_2\ w_3\ \ldots w_N)^{-\frac{1}{N}} \\ &= (\frac{1}{10}^{N})^{-\frac{1}{N}} \\ &= \frac{1}{10}^{-1} \\ &= 10 \end{aligned} \tag{2.10}$$

Minimizing the perplexity is the same as maximizing the probability and having a better model. For instance, an experiment that produced a model trained on 38 million words from the Wall Street Journal using a 19,979 word vocabulary, showed that with a uni-gram model the perplexity was 962, while with bi-grams the perplexity felt dramatically down to 170.

## 2.6 Unknown words handling

How to handle words in the test corpus that did not occur in the training data, i.e. out of vocabulary (OOV) words? We could train a model that includes an explicit symbol for an unknown word: UNK. In particular:

- We create a fixed lexicon L of size V.
- Any training word not in L changed to UNK.
- We train its probabilities like a normal word.

Then at decoding time, in text input, we use UNK probabilities for any word that is not in the training set.

Chapter 3

# Representation of Words

## 3.1 Word Meaning

From the **Webster dictionary**, the word "meaning" means:

- the idea that is represented by a word, phrase, etc.
- the idea that a person wants to express by using words, signs, etc.
- the idea that is expressed in a work of writing, art, etc.

Besides, the philosophy of language states that there is a bidirectional mapping between the symbol and the idea denoted by the symbol. Our goal is to find this mapping.

A simple way to assign meaning to words is "just" write down a dictionary. For instance, we could use **Wordnet** which provides word definitions, as well as synonyms, hypernyms and antonyms.

Listing 3.1: "good" synonyms in Wordnet

```
from nltk.corpus import wordnet as wn
for synset in wn.synsets('good'):
    print(synset.pos()),
    print(', '.join([l.name() for l in synset.lemmas()
        ]))

OUTPUT:
```

```
noun good, goodness
noun commodity, trade_good, good
adjective adept, expert, good, practiced, proficient,
   skillful
adjective full, good
adjective estimable, good, honorable, respectable
adjective beneficial, good
adjective good, just, upright
```

---

This method has many problems:

- It is imperfect: words do not live in isolation but are related to each other. And much of the time, the meaning of a word depends on the context (the surrounding words), for instance, in "the **good** is in the warehouse" and "Jim is a **good** boy", the word "good" has two completely different meanings.

- It can miss new meaning of words: wicked, badass, nifty, wizard, genius, ninja and so on. It is impossible to keep it up to date.

- Require human labor to create and adapt.

- It is hard to compute accurate word similarity.

## 3.2 The vector space model

The vector space model (VSM) is a representation for text used in information retrieval. A document/sentence is represented by a vector in a $n$-dimensional space:

$$v(d_1) = [t_1, t_2, \dots \ \dots, t_{n-1}, t_n]$$

Each dimension corresponds to a separate term. In other terms, the vector space model represents words by using a one-hot encoding. Each word is represented by a vector of size $|V|$ (cardinality of the vocabulary) that is 1 in one position and 0 elsewhere. Two vectors relative to two different words will have the 1 in different positions. The document representation is obtained by summing all the vectors of the words that it contains. The document

similarity is measured by the cosine distance, that, given two vectors $v$ and $u$ is equal to:

$$cosine\_similarity(v, u) = \frac{v \cdot u}{\|v\| \ \|u\|} \tag{3.1}$$

**Example 3.2.1** one-hot encoding

Let us suppose that our vocabulary contains three terms: hotel, city and motel. Then, we can represent each term as follow:

$$hotel = [1\ 0\ 0] \quad city = [0\ 1\ 0] \quad motel = [0\ 0\ 1]$$

The representation for the sentence "hotel motel" is:

$$hotel\ motel = [1\ 0\ 1]$$

### 3.2.1 tf-idf weighting

Salton suggested the tf-idf weighting to represent documents, where:

- tf stands for term frequency. So, the frequency of the word.
- idf stands for inverse document frequency:

$$idf_i = log\frac{N}{df_i}$$

  Where $df_i$ is the number of documents with word $i$.

We can represent each term $t$ by using the one-hot encoding, putting the $idf_t$ instead of 1:

$$v(t) = [0, \ldots, idf_t, \ldots, 0]$$

We can obtain the vector for a document summing all the term vectors and weighting each summand by its frequency in the document. For instance:

$$\begin{aligned} v(d) = v(you\ are\ great) = &[0, \ldots, tf_{you,d} \times idf_{you}, \ldots, \\ &tf_{are,d} \times idf_{are}, \ldots, \\ &tf_{great,d} \times idf_{great}, \ldots, 0] \end{aligned}$$

Note that tf depends both on the term and the document. The resulting document vectors are **sparse**:

$$|\{i|v_i(d) \neq 0\}| \ll n$$

The main limitation of this approach is that it can't capture similarities between terms. For example, if we have the following sentences:

$$d_1 = I\ go\ to\ the\ cinema$$
$$d_2 = I\ go\ to\ the\ movies$$
$$d_3 = I\ go\ to\ the\ beach$$

The corresponding vector representation is:

$$v(d_1) = [0, w_{I,d_1}, w_{go,d_1}, 0, 0, w_{to,d_1}, 0, w_{the,d_1}, w_{cinema,d_1}, 0, 0]$$
$$v(d_2) = [0, w_{I,d_2}, w_{go,d_2}, 0, 0, w_{to,d_2}, 0, w_{the,d_2}, 0, w_{movies,d_2}, 0]$$
$$v(d_3) = [0, w_{I,d_3}, w_{go,d_3}, 0, 0, w_{to,d_3}, 0, w_{the,d_3}, 0, 0, w_{beach,d_3}]$$

If we compute their similarity using the cosine similarity, we get that:

$$sim(v(d_1), v(d_2)) \approx sim(v(d_1), v(d_3)) \approx sim(v(d_2), v(d_3))$$

However, the first sentence is more bounded with the second one since they both talk about cinema.

So, in general there is no notion of similarity: words like cinema and movies have nothing in common. Search engine tried to address the issue using Wordnet synonyms, but this led to further problems.

The alternative solution was encoding similarity in the vector themselves, as we will see in the next section.

### 3.2.2 Embedded similarity

The intuition is model the meaning of a word by "embedding" it in a vector space. Thus, the meaning of a word is a vectors of numbers, also called **embeddings**. Each word will be represented by a **dense** vector, chosen so

that it is similar to vectors of words that appear in similar contexts:

$$movie = \begin{pmatrix} 0.23 \\ -0.1 \\ 0.34 \\ 0.15 \\ 0.07 \\ 0.7 \\ -0.12 \end{pmatrix}$$

The fundamental hypothesis used to model word similarity is that **co-occurring words are semantically related**. J.R.Firth first mentioned this intuition in 1957:

> *"You shall know a word by the company it keeps."*

So, it is quite an old idea that remains unheard of until the rise of modern statistical NLP. When a word $w$ appears in a text, its **context** is the set of words that appear nearby (if the context has a fixed size, we call it a context window). So, we will use the many contexts of $w$ to build up a representation of $w$.

The main approaches are two:

- Sparse vector representation:
  - Mutual information weighted word co-occurrences matrices.
- Dense vector representation:
  - Singular value decomposition.
  - Neural-network-inspired models.
  - Brown clusters.

### Word-Context Matrix (sparse vector rep.)

A word-context (or word-word) matrix is a $|V| \times |V|$ matrix $|X|$ that **counts** the frequencies of co-occurrence of words in a collection of contexts (i.e text spans of a given length). So, in a position [i,j], we will have the number of times the word i appears within a given context window with word j. Figure 3.1 shows an example of co-occurrence matrix.

This method unfortunately does not work so well. Figure 3.2 shows what are the words with which the words on the top co-occur the more and as you will notice something is wrong.

| | | Context words | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | ... | cook | eat | ... | changed | broke | ... |
| Words | cake | ... | 10 | 20 | ... | 0 | 0 | ... |
| | steak | ... | 12 | 22 | ... | 0 | 0 | ... |
| | engine | ... | 0 | 0 | ... | 3 | 10 | ... |
| | tire | ... | 0 | 0 | ... | 10 | 1 | ... |
| | ... | ... | ... | ... | ... | ... | ... | ... |

Figure 3.1: An example of a co-occurrence matrix. In positions [i,j] it contains the number of time the ith word appear with the jth word within a certain context window

| FRANCE<br>454 | JESUS<br>1973 | XBOX<br>6909 | REDDISH<br>11724 | SCRATCHED<br>29869 | MEGABITS<br>87025 |
|---|---|---|---|---|---|
| PERSUADE | THICKETS | DECADENT | WIDESCREEN | ODD | PPA |
| FAW | SAVARY | DIVO | ANTICA | ANCHIETA | UDDIN |
| BLACKSTOCK | SYMPATHETIC | VERUS | SHABBY | EMIGRATION | BIOLOGICALLY |
| GIORGI | JFK | OXIDE | AWE | MARKING | KAYAK |
| SHAFFEED | KHWARAZM | URBINA | THUD | HEUER | MCLARENS |
| RUMELLA | STATIONERY | EPOS | OCCUPANT | SAMBHAJI | GLADWIN |
| PLANUM | GSNUMBER | EGLINTON | REVISED | WORSHIPPERS | CENTRALLY |
| GOA'ULD | OPERATOR | EDGING | LEAVENED | RITSUKO | INDONESIA |
| COLLATION | OPERATOR | FRG | PANDIONIDAE | LIFELESS | MONEO |
| BACHA | W.J. | NAMSOS | SHIRT | MAHAN | NILGRIS |

Figure 3.2: Apparently, Xbox is more related to "decadent" or "divo" words than "game". Maybe, here something is wrong.

## 3.3 Word Embeddings

The idea is project word vectors $v(t)$ into a **low dimensional space** $\mathbb{R}^k$, $k \ll |v|$, of **continuous space word representations**(a.k.a., embeddings):

$$Embed : \mathbb{R}^{|V|} \rightarrow \mathbb{R}^k$$

$$Embed(v(t)) = e(t)$$

We would like to do this mapping in such a way that words with syntactic/semantic similarities group close in space.

The first attempt to do this was done by Collobert in 2011 [2]. The title of the paper was *Natural Langauge Processing (Almost) from Scratch* because using the word embedding, he could do NLP without using any external resource.

The method is a supervised learning method with a training set in which every example is composed of a center word and a context window that contains several words. For instance, if the sentence is "now the cat sits on" then the center word is **cat** while the context words are "now", "the", "sits", and "on". More in detail, we have:

- Positive examples from texts written by humans.
- Negative examples obtained replacing the center word of a sentence with a random one. For instance, if the sentence is "now the **cat** sits on", we can obtain a negative example by substituting cat with glue, getting "now the **glue** sits on".

Each word is associated to a word vector of fixed size $k$. At the beginning each word vector is initialized at random.

For every example, we concatenate the vectors. A multi-layer perceptron $U$ takes in input the concatenation. We train $U$ to return 1 if the center is related to the window context 0 otherwise. The loss used is the following:

$$Loss(\theta) = \sum_{x \in X} \sum_{w \in V} \max\{0, 1 - f_\theta(x) + f_\theta(x^{(w)})\} \tag{3.2}$$

Where, $f_\theta(x)$ is the output of $U$ and $x^{(w)}$ is the example x where we substitute the center with a random word. Ideally, if $U$ is well trained, for every $x$, we would have $f_\theta(x) = 1$ and $f_\theta(x^{(w)}) = 0$. Thus, the error would be 0. The

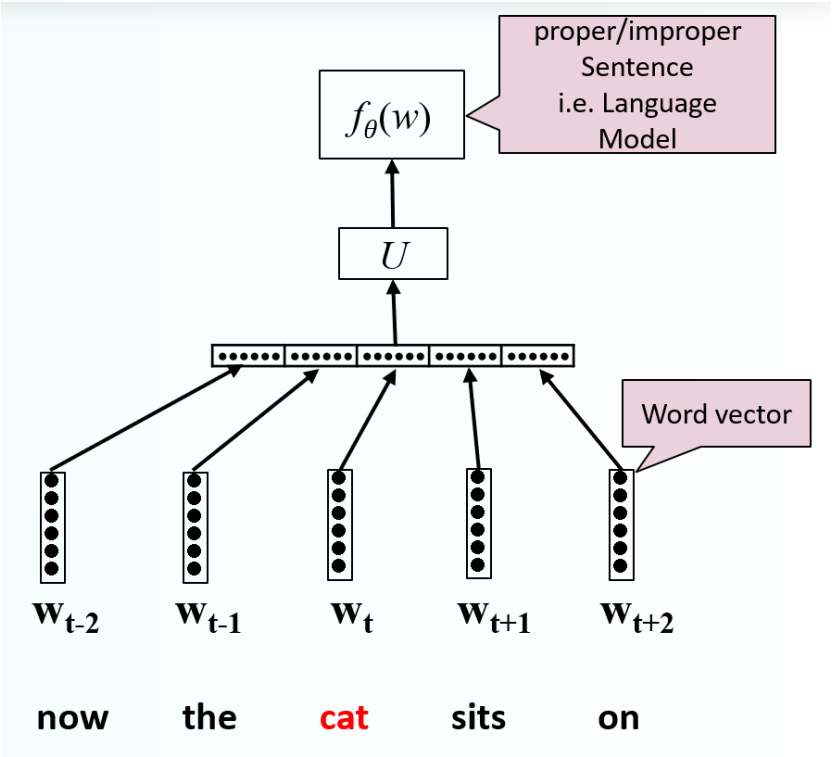



Figure 3.3: Collobert method schema

training is done by back-propagation ,and the error is propagated till the word vectors. Therefore, the word vectors are also modified during the training. Eventually, we throw away $U$ and we keep the word vectors.

The main problem of this algorithm was that it was very slow to be trained (a couple of weeks to train).

### 3.3.1 Word2Vec

Word2Vec (Mikolov et al. 2013) is a framework for learning word vectors. It represents an evolution of the Collobert method. Indeed, we pass from weeks of training to tens of minutes. The method idea was the following:

1. Collect a large corpus of text.

2. Every word in a fixed vocabulary is represented by a **vector**.

3. Go through each position $i$ in the text, which has a **center word** $c$ and **context("outside") words** $o$.

4. Use the **similarity of the words vectors** for $c$ and $o$ to calculate the probability of $o$ given $c$: $P(o|c)$.

5. **Keep adjusting the word vectors** to maximize this **probability**.

The paper mentions two methods:

- CBoW (Continuous Bag of Words): predict the center word $w_c$ given context words within window of width $m$.

- Skip-gram: predict the context words within window of width $m$, surrounding center word $w_c$.

We will see in detail Skip-gram since it is more effective than CBoW.

### Skip-gram

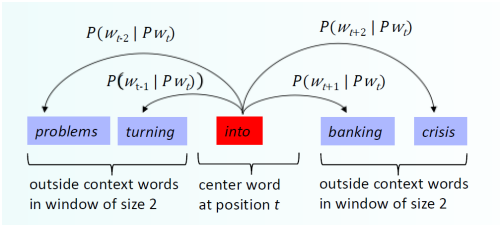
In the skip-gram method, we generate the embeddings by training a multi-layer perceptron that is trained to return how likely a word is in the context of a given term $t$. The training set is a corpus that we slide during the training using a window of a determined size $m$. As we can see in figure 3.3a, at every step, we take the central word and the context words within the window, we exec one training step, and finally, we slide the window (figure 3.3b).

As for the Collobert method, also here we get rid of the multi-layer perceptron saving only the word vectors. Besides, even though it is a prediction task, the network can be trained on any text, there is no need for **human-labelled data**. The usual context window size is 5 word before and 5 word after. Longer windows capture more semantics, but less syntax. And moreover, the network requires more time to be trained. We usually have embedding of size 200-300.

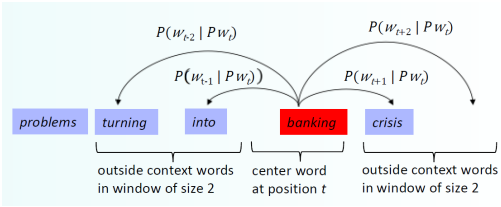
During the execution of the algorithm we maintain several data structure:

- $w \in \mathbb{R}^{|V|}$ for each word. It is a one-hot encoding.

- A matrix $V \in \mathbb{R}^{|V| \times d}$ that contains the embedded representation of words when they are central. $d$ is the size of the embedding.




(a) step i




(b) step i+1

Figure 3.3: Two steps of the Skip-gram method.

- A matrix $U \in \mathbb{R}^{d \times |V|}$ that contains the embedded representation of words when they are in the context. $d$ is the size of the embedding.

The matrices $V$ and $U$ are the parameters that are tuned during the training. At every step we:

- Compute the **embedding** of the central word $w_c$:

$$v_c = w_c V$$

  This multiplication takes the row relative to $w_c$ in V. So, it is like to do $v_c = V[c]$, where $c = w_c$ (it is the central word).

- Generate a score vector $z = v_c U$. Since U contains the embeddings of the context words, this is like calculate the similarity between the embedding of the central word and the embeddings of the context words (Remember that the dot product is equal to the cosine similarity if the vectors are normalized).

- Turn the score vector into a probability distribution using **softmax**.

$$\hat{y} = softmax(z)$$

$\ldots, \hat{y}_{c-m}, \ldots, \hat{y}_{c-1}, \hat{y}_{c+1}, \ldots, \hat{y}_{c+m}, \ldots$ are estimates of the probabilities of observing each context word. We expect that the probability will be high for the words that appear together with $w_c$.

The loss used during the training is very straightforward. We merely want to maximize the fact that words that appear in the context of a certain word receive a high probability. Therefore, for each position $t = 1, \ldots, T$, with a window size $m$, a given center word $w_c$, and a set of parameter $\theta$, which are the matrices $U$ and $V$, we want to find the $\theta$ that maximize:

$$Likelihood = L(\theta) = \prod_{c=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} P(w_{c+j}|w_c; \theta) \tag{3.3}$$

To make 3.3 more tractable, we take the negative log-likelihood. In this way, the problem turns into a minimization of $J(\theta)$:

$$\begin{aligned} J(\theta) &= -\frac{1}{T} log\ L(\theta) \\ &= -\frac{1}{T} \sum_{c=1}^{T} \sum_{-m \leq j \leq m,\ j \neq 0} log(P(w_{c+j}|w_c; \theta)) \end{aligned} \tag{3.4}$$

As already explained, we compute $P(w_{c+j}|w_c; \theta)$ by using the softmax function. The softmax function is defined as follow:

$$p_i = softmax(x_i) = \frac{e^{x_i}}{\sum_{k=1}^{V} e^{x_k}} \tag{3.5}$$

Therefore, substituting $x_i$ with the score vector $z$:

$$P(w_{c+j}|w_c; \theta) = \frac{e^{z_{c+j}}}{\sum_{w \in V} e^{z_w}} \tag{3.6}$$

That can also be written as:

$$P(w_{c+j}|w_c; \theta) = \frac{e^{u_{w_{c+j}} v_{w_c}}}{\sum_{w \in V} e^{u_w v_{w_c}}} \tag{3.7}$$

Where $u_i = w_i U$ is the embedding of the word $w_i$ when it is in the context and $V$ is not the matrix we were talking about before, but it is the vocabulary. Plainly to simplify the notation, we can write equation 3.4 as:

$$P(o|c; \theta) = \frac{e^{u_o v_c}}{\sum_{w \in V} e^{u_w v_c}} \tag{3.8}$$

Where $o$ is the context word and $c$ is the center word.

### Train the Skip-Gram model

We are going to tune the parameters $\theta$ using the gradient descent algorithm. The parameters are the matrices $U$ and $V$ that we can write as one unique vector of vectors:

$$\theta = \begin{pmatrix} v_1 \\ \dots \\ v_{|V|} \\ u_1 \\ \dots \\ u_{|V|} \end{pmatrix} \tag{3.9}$$

Where each row is a vector of size $d$ and represents an embedding to a specific word. Note that we have two vectors per word: the embedding stored in V and the embedding stored in U.

Our goal is to minimize the loss function 3.4 ($J(\theta)$) by computing the gradient and using it to update the parameters $\theta$:

Listing 3.2: Gradient Descent Algorithm

```
while True:
    grad -= gradient(J, corpus, theta)
    theta -= alpha * grad
```

In listing 3.2 alpha ($\alpha$) is the learning rate.

We now have to compute the gradient by computing the partial derivative of $J(\theta)$ with respect to $u_o$ and with respect to $v_c$. In the following, we show only the calculation to compute the partial derivative with respect to v. We can rewrite the loss function using the notation used in formula 3.8:

$$J(\theta) = -\frac{1}{T}\sum_{c\in\Gamma}\sum_{o\in context(c,m)} log(P(o|c;\theta)) \tag{3.10}$$

Where $\Gamma$ is the sequence of words $w_1, w_2, \dots, w_T$, and $context(c, m)$ is the set of words in the context of $c$ within a window of size $m$. We want to compute:

$$\begin{aligned} &\frac{\partial}{\partial v_c} - \frac{1}{T}\sum_{c\in\Gamma}\sum_{o\in context(c,m)} log(P(o|c;\theta)) \\ &= -\frac{1}{T}\sum_{c\in\Gamma}\sum_{o\in context(c,m)} \frac{\partial}{\partial v_c} log(P(o|c;\theta)) \end{aligned} \tag{3.11}$$

That using 3.8 becomes:

$$-\frac{1}{T}\sum_{c\in\Gamma}\sum_{o\in context(c,m)}\frac{\partial}{\partial v_c}log(\frac{e^{u_o v_c}}{\sum_{w\in V}e^{u_w v_c}}) \tag{3.12}$$

Deriving the logarithm in 3.12:

$$\begin{aligned}
\frac{\partial}{\partial v_c}log(\frac{e^{u_o v_c}}{\sum_{w\in V}e^{u_w v_c}}) &= \frac{\partial}{\partial v_c}log(e^{u_o v_c}) - \frac{\partial}{\partial v_c}log(\sum_{w\in V}e^{u_w v_c}) \\
\frac{\partial}{\partial v_c}log(e^{u_o v_c}) &= \frac{\partial}{\partial v_c}u_o v_c log(e) = \frac{\partial}{\partial v_c}u_o v_c = u_o \\
\frac{\partial}{\partial v_c}log(\sum_{w\in V}e^{u_w v_c}) &= \frac{1}{\sum_{w\in V}e^{u_w v_c}}\frac{\partial}{\partial v_c}\sum_{w\in V}e^{u_w v_c} = \frac{1}{\sum_{w\in V}e^{u_w v_c}}\sum_{x\in V}\frac{\partial}{\partial v_c}e^{u_x v_c} \\
&= \frac{1}{\sum_{w\in V}e^{u_w v_c}}\sum_{x\in V}e^{u_x v_c}u_x = \sum_{x\in V}\frac{1}{\sum_{w\in V}e^{u_w v_c}}e^{u_x v_c}u_x = \sum_{x\in V}P(x|c)u_x
\end{aligned}$$

Therefore:

$$\frac{\partial}{\partial v_c}J(\theta) = -\frac{1}{T}\sum_{c\in\Gamma}\sum_{o\in context(c,m)}(u_o - \sum_{x\in V}P(x|c)u_x) \tag{3.13}$$

With similar computations it is simple to also calculate $\frac{\partial}{\partial u_o}J(\theta)$.

### Optimizations

Note that $J(\theta)$ is a function of all windows in the corpus, and the windows are potentially billions. Therefore it might be too expensive to compute, or, it'd require a lot of time to make a single gradient update. So, we usually use a **stochastic gradient descent** algorithm in which we consider one window at a time.

Listing 3.3: Stochastic Gradient Descent Algorithm

```
while True:
    window = sample_window(corpus)
    grad -= gradient(J, window, theta)
    theta -= alpha * grad
```

Another computation that would require much time is the $log(\sum_{k \in V} e^{u_k})$ factor in the log-likelihood denominator because it sums over all the words in the dictionary. To overpass this problem, we compute it only on a small sample of negative examples:

$$log(\sum_{k \in E} e^{u_k}) \tag{3.14}$$

Where words $E$ are just a few and they are **sampled** using a biased uni-gram distribution U computed on training data.

At the end both $U$ and $V$ will define embeddings, so we can:

- Throw away $V$ and use just $V$.
- Average pairs of vectors for $V$ and $U$ into a single one.
- Append one embedding vector after the other, doubling the length.

The training cost of Word2Vec is linear in the size of the input. The training algorithm works well in parallel, given the sparsity of words in contexts and the use of negative sampling. Indeed, we can use multi-core CPU (or even GPU) to run SGD in parallel on each one. U and V would be shared between the two processes. The access to the parameters should be synchronized, but the fact that:

- Computation is **stochastic**, hence it is approximate anyhow.
- Parameters are huge: low likelihood of concurrent access to the same memory cell.

led to a non-synchronized program.

There exist other methods besides Word2Vec. For instance GLoVe [9] or fastText.

## 3.4 Matrix reduction

Another way to find similarities is to pack the co-occurrences matrix into a smaller matrix. There are many methods that do this such as SVD.

SVD (Singular Value Decomposition) is a method to decompose a matrix $X$ of size $m \times n$ into three matrices $U\Sigma V^*$ where:

- $U$ is orthonormal matrix of size $m \times n$.
- $\Sigma$ is a diagonal matrix of size $n \times n$. The element on the diagonal are called **singular values** of $X$ and are sorted decreasingly.
- $V$ is an orthonormal matrix of size $n \times n$, $V^*$ is its conjugate transpose.

As figure 3.4 shows, the basic idea is to keep the top $k$ singular values and set to zero all the other. Thus, the rows $U_k$ of size $m \times k$ are the dense representations of the feature.

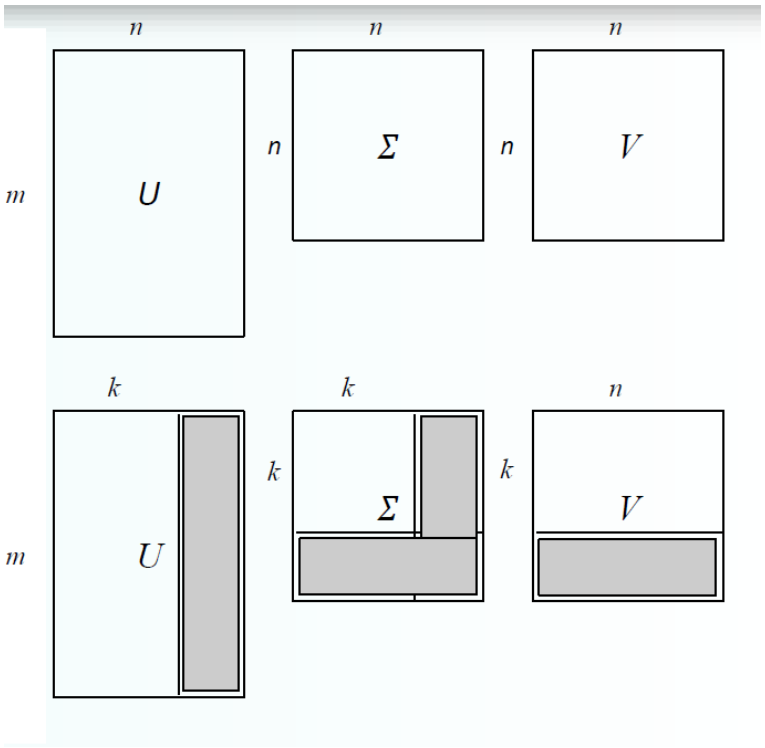



Figure 3.4: SVD method schema

Unfortunately, SVD computational cost grows quadratically for $n \times m$ matrices: $O(mn^2)$ FLOPS (when $n < m$).

Levy and Goldberg proved that Word2Vec Skip-Gram with negative sampling (SGNS) implicitly computes a factorization of a variant of SVD [6]. The same researcher showed that there is not any performance difference between SGNS and SVD methods. Hence, SGNS is suggested to be a good baseline, given its lower computational cost in time and memory [7].

## 3.5 Evaluation

### 3.5.1 Intrinsic Embeddings Evaluation

In this section, we will see some techniques used to do intrinsic evaluation where we just look at the vectors and see if some relations are visible and that corresponds to what we expect.

We can evaluate models through analogy evaluation using different models with different hyper-parameters. In picture 3.5, we compare various models in their ability to find syntactic similarities (slow and slower) and semantic similarities (Obama and Busch) inside the text. We can see that there is a kind of correlation between the size of the model and the accuracy.

| Model | Dim. | Size | Sem. | Syn. | Tot. |
|---|---|---|---|---|---|
| ivLBL | 100 | 1.5B | 55.9 | 50.1 | 53.2 |
| HPCA | 100 | 1.6B | 4.2 | 16.4 | 10.8 |
| GloVe | 100 | 1.6B | 67.5 | 54.3 | 60.3 |
| SG | 300 | 1B | 61 | 61 | 61 |
| CBOW | 300 | 1.6B | 16.1 | 52.6 | 36.1 |
| vLBL | 300 | 1.5B | 54.2 | 64.8 | 60.0 |
| ivLBL | 300 | 1.5B | 65.2 | 63.0 | 64.0 |
| GloVe | 300 | 1.6B | 80.8 | 61.5 | 70.3 |
| SVD | 300 | 6B | 6.3 | 8.1 | 7.3 |
| SVD-S | 300 | 6B | 36.7 | 46.6 | 42.1 |
| SVD-L | 300 | 6B | 56.6 | 63.0 | 60.1 |
| CBOW | 300 | 6B | 63.6 | 67.4 | 65.7 |
| SG† | 300 | 6B | 73.0 | 66.0 | 69.1 |
| GloVe | 300 | 6B | 77.4 | 67.0 | 71.7 |
| CBOW | 1000 | 6B | 57.3 | 68.9 | 63.7 |
| SG | 1000 | 6B | 66.1 | 65.1 | 65.6 |
| SVD-L | 300 | 42B | 38.4 | 58.2 | 49.2 |
| GloVe | 300 | 42B | 81.9 | 69.3 | 75.0 |

Figure 3.5: Analogy evaluation experiments

#### Polysemous words

Words can be polysemous: they can have multiple meanings. For instance, "bear" is an animal, while "to bear" is a verb. If a word is polysemous, we expect that its embedding will be close to the embeddings of all its meanings. Researches showed that the embedding of a word is a linear combination of its words senses[1]:

$$v_{bear} = \alpha_1 v_{bear_1} + \alpha_1 v_{bear_2} + \cdots + \alpha_N v_{bear_N}$$

Where:

$$\alpha_i = \frac{f_i}{\sum_{i=1}^{N} f_i} \quad for\ frequencies\ f_i$$

So we can separate the meanings by splitting them into clusters as shown in figure 3.6.

| tie | | | | |
|---|---|---|---|---|
| trousers | season | scoreline | wires | operatic |
| blouse | teams | goalless | cables | soprano |
| waistcoat | winning | equaliser | wiring | mezzo |
| skirt | league | clinching | electrical | contralto |
| sleeved | finished | scoreless | wire | baritone |
| pants | championship | replay | cable | coloratura |

Figure 3.6: In this example, we clustered the meanings of the word tie. As you can notice, we have something relative to clothes (the first column), something related to competitions, and so on.

### 3.5.2 Extrinsic Vector Evaluation

Another type of evaluation is the extrinsic evaluation which consists of testing/applying the embeddings to specific tasks. For instance, figure 3.7 compares two models in different tasks: SQuAD is a question-answer data set, NER is a named entity recognition task.

## 3.6 Laboratory

In this section, we will use some utilities implemented in the **Gensim** package.

### 3.6.1 Train a word2vec model

First thing first, we download the corpus from Peter Norvig (URL) that contains a novel about Sherlock Holmes, and we memorize it into a file that,

for the example, we called `big_text_file`. Then we divide it into tokenized sentences using the re package:

```
import re

sentences = []
with open("big_text_file.txt", mode='r', encoding='utf-8') as infile:
    for line in infile:
        sentences.append(
            re.split('[\W\d_]+', line.lower())
        )
```

More in detail, we read the file line by line, we lower case the line, and finally, we divide it into tokens using a regular expression.

We can enable the logging in the execution of `gensim` by writing:

```
import logging
logging.basicConfig(
    format='%(asctime)s : %(levelname)s : %(message)s',
    level=logging.INFO)
```

We can now train the word2vec on this corpus:

```
from gensim.models import Word2Vec
model = Word2Vec(sentences, size=100, window=10, min_count=5,
                 sg=1, iter=20, workers=8, negative=10)
```

Where, in the call of Word2Vec (see gensimDoc for more info):

- `sentences` is the list of tokenized sentences we have built before.
- `size` is the size of the embeddings.
- `window` is the size of the context window, in this case it will consider five words before and five words after the center word.
- `min_count = 5` indicates the algorithm to ignore all words with total frequency lower than 5.
- `sg=1` means that it uses the skip-gram algorithm.
- `iter` is the number of iterations for the training.

- `workers` is the number of worker threads to train the model.
- `negative` points out the number of samples to consider for the negative sampling.

Once we have trained the model, we can finally see how an embedding of a word looks like:

```
model.wv['fish']
```

```
array([ 0.74756837,  0.05586274,  1.1628306 , -0.17278896,  0.9122068 ,
       -0.09628863, -0.22926691, -0.20254345,  0.5017881 , -0.09342537,
        0.82623684,  0.24685174,  0.33824706, -0.30289736, -0.0443881 ,
        0.5018746 ,  0.53090125,  0.19504587, -0.46151868, -0.21057397,
        0.03885385, -0.12235668,  0.28065455,  0.19090518, -0.00345148,
       -0.4478233 , -0.71739995,  0.49980634, -0.52869207,  0.27513522,
       -0.6995776 , -0.6030818 , -0.15479267, -0.22906227,  0.10143632,
       -0.97097427,  0.34451795, -0.20441487, -0.1030857 , -0.30255976,
       -0.15010224, -0.854128  ,  0.15726735,  0.6295392 ,  0.21684332,
        0.08921834,  0.15276545, -0.25219816, -0.21228698, -0.11791059,
        0.2484122 , -0.18863186, -0.05399549, -0.39046487, -0.06947114,
        0.2185792 , -0.06159685, -0.15908414, -0.2626319 , -0.5016568 ,
       -0.24038818, -0.31661433,  0.53444266, -0.16337782,  0.26580048,
       -0.19544926,  0.04993691, -0.49606174,  0.21571356, -0.12664121,
       -0.1872635 , -0.13406026,  0.07689974,  0.0869696 ,  0.28113565,
        0.39695054,  0.4858435 ,  0.41348574, -0.28215876,  0.25510657,
        0.56208086,  0.6657028 , -0.06572041, -0.5161947 , -0.18938173,
        0.376724  , -0.08448596, -0.10193489,  0.5941915 ,  0.5111936 ,
       -0.58766836, -0.21560444,  0.20177682, -0.26430747,  0.05069062,
        0.02757518,  0.12745291, -0.19351438,  0.02847496, -0.18568787],
      dtype=float32)
```

We can also look at the words most similar to "crime" using the `most_similar` method that returns the top 10 closest terms, according to the cosine similarity of their embedding vectors:

```
model.wv.most_similar('crime')
```

```
[('committed', 0.5876615047454834),W
 ('selfish', 0.5843631029129028),
```

```
 ('defence', 0.5797097682952881),
 ('objection', 0.5732696056365967),
 ('inevitability', 0.55591881275177),
 ('evil', 0.5405635833740234),
 ('understands', 0.5341767072677612),
 ('crimes', 0.5314351320266724),
 ('corrupt', 0.5305241346359253),
 ('ascribe', 0.5263623595237732)]
```

Of course, we could use other measures as the plain euclidean (L2) distance. As you will notice, the results are a bit different:

```
[('defence', 3.154787063598633),
 ('selfish', 3.188154697418213),
 ('objection', 3.1942381858825684),
 ('ascribe', 3.2900640964508057),
 ('understands', 3.3091299533843994),
 ('attaining', 3.3279147148132324),
 ('lengths', 3.3338091373443604),
 ('der', 3.3694865703582764),
 ('creator', 3.4056594371795654)]
```

### 3.6.2 Find analogies

We want to see if the relation between the words is preserved. For instance, Paris stands at France as Rome stands at Italy. We can represent the relation between Paris and France by taking the difference between their embeddings. We would like to see if summing the Rome embedding with the embedding of the relation between Paris and France we get something close to the embedding of Italy:

$$e(France) - e(Paris) + e(Rome) \approx e(Italy) \tag{3.15}$$

We download an already trained model on the Google News Corpus from this URL. We decompress the file obtaining a 3.5 Gb file, and we load the model using the following code:

```
from gensim.models import KeyedVectors
news_model = KeyedVectors.load_word2vec_format(google_w2v_file, binary=True)
```

We can compute the most similar embeddings to the embedding obtained computing equation 3.15 by writing:

```
news_model.most_similar(
    positive=['rome', 'france'],
    negative=['paris']
)
```

```
[('italy', 0.519952118396759),
 ('european', 0.5075846314430237),
 ('italian', 0.5057743787765503),
 ('epl', 0.49074438214302063),
 ('spain', 0.4888668954372406),
 ('england', 0.4852672219276428),
 ('italians', 0.48424220085144043),
 ('kosovo', 0.4813492000102997),
 ('lampard', 0.4807734787464142),
 ('malta', 0.4788566827774048)]
```

We can also write a general function to find analogies:

```
def analogy(x1, x2, y1):
    result = news_model.most_similar(positive=[y1, x2], negative=[x1])
    return result[0][0]
```

## 3.7 Other topics

### 3.7.1 Embeddings in Neural Networks

If we want to process text with a neural network, we can add an embedding layer as the first layer. The layer consists of a matrix $W$ of size $V \times d$, where $d$ is the size of the embedding space. $W$ maps words to dense representations, and it can be:

- initialized at random. In that case, $W$ is updated to adapt the embeddings to the task.
- pretrained. In this case, we can both update $W$ or keep $W$ fixed.

### 3.7.2 Limits of Word Embeddings

The limits of word embeddings are:

- **Polysemous words**: the embedding will not be too close to its meaning because it will be just an average.
- **Limited to words**.
- **Represent similarities**, therefore, antonyms often appear similar (e.g. good and bad), and that is not so good with sentiment analysis and polysemous words. For instance, *The movie was exiting* and *The movie was boring* are considered similar.

### 3.7.3 Sentiment Specific word embeddings

A way to do sentiment analysis using embeddings is by training embeddings providing as output both the words in the context (as in the standard case) and whether the words in the context have a positive or negative attitude (the polarity). The main problem of this approach is providing this kind of data.
In the area of social network analysis, and specifically on twitter, researchers tried to generate this data by noting the presence of emoticons in the sentence [16]. A smiling face or a sad face can distinguish positive and negative sentences. This kind of learning is called **distance learning**.

### 3.7.4 Context Aware word embeddings

Instead of creating a single vector for each word, we could build different vectors depending of the context in which the word appear. The applications of context aware word embeddings are for example word sense disambiguation, document classification, polarity detection or adwords matching.

**ELMo** (Deep contextualized word representations)[10] represents the first attempt to do context aware word embeddings. ELMo generates embeddings for a word, based on the context it appears. Its main features are:

- The representation for each word depends on the entire context in which it is used (it is **contextual**).
- World representation combine all layers of a deep pre-trained neural network (it is a **Deep** neural network).

- ELMo representations are character based, allowing the network to use morphological clues (**Character based**). So, if in the training set we have *clue*, but not *clues*, ELMo will be able to build the meaning of *clues* up on the meaning of *clue*.

At the time of presentation, ELMo was state-of-the-art, as shown in figure 3.7.

| Task | SoA | ELMo |
|---|---|---|
| SQuAD | 84.4 | 85.8 |
| SNLI | 88.6 | 88.7 |
| SRL | 81.7 | 84.6 |
| Coref | 67.92 | 70.4 |
| NER | 91.93 | 92.22 |
| Sentiment | 53.7 | 54.7 |

Figure 3.7: Extrinsic evaluation of ELMo and SoA using several tasks. ELMo has always a better accuracy than SoA.

The most recent models are called **transformer** models because they take an input and transforms it into some other output. BERT is an example of **transformer model** [3].

Chapter 4

# Text Classification with Naïve Bayes Classifier

In this chapter, we will present the first algorithm, called Naïve Bayes Algorithm, to do text classification. The algorithm had been state-of-the-art for many years. It indeed received first and second place in KDD-CUP in 1997, among sixteen state-of-the-art algorithms.

The applications of text classification are many and variegate: SPAM recognition, author identification, recognize if a review is positive or negative, or figure out what is the subject of an article.

Let us now formalize what a classifier is and what we would like a learner to do:

**The classifier** implements a function that given a document $d \in D$ returns an element within a set of classes $C$. Formally: given a set of documents $D = \{d_1, d_2, \ldots, d_D\}$, a fixed set of classes $C = \{c_1, c_2, \ldots, c_K\}$, a classifier implements the function $f : D \to C$.

**The learner** takes a set of $N$ hand-labelled documents $T = \{(d_1, c_1), \ldots, (d_N, c_N)\}$, for example $T = \{(d_1, spam), (d_2, not\_spam), \ldots, (d_N, spam)\}$, and returns a learned classifier $f : D \to C$.

In the past, text classification was implemented using **hand-coded** rules, but even if the accuracy was very high, building and maintaining these rules was expensive. So, let us introduce the intuition behind the Naïve Bayes algorithm.

## 4.1 Naïve Bayes Intuition

We call this algorithm "Naïve Bayes algorithm" because it uses the Bayes's rule:

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)} \tag{4.1}$$

Another rule we will use during the chapter is the conditional probability:

$$P(B|A) = \frac{P(A \cap B)}{P(A)} \tag{4.2}$$

The goal of the algorithm is to find the "**best**" class $c$ for the document $d$. So, among all the classes in $C$, we search the class $c$ that more likely the document $d$ belongs to:

$$c_{map} = \arg\max_{c \in C} P(c|d) \tag{4.3}$$

Using the Bayes' rule (4.1) with $A = d$ and $B = c$:

$$c_{map} = \arg\max_{c \in C} \frac{P(c)P(d|c)}{P(d)} \tag{4.4}$$

Since we are maximizing for $c$, $P(d)$ is a constant and, therefore, we can merely drop it:

$$c_{map} = \arg\max_{c \in C} P(c)P(d|c) \tag{4.5}$$

We have the formula, but we have to formalize how to represent a document yet. We can represent a document $\in D$ using a vector $d$ such that $d[i]$ is the $i$-th word in that document:

$$d = \langle\ x_1, x_2, \ldots, x_n\ \rangle \tag{4.6}$$

Substituting $d$ with its representation in formula 4.5:

$$c_{map} = \arg\max_{c \in C} P(c)P(x_1, x_2, \ldots, x_n|c) \tag{4.7}$$

We have now to understand how to compute the two factors of the formula 4.7:

$P(c)$ can be estimated from the **frequency of classes** in the training examples that is:

$$\hat{P}(c_j) = \frac{\#Docs_j}{\#Docs} \tag{4.8}$$

$\#Docs_j$ is the number of documents labeled in category $c_j$ in the training set $T$, and $\#Docs$ is the total number of documents in the training set $T$.

$P(x_1, x_2, \ldots, x_n|c)$ is hard to be computed. Indeed, we should compute this probability for every combination of words and every possible class, bringing the number of parameters to consider toward $O(|V|^n \cdot |C|)$ where $V$ is the vocabulary and may contain millions of words.

We simplify the calculus of $P(x_1, x_2, \ldots, x_n|c)$ using the **Naïve Bayes conditional independence assumption**: we assume that the probability of observing the conjunction of attributes is equal to the product of the individual probabilities:

$$P(x_1, x_2, \ldots, x_n|c) = \prod_{i=1}^{n} P(x_i|c) \tag{4.9}$$

This assumption is not always right. For instance, looking at figure 4.1, it is not true that if you have the flu, then having the fever is independent of having muscle-ache.

**Example 4.1.1** Naïve Bayes assumption

Assume that our document is the one pictured in figure 4.1. Using the conditional independence assumption we have that:

$$\begin{aligned} &P(x_1 = \text{"}runnynose\text{"}, \ldots, x_5 = \text{"}muscle\_ache\text{"}|c) \\ &= P(x_1 = \text{"}runnynose\text{"}|c) \cdot \ \ldots \ \cdot P(x_5 = \text{"}muscle\_ache\text{"}|c) \end{aligned}$$

Using this assumption, equation 4.7 becomes:

$$c_{NB} = \arg\max_{c_j \in C} P(c_j) \prod_i P(x_i|c_j) \tag{4.10}$$

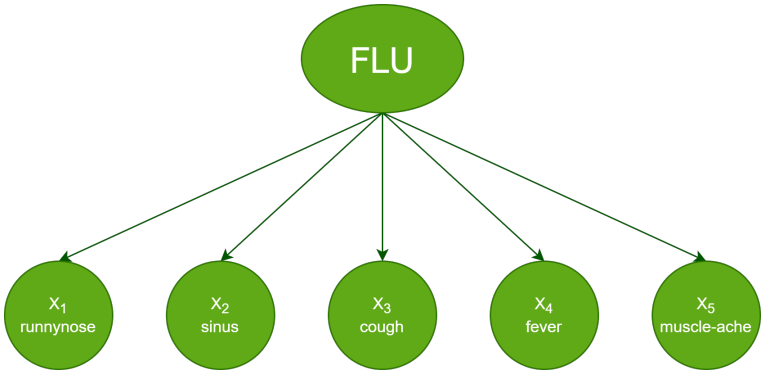



Figure 4.1: Representation of a document that indicates the flu synptoms.

There are still too many possibilities since we still have to consider all the possible positions of words in the document. For instance, $P(x_1 = "runnynose", \ldots, x_5 = "muscle\_ache"|c) \neq P(x_1 = "muscle\_ache", \ldots, x_5 = "runnynose"|c)$. Therefore, we slightly change the document representation in a way that we are independent of the positioning of words: "we put the sequence of words inside a **bag of words**".

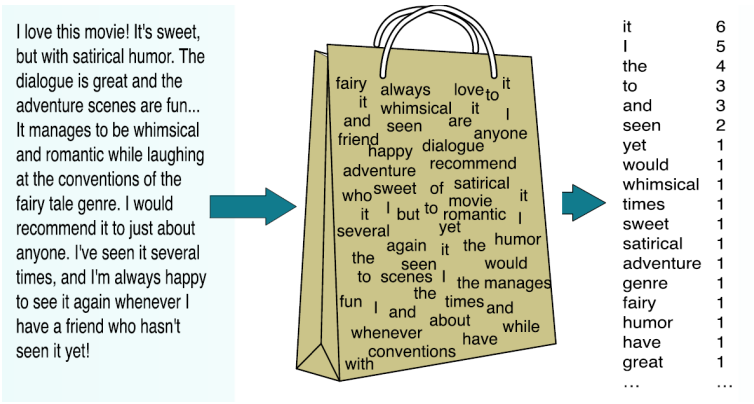



Figure 4.2: The bag of words representation. The words are filtered and put inside a "bag", and then word frequencies are counted.

### 4.1.1 Bag of Words

A bag of word is like a histogram: we assign every position of the vector to a specific word, and inside every position, we store the frequency of the word

in the document.

Before creating a bag of words, we usually filter the words that we should consider and the ones that we should throw away.

As we can notice from figure 4.2, the representation is entirely independent of the positioning of words in the document. Indeed, we totally lose that information.

### 4.1.2 Learning

We can eventually formalize the final formula to compute $P(x_i|c_j)$, that using a maximum likelihood estimate is:

$$\hat{P}(x_i|c_j) = \frac{count(X_i = x_i, C = c_j)}{count(C = c_j)} \tag{4.11}$$

Where $count(X_i = x_i, C = c_j)$ counts the number of time the word $x_i$ occurs in the documents of category $c_j$, while $count(C = c_j)$ counts the words in all the documents belonging to category $c_j$.

As we have already seen in chapter 2, using 4.11, we will incur problems if we meet a word that is not present in the training set. Indeed, assuming $x_c$ is not in the training set, we would have that $\forall j \ \hat{P}(x_c|c_j) = 0$, and therefore $\forall j \ P(d|c_j) = P(c_j) \prod_i \hat{P}(x_i|c_j) = 0$ for any document $d$ that contains $x_c$.

The solution to this problem, as already mentioned in chapter 2, is smoothing:

**Laplace Smoothing**:

$$\hat{P}(x_i|c_j) = \frac{count(X_i = x_i, C = c_j) + 1}{count(C = c_j) + k} \tag{4.12}$$

**Bayesian Unigram Prior**:

$$\hat{P}(x_i|c_j) = \frac{count(X_i = x_{i,k}, C = c_j) + mp_{i,k}}{count(C = c_j) + m} \tag{4.13}$$

Wrapping up, the learning algorithm will straightforwardly consists in computing the probabilities $\hat{P}(c_j)$ and $\hat{P}(x_i|c_j)$ for every word $x_i$ in the vocabulary of the collection of documents, and for every $c_j \in C$. Listing 4.1 shows the pseudo-code of the algorithm.

### 4.1.3 Classification

In the classification phase we have to find the category $c_{NB}$ such that:

$$c_{NB} = \arg\max_{c_j \in C} P(c_j) \prod_{i \in positions} P(x_i|c_j) \tag{4.14}$$

Where *positions* contains the word positions in current document which contain tokens that are present in the *Vocabulary*.

**Example 4.1.2** positions

Let us assume that the vocabulary is [America, And, Hello, Italians, Italy, World, Zoo] and that the document we want to classify contains "Hello Italians and Europeans" then *positions* will be $[2, 3, 1]$. We discarded Europeans because it is unknown in our vocabulary.

Multiplying lots of probabilities, which are between 0 and 1 by definition, can result in **floating-point underflow**. To avoid this problem, we use logarithm in formula 4.14:

$$c_{NB} = \arg\max_{c_j \in C} \log P(c_j) \sum_{i \in positions} \log P(x_i|c_j) \tag{4.15}$$

Listing 4.2 shows the pseudo-code of the algorithm.

## 4.2 Algorithm summary

In this section we wrap up the algorithm explained in the previous section.

Listing 4.1: Learner Algorithm

```
TrainMultinomialNB(C,D)

#Extract the set of words that appear in the
   collection of documents D to generate vocabulary V
V = ExtractVocabulary(D)
length_V = length(V)
#compute total number of documents in D
N = CountDocs(D)
```

```
#calculate P(c_j) and P(t_k|c_j) terms, iterating by
   category
for c_j in C:
    #count the number of documents labeled in category
        c_j
    N_j = CountDocsClassJ(D,c_j)
    #computing P(c_j) using formula 4.8
    prior[c_j] = N_j/N

    #build single text that contains all the words of
       all the documents that belong to category c_j
    text_j = TextOfAllDocsInClass(D,c_j)
    #computing number of occurences in text_j
    length_tj = length(text_j)

    for t_k in V:
        #count number of occurences of t_k in Text_j
        F_tk = CountOccurencesofTerm(t,text_j)
        #computing P(t_k|c_j) using smoothing
        condprob[t_k][c_j] =
           (F_tk + alpha)/(length_tj + alpha*length_V)

return V, prior, condprob
```

Listing 4.2: Classification Algorithm

```
ApplyMultinomialNB(C, V, prior, condProb, d)

#Extract from document d the tokens that appear in V
positions = extractPositionsFromDoc(V,d)

#compute probability for each category
for c_j in C:
    score[c_j] = log(prior(c_j))
    for t in positions:
        score[c_j] += log(condprob[t][c_j])

return c_j such that score[c_j] = max(score[c_j])
```

## 4.3 Evaluation measures

Evaluation, **as always**, must be done on test data that are **independent** of the training data. A simple measure that we can use when we have one class per document is the **classification accuracy**:

$$classification\_accuracy = \frac{c}{n} \tag{4.16}$$

Where $n$ is the total number of test instances, and $c$ is the number of test instances correctly classified by the system.

Other useful measures for the binary classification are the **precision** and the **recall** that from the point of view of a spam filter are equal to:

$$Precision = \frac{good\ messages\ kept}{good\ messages\ kept} \tag{4.17}$$

$$Recall = \frac{good\ messages\ kept}{all\ good\ messages} \tag{4.18}$$

We usually have to find a trade-off between these two measures. indeed, for a filter spammer, it is better to have a higher recall than a higher precision because we don't want to risk losing good messages.

Another way to see whether the system works well is by building a confusion matrix with as many rows and columns as the classes in $C$. Each row of the matrix represents the instances in a predicted class, while each column represents the instances in an actual class (or vice versa)[12]. Figure 4.3 shows a confusion matrix for a binary classification task. The name derives from the fact that it is simple to see if the system is confusing two classes.

| | Correct | Incorrect |
|---|---|---|
| Selected | True Positive | False Positive |
| Not selected | False Negative | True Negative |

Figure 4.3: Confusion matrix for binary classification task.

Recall and precision can also be written in terms of the confusion matrix:

$$Precision = \frac{true\ positive}{true\ positive + false\ positive} \tag{4.19}$$

$$Recall = \frac{true\ positive}{true\ positive + false\ negative} \tag{4.20}$$

Since we usually need both precision and recall to assess if a system is adequately working, there exist a measure, called **F measure** that "merge" these measures through a weighted harmonic mean:

$$F_b = \frac{(b^2 + 1)\ precision \cdot recall}{b^2 precision + recall} \tag{4.21}$$

$$F1 = \frac{2\ precision \cdot recall}{precision + recall} \tag{4.22}$$

Where 4.22 is 4.21 with b = 1.

A standard approach when there are more than two classes is 1) for each class $c \in C$ build a binary classifier $Y_c$ to distinguish $c$ from $\neg c$, 2) assign every test document $d$ to each class $c$ for which $Y_c$ returns true. Eventually, we can combine the performance measures into one quantity by using two approaches:

**Macroaveraging**: compute the performance for **each class**, and then take the average.

**Microaveraging**: collect decision for **all classes**, compute a contingency table that sum up all the false/true positives/negatives, and finally evaluate.

**Example 4.3.1** Macroaveraging vs Microaveraging

Let us suppose that we have two classes with the following confusion matrices that summarize the performance of our classifier:

| Classifier | | Truth: Yes | Truth: No |
|---|---|---|---|
| | Yes | 10 | 10 |
| | No | 15 | 970 |

| Classifier | | Truth: Yes | Truth: No |
|---|---|---|---|
| | Yes | 90 | 10 |
| | No | 15 | 890 |

In the **Macroaveraged** approach, we first compute separately the

precision of the classifier on both classes. Using formula 4.19:

$$P(c_1) = \frac{10}{10+10} = 0.5$$
$$P(c_2) = \frac{90}{90+10} = 0.9$$

Finally, we take the average of $P(c_1)$ and $P(c_2)$: $\frac{0.5+0.9}{2} = 0.7$.
In the **Microaveraged** approach, we first sum up the two matrices in one single matrix:

| | | Truth | |
|---|---|---|---|
| | | Yes | No |
| Classifier | Yes | 100 | 20 |
| | No | 30 | 1860 |

And then, we compute the precision using formula 4.19:

$$P(c_1, c_2) = \frac{100}{100+20} = 0.83$$

## 4.4 Tips

The larger is the training set, the better the classification will be, as we can notice looking at figure 4.4. Whenever we don't have much data, one strategy may be to use cross-validation.

We conclude this chapter listing some of the useful properties of the Naïve Bayes model:

- It is robust to irrelevant features, indeed, irrelevant features cancel each other without affecting results.
- Very good in domains with many **equally** important features.
- A good dependable baseline for text classification (but not the best).
- Optimal if the Independence Assumptions hold.

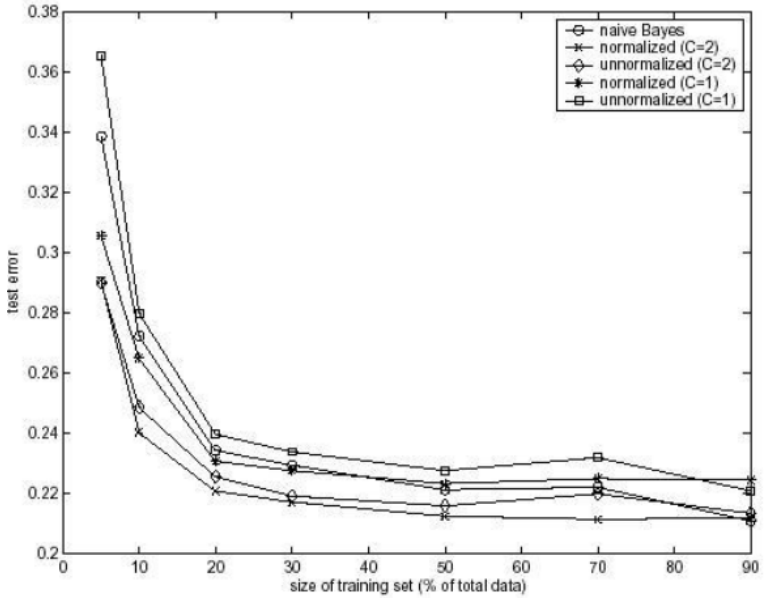



Figure 4.4: The picture shows that as we increase the size of the training set, the test error decrease. Picture from *Improving the Performance of Naive Bayes for Text Classification, Shen and Yang* [15]

.

- It is very fast. Learning with one pass of counting over the data; testing linear in the number of attributes, and document collection size.
- Low storage requirement.
- It is an online algorithm so, we can keep learning from data, adding incrementally new documents.

Chapter 5

# Basic Text Processing

## 5.1 Tokenization

Tokenization is the task of chopping up a document into pieces, called tokens, perhaps at the same time throwing away certain characters, such as punctuation. The major question of the tokenization phase is: **what are the correct tokens to use?**

For instance, you may think that divide words by periods and white spaces is enough, but that is not the case. Indeed think about *U.S.A.* that would be broken into *U*, *S* and *A*, or at *San Francisco* that would become *San* and *Francisco*. However, let us try to write a simple algorithm that performs this naïve tokenization. In particular, we will use the following line of bash code to execute the tokenization:

```
tr 'A-Z' 'a-z'< shakes.txt | tr sc 'A-Za-z' '\n'| sort | uniq -c
    | sort -n -r
```

Where in order, we lower case all the words, then we substitute everything that is not a letter with '\n', then we sort the words, we merge and count the frequency of each different word, and finally, we order the final result according to the frequency. The result is the following:

```
23243 the
22225 i
18618 and
16339 to
```

```
15687 of
12780 a
12163 you
10839 my
10005 in
8954 d
```

Notice that some words are just letters without any sense, such as *d*. Probably this *d* was in words like *I'd love it* or *'you'd that'*, and according to our naïve rule, d is cut off from the rest of the word.

Another difficulty with tokenization is that depending on the language, we can use some strategies but not use others. For instance, in Chinese, words are not separated by white spaces, words are composed of characters which are generally one syllable and one morpheme. The average length of words is 2.4 characters. An algorithm that works astonishingly well in Chinese is the so-called Maximum Matching (also called Greedy). Given a vocabulary of Chinese words, and a string, the algorithm:

1. Start a pointer at the beginning of the string.
2. Find the longest word in the dictionary that matches the string starting at the pointer.
3. Move the pointer over the selected word.
4. Go to 2 until the end of the string.

Unfortunately, this algorithm does not work so well in English. For example, *the table down there* becomes *theta bled own there* because *theta* is longer than *the* and the algorithm search for the longest word in the dictionary (step 2).

### 5.1.1 Terminology

A **Lemma** is a set of lexical forms having the same stem (origin), a major part of speech, and rough word sense. The **Worform** is the full inflected surface form. So, *cat* and *cats* have the same lemma but a different wordform.

The **Type/Form** is an element of the vocabulary. A **Token** is an instance of that type in running text. Church and Gale proved that the size of the vocabulary is approximately the square root of the tokens [4]. Indeed, if you

look at figure 5.1, you will notice that the tokens are many more than the size of the vocabulary $|V|$.

| | Tokens = N | Types = \|V\| |
|---|---|---|
| Switchboard phone conversations | 2.4 million | 20 thousand |
| Shakespeare | 884,000 | 31 thousand |
| Google N-grams | 1 trillion | 13 million |

Figure 5.1: The number of tokens versus the size of the vocabulary.

## 5.2 Word Normalization

Token normalization is the process of canonizing tokens so that matches occur despite superficial differences in the character sequences of the tokens. The most standard way of doing this is through the implicit creation of equivalence classes, e.g., by deleting periods in a term. In alternative we can do asymmetric expansion so that when the user write *windows*, we search both for *window* and *windows*.

In some fields, like information retrieval, it makes sense to lower case all letters even though it is correct or not. Sometimes, this can be a problem: *General Motor* is a company, not a general motor, *Fed* and *CAT* are a government organization. However, it is often best to lower case everything since the user will use lowercase regardless of "correct" capitalization.

### 5.2.1 Lemmatization and Stemming

Lemmatization reduces inflectional variant front to the base form. For instance, we can reduce *am*, *are* and *is* to *be*.

Stemming transforms tokens to their "roots" before indexing. In stemming, we merely take the words, and we make crude affix chopping as show in figure 5.2.

The **Porter's algorithm** is the commonest algorithm for stemming English. It is simply a sequence of phases that consists of a set of rule. In this page, there is the explanation of the entire algorithm and its implementation.

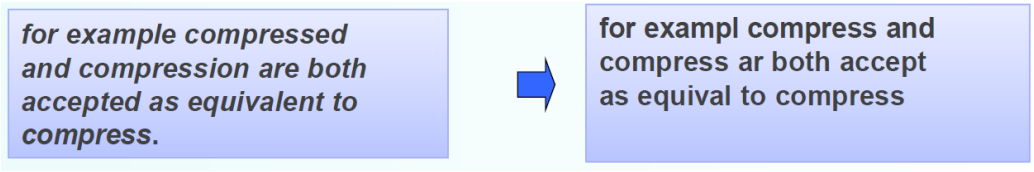



Figure 5.2: Stemming example: we turn example into exampl or compressed into compress

## 5.3 Sentence Segmentation

In sentence segmentation, we try to divide the written text into meaningfull units. There are some symbols such as *!* and *?* that are relatively unambiguous concerning the fact that they represent a separator between words. On the other hand, there are other words like the period . that are quite ambiguous. Indeed, a period can be both a sentence boundary or an abbreviation like Inc. or Dr.

The general idea to solve the problem is by building a binary classifier that looks at a . and decides whether it is an EOF (End of Sentence) or not. We could write a decision tree classifier, but this solution is rarely used. Hand-building is possible only for very simple features/domains.

A more suitable solution is to train a classifier to determine whether a punctuation character is an end of the sentence. For example, **Splitta** is an SVM classifier that does right this.

Nltk provides an already trained splitter:

```
import nltk.data
splitter = nltk.data.load(
                'tokenizers/punkt/english.pickle')

for line in file:
    for sent in splitter.tokenize(line.strip()):
        print sent
```

Chapter 6

# Hidden Markov Model

There are cases in which the order in which words or events appear is relevant. For those cases, the Naïve Bayes classifier is not enough since it does not consider the ordering. A possible solution is by using a hidden Markov model. In this chapter, we will describe what a Markov chain is in section 6.1. Section 6.2 explains the hidden Markov model and the tasks we can perform using that model. We will finally explain a simple use case about speech tagging in section 6.3.

## 6.1 Markov Chain

A Markov chain is a **random** (stochastic) model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event. For instance, a drunk man that at every minute steps backward or forward with a certain probability that depends only on the fact that the man was proceeding back or ahead in the earlier moment is a Markov chain.
A Markov chain can be represented by a **transition** diagram, where:

- each node represents a state.
- Each arc $a_{ij}$ is weighted by a transition probability, that is, how likely a random process will move from state $i$ to state $j$. The sum of all the probabilities leaving any node must sum to one.

Another aspect we have to define is how likely the random process starts from a specific state. We do this by defining a vector, usually indicated with $\pi$, such that $\pi[i]$ designates how likely the process will start in the $i$th state. The sum of all the entries in $\pi$ must sum to one.
So, we can represent the drunk man Markov chain as in figure 6.1.

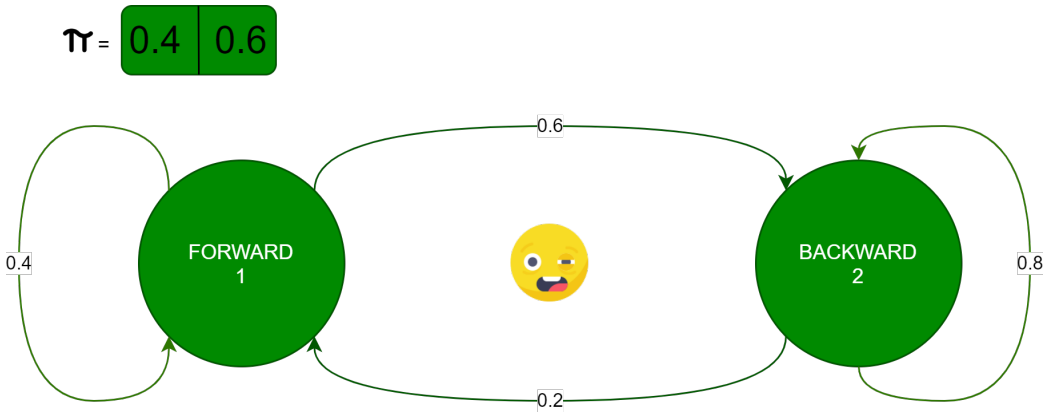



Figure 6.1: The drunk man Markov chain representation. In the beginning, the main will step backward with a probability of 60%, and forward in 40% of the cases. If the man is proceeding forward, he will continue in that direction with a probability of 40%. Otherwise, he will start to go backward with a probability of 60%

Formally, a **first-order observable Markov Model** (aka Markov chain) consists in:

A **set of states** $Q = q_1, q_2, \ldots, q_N$. The state at time t is $q_t$.

A set of **transition probabilities** $A = a_{01}a_{02}\ldots a_{n1}\ldots a_{nn}$. Each $a_{ij}$ represents the probability of transitioning from state $i$ to state $j$:

$$a_{ij} = P(q_t = j | q_{t-1} = i) \; 1 \leq i, j \leq N \tag{6.1}$$

$$\sum_{j=1}^{N} a_{ij} = 1 \; 1 \leq i \leq N$$

Distinguished **start and end states**. As already mentioned, we can use a special probability vector $\pi$ that represents the initial distribution over

probability of start states::

$$\pi_i = P(q_1 = i) 1 \leq i \leq N \tag{6.2}$$

$$\sum_{j=1}^{N} pi_i = 1$$

The model is called first-order observable Markov Model because it uses the first-order **Markov assumption**: *the current state only depends on* ***previous state***:

$$P(q_i|q_1 \ \ldots \ q_{i-1}) = P(q_i|q_{i-1}) \tag{6.3}$$

Let us take another Markov chain we can call **the weather model**. As shown by figure 6.2, there are three states related to the temperature level: cold, warm, and hot. We want to compute the probability to had four consecutive warm days.

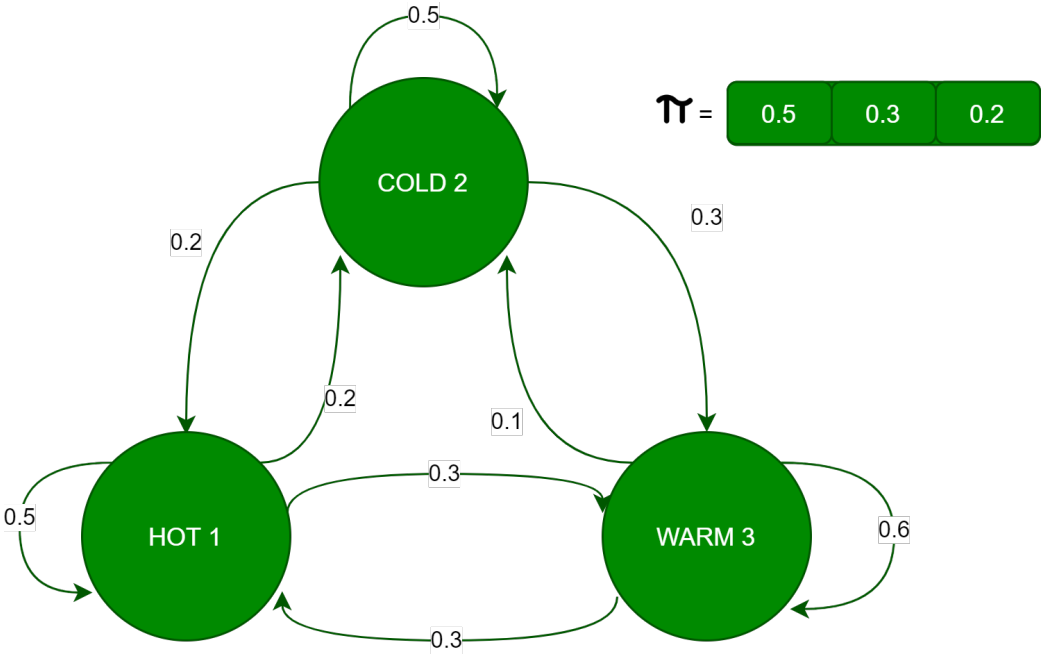



Figure 6.2: The Markov chain of the weather model example

So, we initially see how likely the first day was warm. Then we multiply it with the probability that the day after that warm day was still warm. We

do the same for the other days **using the same transition probability**, and the final result is:

$$P(3,3,3,3) = P(3)\cdot P(3|3)\cdot P(3|3)\cdot P(3|3) = \pi_3 \cdot a_{33} \cdot a_{33} \cdot a_{33} = 0.2 \cdot (0.6)^3 = 0.0432$$

Note that we used the Markov assumption. Otherwise, the formula would have become:

$$P(3,3,3,3) = P(3) \cdot P(3|3) \cdot P(3|3,3) \cdot P(3|3,3,3)$$

That is much more complicated to be computed.

## 6.2 Hidden Markov model

A hidden Markov model (HMM) is an extension of a Markov chain in which **we don't know which state we are in**: we do know the effects of the state through some observations, not directly the state in which we are. That is why we call it "hidden".
A hidden Markov model consists in:

A set of N **hidden states**.

$$Q = q_1 q_2 \dots q_N$$

A **transition probability matrix** $A$, each $a_{ij}$ represents the probability of moving from state i to state j, such that $\sum_{j=1}^{n} a_{ij} = 1 \ \forall i$.

$$A = a_{11} a_{12} \dots a_{n1} \dots a_{nn}$$

A sequence of $T$ **observations** each one drawn from a vocabulary $V = v_1, v_2, \dots, v_V$.

$$O = o_1 o_2 \dots o_T$$

A sequence of **observation likelihood**, also called **emission probabilities**, each expressing the probability of an observation $o_t$ being generated from a state $i$.

$$B = P(o_t|q_i) = b_i(o_t)$$

A special **start state** and an **end state** that are not associated with observations. We represent both states into the matrix $A$ with transition probabilities $a_{01} a_{02} \dots a_{0n}$ out of the start state and $a_{1F} a_{2F} \dots a_{nF}$ into the end state.

$$q_0, \ q_F$$

The model utilizes the Markov assumption:

$$P(q_i|q_1 \dots q_{i-1}) = P(q_i|q_{i-1})$$

and the **output-independence assumption**:

$$P(o_t|O_1^{t-1}, q_1^t) = P(o_t|q_t) \tag{6.4}$$

That states that the likelihood of observation at time $t$ depends only on the hidden state at time $t$. Indeed, we assume that, given the state $q_t$, $o_t$ is conditional independent both of all the earlier observations from time 1 to time $t-1$ ($O_1^{t-1}$) and of all the states from time 1 to time $t-1$ ($q_1^{t-1}$).

Let us practically see how is a hidden Markov model. Suppose you are a climatologist in the year 2799 studying global warming. You cannot find any records of the weather in Pisa for the summer of 2021, but you find Jason Eisner's diary that lists how many gelatos Jason ate every date that summer. Our job may be to figure out if the day was cold or hot, so, given a sequence of gelato observations: $1, 2, 3, 2, 2, 2, 3, \dots$ that indicate how many gelatos Jason ate for each day, we want to produce a weather sequence like $H, C, H, H, H, C \dots$.
The hidden Markov model of this problem is pictured in figure 6.3. In ad-

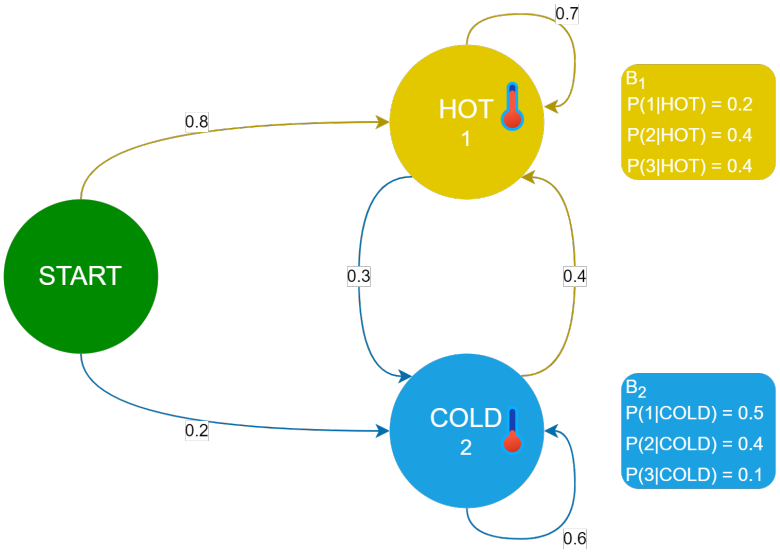



Figure 6.3: The Markov chain of the weather model example

dition to the transition probabilities and the start state we already have in

the Markov model, every state is associated with a table B that describes the observation likelihood for that particular state. So, if it was a hot day, Jason ate one gelato with a probability of 0.2, two gelatos with a probability of 0.4, and so on.

Figure 6.3 shows an example of **Ergodic** structure, that are fully-connected structures. As an example, these structures have been used in modeling the web for the page-rank algorithm. On the other hand, we have left-to-right structures, called **Bakis** (see figure 6.4).

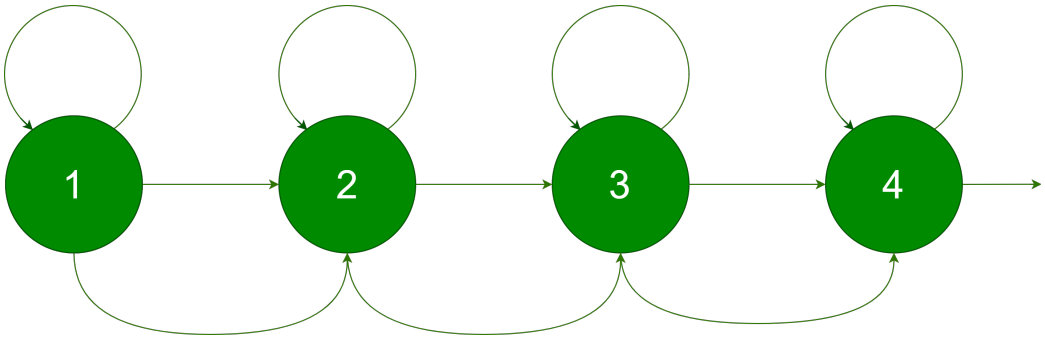



Figure 6.4: Bakis structure example

The three principal problems we can face using a hidden Markov model are:

**Evaluation**: given the observation sequence $O = (o_1 o_2 \dots o_T)$, and an HMM model $\Phi = (A, B)$, **how do we efficiently compute** $P(O|\Phi)$, that is the probability of the observation sequence, given the model.

**Decoding**: given the observation sequence $O = (o_1 o_2 \dots o_T)$, and an HMM model $\Phi = (A, B)$, **how do we choose a corresponding state sequence** $Q = (q_1 q_2 \dots q_T)$ that is optimal in some sense (i.e., best explains the observations). This is for instance the problem described in the gelato example.

**Learning**: **how we adjust the model parameters** $\Phi = (A, B)$ to maximize $P(O|\Phi)$.

Let us now see two algorithms to deal with respectively evaluation and decoding.

### 6.2.1 Evaluation - The forward algorithm

In evaluation, given an HMM $\Phi = (A, B)$ and an observation sequence $O = o_1 o_2 \dots o_T$ of length $T$, our job is to determine the likelihood $P(O, \Phi)$.

First, we need to know how to compute $P(O|\Phi)$. It is pretty simple to compute the likelihood of a sequence of observations $O = o_1 o_2 \dots o_T$ given the hidden state sequence $Q = q_1 q_2 \dots q_T$. Indeed, using the output-independent assumption (6.4) we have that:

$$P(o_1 o_2 \dots o_T | q_1 q_2 \dots q_T) = P(o_1|q_1)P(o_2|q_2) \dots P(o_T|q_T) = \prod_{i=1}^{T} P(o_i|q_i) \tag{6.5}$$

Note that the hidden state sequence and the observation sequence have the same length $T$. That is because there is a one-to-one mapping between the observation sequence and the hidden state sequence, which comes from the fact that each hidden state emits only a single observation.

To have 6.5 into $P(O|\Phi)$ we can marginalize with respect to all the possible hidden state sequences, getting:

$$P(O, \Phi) = \sum_{Q} P(O, Q) \tag{6.6}$$

Where $Q$ is the set of all the possible hidden state sequences of length T we can obtain in the HMM $\Phi$. For instance, in the HMM shown by figure 6.3, HOT HOT COLD is an element of Q if the observation sequence has a length of 3.
Using the chain rule (2.1), 6.6 turns into the following formula:

$$\sum_{Q} P(O, Q) = \sum_{Q} P(O|Q) \times P(Q) \tag{6.7}$$

And the first factor is indeed our beloved $P(O|Q)$ in 6.5. Concerning $P(Q)$, we can apply the chain rule (2.1) and then use the Markov assumption(6.3):

$$\begin{aligned} P(Q) &= P(q_T|q_1, \dots, q_{T-1}) \times P(q_{T-1}|q_1, \dots, q_{T-2}) \times \dots \times P(q_1) && (6.8) \\ &= P(q_T|q_{T-1}) \times P(q_{T-1}|q_{T-2}) \times \dots \times P(q_1) \\ &= \prod_{i=1}^{T} P(q_i|q_{i-1}) && (6.9) \end{aligned}$$

The final formula thus is:

$$P(O|\Phi) = \sum_{Q} \prod_{i=1}^{T} P(o_i|q_i) \times \prod_{i=1}^{T} P(q_i|q_{i-1}) \tag{6.10}$$

So, a simple solution would be plainly compute the formula 6.10 since we have both $P(o_i|q_i)$ that is $b_i(o_i)$, and $P(q_i|q_{i-1})$ that is $a_{i,i-1}$. But in a HMM with $N$ hidden states, the number of possible sequences is $N^T$. Hence, the cost of the algorithm will grow exponentially as we increase the size of the sequence.

**Example 6.2.1**

Given the HMM in figure 6.3, we want to compute the likelihood of the sequence 3 1 3. We should consider all the possible sequences, but let just consider the sequence HOT HOT COLD. We have to compute:

$$P(3\ 1\ 3,\ HOT\ HOT\ COLD)$$

And according to 6.10:

$$\begin{aligned} &P(3\ 1\ 3,\ HOT\ HOT\ COLD) \\ &= P(3|HOT) \times P(1|HOT) \times P(3|COLD) \\ &\times P(HOT|START) \times P(HOT|HOT) \times P(COLD|HOT) \end{aligned}$$

That is equal to:

$$\begin{aligned} &b_1(3) \times b_1(1) \times b_2(3) \times a_{0,1} \times a_{1,1} \times a_{1,2} \\ &= 0.4 \times 0.2 \times 0.1 \times 0.8 \times 0.7 \times 0.3 \end{aligned}$$

The **forward algorithm** computes 6.10 using a **dynamic programming** fashion in $O(N^2T)$. The problem of the Naïve solution is that it computes many times the same thing. On the other hand, the forward algorithm computes everything one single time by folding all the sequences into a single **trellis**.
The algorithm computes $P(O|\Phi)$ incrementally: on step $t$, it computes:

$$P(o_1, o_2, \ldots o_t, q_t = j|\Phi)\ 1 \leq j \leq N \tag{6.11}$$

Where $q_t = j$ means the probability that the $t$th state in the hidden state sequence is j. By induction, we have that 6.11 is equal to:

$$\sum_{i=1}^{N} P(o_1, o_2, \dots o_{t-1}, q_{t-1} = i|\Phi) \times P(q_t = j|q_{t-1} = i) \times P(o_t|q_t = j) \quad (6.12)$$

Where the first factor is the information the algorithm calculated at the previous step that we will indicate with $\alpha_{t-1}(i)$, $P(q_t = j|q_{t-1} = i)$ is the transition probability $a_i, j$ from previous state $q_i$ to current state $q_j$ and $P(o_t|q_t)$ is the observation likelihood of the observation $o_t$ given the current state $j$:

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) \cdot a_i, j \cdot b_j(o_t) \quad 1 \leq j \leq N, 1 \leq t \leq T \quad (6.13)$$

Note that $\alpha$ is a matrix and that $\alpha_t(j) = \alpha[j, t]$.

The algorithm **initializes** the first column of the matrix by calculating:

$$\alpha_1(j) = a_0, j \cdot b_j(o_1) \; 1 \leq j \leq N \quad (6.14)$$

Figure 6.5 shows the initialization step if the first element of the observation sequence is 3.

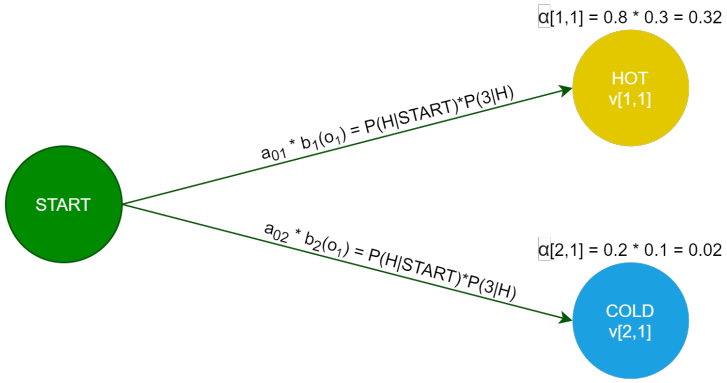



Figure 6.5: Example of initialization step of the forward algorithm.

Then the algorithm continues applying 6.13 as shown in figure 6.6. And once it computes the probabilities for $t = T$, it can lastly compute $P(O|\Phi)$:

$$P(O|\Phi) = \sum_{i=1}^{N} \alpha_T(q_f) = \sum_{i=1}^{N} \alpha_T(i) a_{i,F} \quad (6.15)$$

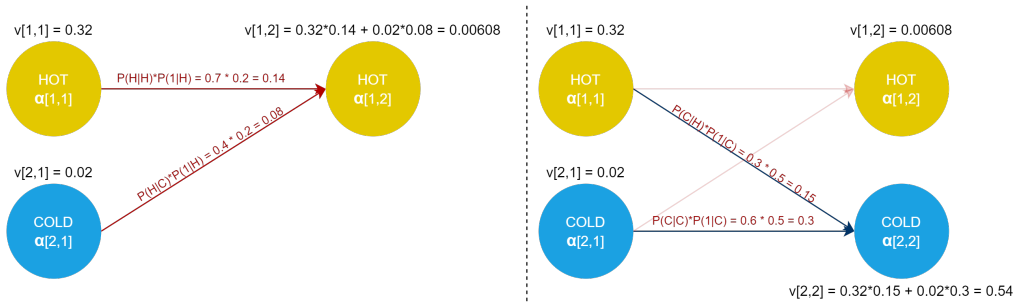



Figure 6.6: Second step of the algorithm if the second observation is 1.

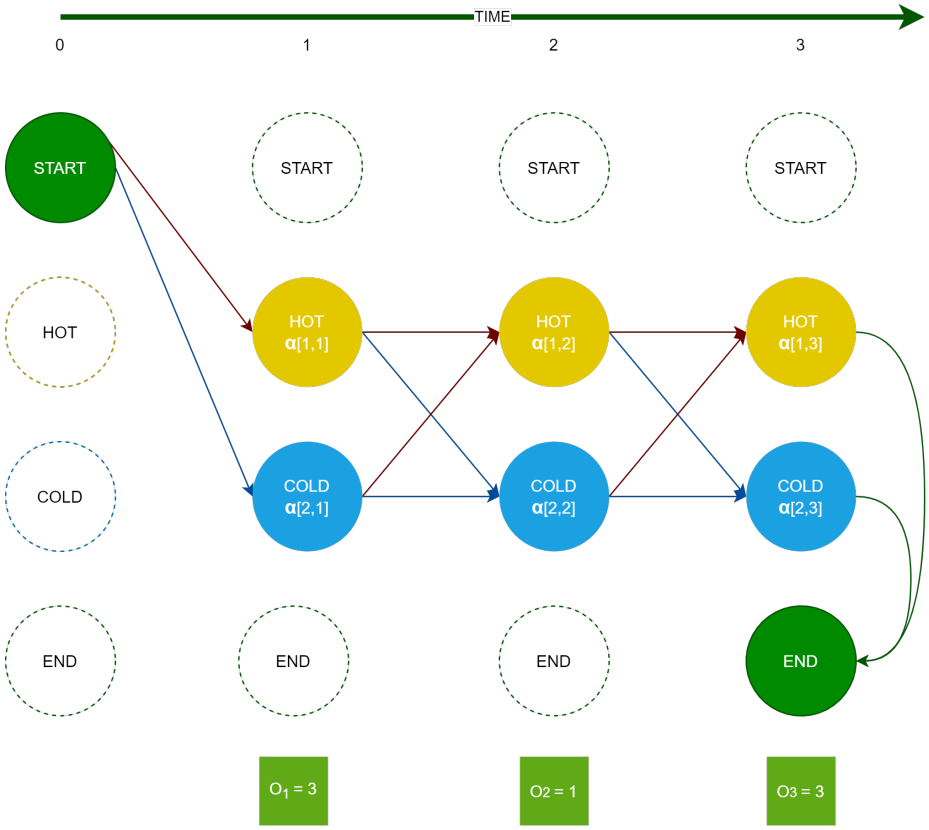



Figure 6.7: The picture shows the forward algorithm on its whole. Every $\alpha$ is computed precisely one time, avoiding the problem of the naïve solution.

### 6.2.2 Decoding - The Viterbi algorithm

Given an observation sequence (e.g. 3 1 3), and an HMM $\Phi = (A, B)$, the task of the **decoder** is to find the best **hidden** state sequence (e.g. HOT COLD HOT). A Naïve decoder would try each state sequence $Q$ and return the one that maximize $P(q_0, q_1, q_2, \ldots, q_T, o_1, o_2, \ldots, o_T, q_T = q_F|\Phi)$. This solution would not scale so much since we should try $N^T$ different combinations ($N$ is the number of hidden states).

A much better algorithm is the **Viterbi algorithm** that uses dynamic programming techniques to reduce the computations and make the problem resolution more feasible. The algorithm during the computation fills a matrix `v` such that `v[s,t]` contains the maximum likelihood over all the paths of length `t` that brings to state `j`. Formally:

$$v[s,t] = \max_{q_0,q_1,q_2,\ldots,q_{t-1}} P(q_0, q_1, q_2, \ldots, q_{t-1}, o_1, o_2, \ldots, o_t, q_t = s|\Phi) \tag{6.16}$$

At step $t-1$, the algorithm calculates the $t-1$th column of $v$ so that, at step $t$, it can use the previous calculations to fill the $t$th column:

$$v[s,t] = \max_{s'=1}^{N} v[s', t-1] \cdot a_{s',s} \cdot b_s(o_t) \tag{6.17}$$

Where the three factors are:

$v[s', t-1]$ the previous Viterbi path probability from step $t-1$.

$a_{s',s}$ the transition probability from state $q_{s'}$ to state $q_s$.

$b_s(o_t)$ the observation likelihood of the observation $o_t$ given the current state $q_s$.

So the algorithm recursively takes the likelihood of the most probable extension of the paths that lead to the current cell. In addition to the matrix $v$, the algorithm uses another matrix $b$ of back-pointers that it eventually uses to backtrace the path:

$$b[s,t] = \arg\max_{s'=1}^{N} v[s', t-1] \cdot a_{s',s} \tag{6.18}$$

Recall that max returns the maximum value while arg max returns the index $i$ for which we get the maximum value.

The initialization step fills the first column of $v$ and $b$ using the following formulas:

$$v[s,1] = a_{0,s} \cdot b_s(o_1) \ 1 \leq s \leq N \tag{6.19}$$
$$b[s,1] = 0 \ 1 \leq s \leq N \tag{6.20}$$

Figure 6.8 shows the initialization step with our gelato example where on the first day, Jason ate three gelatos.

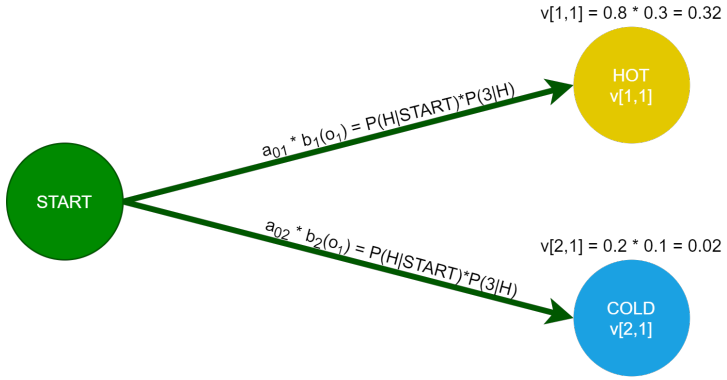



Figure 6.8: Initialization step of the rows, in the first column, relative to the HOT and COLD states. We used formula 6.19 to calculate v[1,1] and v[2,1]. Note also that both b[1,1] and b[2,1] are 0.

Then, the algorithm continues filling the columns of the matrices using formulas 6.17 and 6.18. Figure 6.9 shows the calculations in the case the second day Jason ate one gelato.

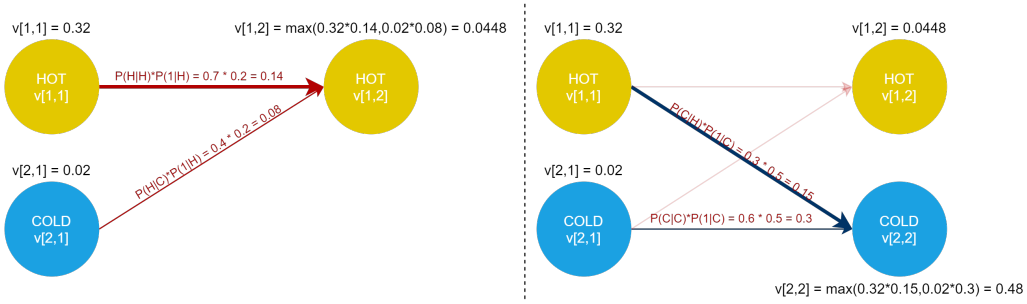



Figure 6.9: Second step of the Viterbi algorithm. Note that both `b[1,2]` and `b[2,2]` are 1.

In the end, we compute the best score and the start of the backtrace in the final state $q_F$:

$$v[q_F, T] = \max_{s=1}^{N} v[s, T] \cdot a_{s,q_F} \tag{6.21}$$

$$b[q_F, T] = \arg\max_{s=1}^{N} v[s, T] \cdot a_{s,q_F} \tag{6.22}$$

The result is the backtrace path we obtain by following the back-pointers from $b[q_F, T]$ back to the start.

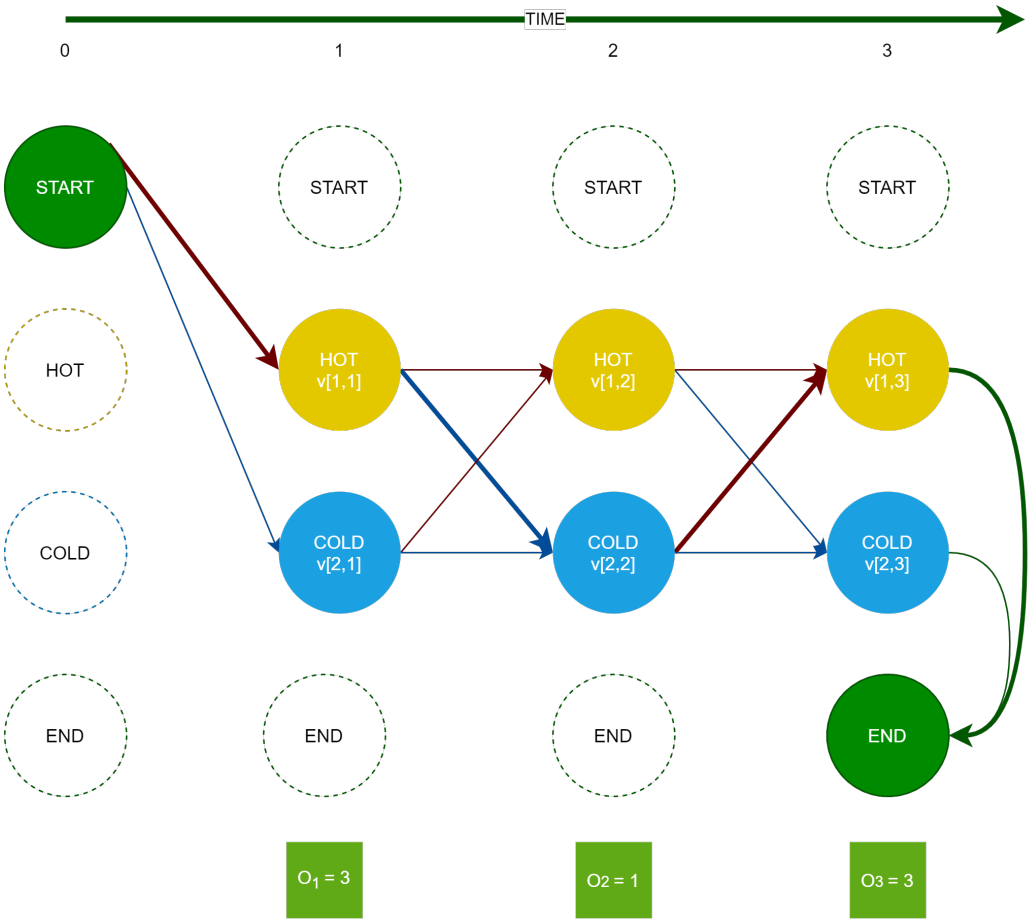



Figure 6.10: The picture shows the Viterbi algorithm on its whole. In the end, the algorithm follows the thicker arrows (using the back-pointers) and returns the path H, C, H.

## 6.3 Speech tagging

Speech tagging is the task of assigning to every word a part-of-speech tag. For instance, given a phrase such as *The koala puts the keys on the table*, we want to find the best sequence of tags that corresponds to this sequence of words: DET NOUN VERB DET NOUN PREP DET NOUN. From an HMM point of view, we can see words as observations and tags as hidden states. Our goal is to sick for the most likely sequence of tags (the hidden state sequence) given the observation sequence of words ($w_1^n$):

$$\hat{t}_1^n = \arg\max_{t_1^n} P(t_1 \dots t_n | w_1 \dots w_n) = \arg\max_{t_1^n} P(t_1^n | w_1^n) \tag{6.23}$$

The hat means "our estimate of the best one". 6.23 is made operational by using the Bayes rule (4.1) that transforms it into a set of other probabilities that are easier to compute:

$$\hat{t}_1^n = \arg\max_{t_1^n} \frac{P(w_1^n | t_1^n) P(t_1^n)}{P(w_1^n)}$$

Since $P(w_1^n)$ does not change for each tag, we can drop it:

$$\hat{t}_1^n = \arg\max_{t_1^n} \underbrace{P(w_1^n | t_1^n)}_{\text{likelihood}} \underbrace{P(t_1^n)}_{\text{prior}} \tag{6.24}$$

That can be simplified even more by making some assumptions. The first assumption is the **Naïve Bayes assumption** that is the probability of a word appearing depends only on its part-of-speech tag:

$$P(w_1^n | t_1^n) \approx \prod_{i=1}^{n} P(w_i | t_i)$$

The second assumption is the Markov assumption (6.3):

$$P(t_1^n) \approx \prod_{i=1}^{n} P(t_i | t_{i-1})$$

That says that the probability of a tag depends only on the tag just before. Plugging these simplifications into 6.24 give us the following equation:

$$\hat{t}_1^n \approx \prod_{i=1}^{n} P(w_i | t_i) \cdot P(t_i | t_{i-1}) \tag{6.25}$$

Where, in terms of an HMM, $P(w_i|t_i)$ is the observation likelihood $b_i(o_i)$ and $P(t_i|t_{i-1})$ is the transition probability from previous state $t_{i-1}$ to current state $t_i$.

We can compute both terms in 6.25 by a maximum likelihood estimate. The first term is the probability to read in position $i$ a word, kenning that in $i$ there is a specific tag $t_i$. For instance, knowing that in position $i$ there is a verb, we want to know how likely "is" is in that position. So, we can take a corpus in which parts-of-speech are labeled and then count how many times "is" appears with the tag "VERB" and divide by the total times in which the tag is "VERB". For a general word $w_i$ and tag $t_i$ that is equal to:

$$P(w_i|t_i) = \frac{C(t_i, w_i)}{C(t_i)} \tag{6.26}$$

For our example, in the 45-tag Treebank Brown corpus:

$$P(is|VERB) = \frac{C(VERB, is)}{C(VERB)} = \frac{10,073}{21,627} = 0.47$$

The second term of 6.25 is the probability of a tag given the previous one. Also, in this case, we can think of counting occurrences. If $t_i$=NOUN and $t_{i-1}$=ADJ, we should compute how many times, in our corpus, NOUN appears after ADJ and then divide by the times we see ADJ. Generalizing, that is equal to the following ratio of counts:

$$P(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i}{C(t_{i-1})} \tag{6.27}$$

For our example, in the 45-tag Treebank Brown corpus:

$$P(NOUN|ADJ) = \frac{C(ADJ, NOUN)}{C(ADJ)} = \frac{56,509}{116,454} = 0.49$$

**Example 6.3.1** Disambiguate the word race

The word *race* can be either a noun (NN) or a verb (VB). This example will show that for a particular sentence, the correct tag sequence achieves a higher probability than one of the many possible wrong sequences. Let us, for instance, consider this phrase:

- Secretariat/NNP is/BEZ expected/VBN to/TO race/VB tomorrow/NR

Suppose we have been calculating the tags till the word just before *race*. Hence, we have to compute the tag for *race*. We will consider only the tags NN and VB. However, there are many other possible tags we should consider.
For this example, we will use the 87-tag Brown and Switchboard corpus.
Applying 6.27 with word *race* and tags *NN* and *VB*:

$$P(race|NN) = 0.00057$$
$$P(race|VB) = 0.00012$$

Which are the probabilities that, inside the Brown corpus, the word race is a noun or a verb.
We now have to compute the transition probabilities:

$$P(NN|TO) = 0.00047$$
$$P(VB|TO) = 0.83$$

Which are the probabilities of transitioning from state TO to states NN or VB (tags for *race*).

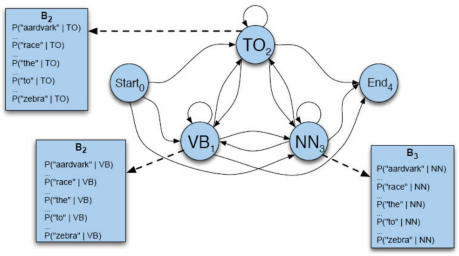
$$P(NR|VB) = 0.0027$$
$$P(NR|NN) = 0.0012$$

Which are the probabilities of transitioning from states VB or NN to state NR (tag for *tomorrow*). We can ultimately use 6.23 and gather that the sequence with the VB tag is more probable than the one with the NN tag:

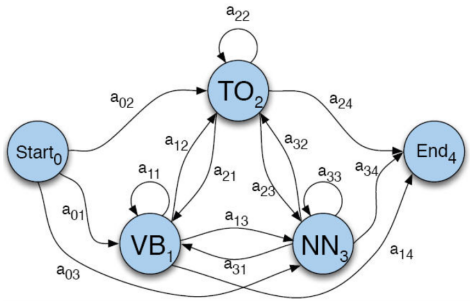
$$P(VB|TO)P(NR|VB)P(race|VB) = 0.00000027$$
$$P(NN|TO)P(NR|NN)P(race|NN) = 0.00000000032$$

To summarizing, we can use 6.27 to calculate the transition probabilities of our HMM, and 6.26 to fill the matrix of observation likelihoods. Figure 6.10a shows an HMM where the tags are TO, VB and NN, while the words

are ”aardvark”, ”race”, ”the”, ”to”, and ”zebra”.




(a) The B observation likelihoods




(b) The transition probabilities

Figure 6.10: HMM for speech tagging.

Once we have defined the HMM, we can exploit the Viterbi algorithm to solve the speech tagging problem.

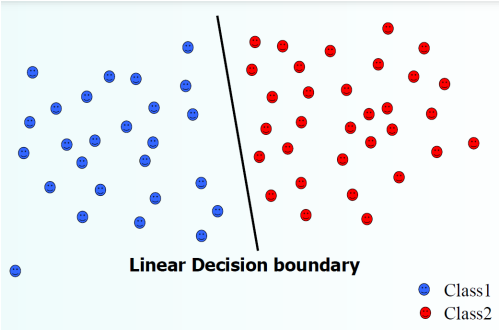
Chapter 7

# Neural Network Classifier

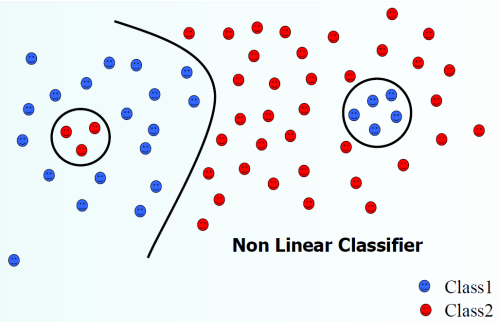
## 7.1 Towards non linearly separable problems

Consider having two sets of points labeled as blue and red points. The two sets are linearly separable if there exists at least one hyperplane with all the blue points on one side of the hyperplane and all the red points on the other side. Figure 7.0a shows an example of linearly separable data. On the other hand, we say that the two sets are not linearly separable if there is not any hyperplane with such property as in figure 7.0b.

We cannot know in advance if a set is linearly or non linearly separable, so




(a) Linearly separable data




(b) Non linearly separable data

Figure 7.0: Examples of linearly and non linearly separable data. Note that in the second case, it is impossible to find a line that divides the points into red and blue sets

usually, we must discover it empirically. Whenever we know that the boundary can be linear, we must use a linear algorithm such as the Naïve Bayes or the Perceptron, since they are more manageable and requires fewer parameters. Unfortunately, high dimensional data (like for NLP) is usually not linearly separable. If that is the case we need to use non linear algorithms even though they are more complicated and has more parameters. For instance, a standard approach is to transform the non linear data into linear data by using a Kernel.

## 7.2 Perceptron

A perceptron (or neuron) is the basic unit of a neural network. Every perceptron has four basic elements:

- A set of **synapses** or weights $w$. Each synapse receive a signal $x_i$, which comes from another neuron, and that is multiplied by the synaptic weight $w_i$.
- A **bias** that has the effect of increasing/decreasing the net input (see next point). The bias is usually referred to as $w_0$ and receives a signal $x_0$ equal to 1.
- An **adder** that computes the weighted sum of the signals. The output is called net:
$$z = \sum_i x_i w_i = wx + b \tag{7.1}$$
Where $w$ is the weight vector, $x$ is the input vector and $b$ is the bias.
- An **activation function** that takes as input the net. The activation function is fundamental to apply a non-linear transformation to the net and thereby learn non-linear functions. It is also used for limiting the amplitude of the output of the perceptron. Indeed, it is also called "squashing" function.
$$y = f(z) \tag{7.2}$$

Every neural network usually has several layers of neurons (at least one, but often more than three). Every layer has no connection within it and is fully connected with the adjacent layers. Picture 7.2 shows a neural network with n input units, two hidden layers, and a final output layer with three

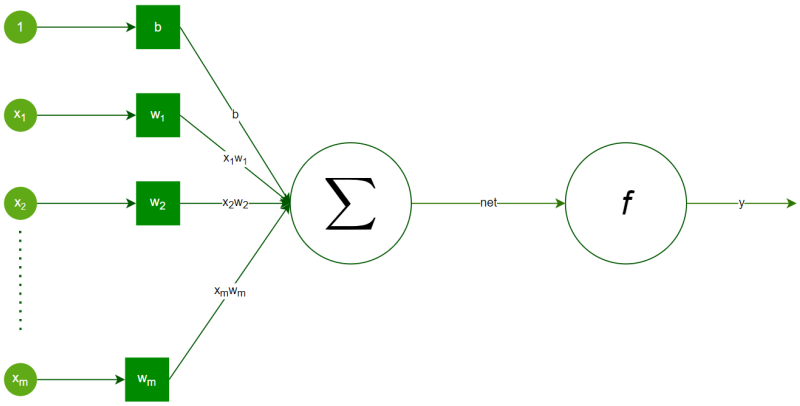



Figure 7.1: Perceptron structure

neurons. As shown, the number of output units is not necessarily equal to the number of input units, and the number of hidden units per layer can be more or less than input or output units. Each unit is a perceptron that computes:

$$y_i = f(\sum_{j=0}^{m} w_{ij}x_j) \tag{7.3}$$

Where $w_ij$ is the weight for the $j$th input in the $i$th unit. Note also that $w_{0j}$ is the bias of the $i$th unit, and $x_0$ is 1.

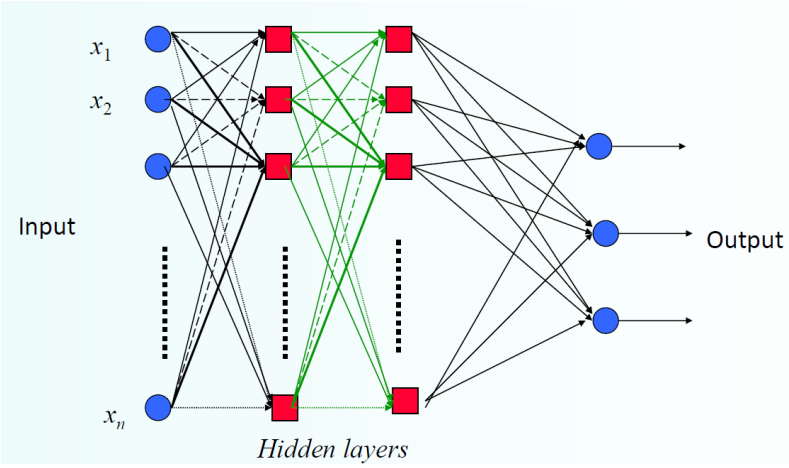



Figure 7.2: Neural network with two hidden layers

The more are the layers, the more complex are the boundaries that the neural network can generate. However, this only holds if the activation function used in the neurons applies a non-linear transformation to the net. If we merely use the linear activation function, the extra layer could be compiled down into a single linear transformation.

A neural network is a universal approximator. Indeed, the universal approximation theorem states that: a 2-layer network can approximate any continuous function.

### 7.2.1 Perceptron Learning Rule

Rosenblatt's perceptron has been the first-ever. It could automatically learn categories or classify input vectors into types. The activation function used was the threshold function:

$$f(z) = \begin{cases} +1, & \text{if } z \geq 0 \\ -1, & \text{otherwise} \end{cases} \tag{7.4}$$

The perceptron weights can be adapted using an error correction rule known as **perceptron convergence algorithm**. The algorithm goes over all existing data patterns whose labeling is known and checks their classification with the current weight vector. If it classifies correctly, it steps on the next sample. Otherwise, it adds to the weights a quantity proportional to the product of the input pattern with the desired output $y$ ($+1$ or $-1$):

Listing 7.1: Perceptron convergence algorithm

```
Initialize the weights (to small random values/zero)
Pick a learning rate epsilon

Until stopping criterion is satisfied
    Pick training pattern <xi,yi>
    if f(z) = yi              #xi is not missclassified
        do nothing
    otherwise
        w = epsilon*x_i*(yi - f(x))
```

The learning rule used is:

$$w_{i+1} = w_i + \Delta w_i \tag{7.5}$$
$$\Delta w_i = \epsilon x_i (y_i - \hat{y}_i)$$

Where $\epsilon$ is the learning rate which indicates how much of the error to use in the correction, $y_i$ is the expected output while $\hat{y}_i$ is the computed output. We can rewrite 7.5 as follow:

$$w_{i+1} = w_i - \epsilon \cdot x_i \quad \text{if } \hat{y}_i > 0 \text{ and } y_i \leq 0 \tag{7.6}$$
$$w_{i+1} = w_i + \epsilon \cdot x_i \quad \text{if } \hat{y}_i \leq 0 \text{ and } y_i > 0 \tag{7.7}$$

## 7.3 Softmax classifier

**logistic regression** is a model used to assign a probability to a set class of events such as alive/dead or above/below. The basic form of the logistic regressor uses a sigmoid/logistic activation function (7.8) to model a binary dependant variable.

$$\sigma(w \cdot x) = \frac{e^{w \cdot x}}{1 + e^{w \cdot x}} \tag{7.8}$$

Given a binary classification task where the output can be $y = \{1, -1\}$, an input $x$ is classified $\hat{y}$ if the probability of $x$ belonging to $\hat{y}$ is higher than the probability of $x$ belonging to all the other classes:

$$\hat{y} = \arg\max_y P(y|x) = \arg\max_y \sigma(w \cdot x) \tag{7.9}$$

We can rewrite 7.9 in the following manner:

$$\hat{y} = 1 \text{ iff } P(y = 1|x) > P(y = -1|x)$$

That is like saying that the ratio between $P(y = 1|x)$ and $P(y = -1|x)$ is greater than 1:

$$\frac{P(y = 1|x)}{P(y = -1|x)} > 1 \quad \leftrightarrow \quad \frac{P(y = 1|x)}{1 - P(y = 1|x)} > 1$$

Substituting $P(y = 1|x)$ with the definition of the sigmoid function:

$$\frac{\frac{e^{w \cdot x}}{1+e^{w \cdot x}}}{1 - \frac{e^{w \cdot x}}{1+e^{w \cdot x}}} = \frac{\frac{e^{w \cdot x}}{1+e^{w \cdot x}}}{\frac{1}{1+e^{w \cdot x}}} = e^{w \cdot x} > 1$$

In the case of multiple classes, we associate a weight $w_c$ to each class $c$. The generalized version of the sigmoid function is the softmax function (3.5):

$$P(c|x) = \frac{e^{w_c \cdot x}}{\sum_{c' \in C} e^{w_{c'} \cdot x}} = \text{softmax}_c(z) \tag{7.10}$$
$$z = (w_1 x, \ldots, w_C x)$$

The softmax function takes a vector of real numbers and normalizes it into a **probability distribution**.
We usually represent $x$ as a vector of features $f_i$:

$$P(c|x) = \frac{e^{w_c \cdot f}}{\sum_{c' \in C} e^{w_{c'} \cdot f}} = \text{softmax}_c(w_c \cdot f)$$

We can find those features either by choosing them by hand (classical approach) or generating them, as in deep neural networks. For instance, we could put a layer in the neural network that map $x$ into $f$ by following several rules that indicate whether $f_i$ is 1 or 0:

$$f_1(c, x) = \begin{cases} 1, & \text{if } word_i = \text{"race"} \wedge c = \text{NN} \\ 0, & \text{otherwise} \end{cases}$$

$$f_2(c, x) = \begin{cases} 1, & \text{if } t_{i-1} = \text{TO} \wedge c = \text{VB} \\ 0, & \text{otherwise} \end{cases}$$

$$f_3(c, x) = \begin{cases} 1, & \text{if suffix}(word_i) = \text{"ing"} \wedge c = \text{VBG} \\ 0, & \text{otherwise} \end{cases}$$

$$\ldots$$

$$f_{125}(c, x) = \begin{cases} 1, & \text{if } word_{i-1} =< s > \wedge \text{isupperfirst}(word_i) \wedge c = \text{NNP} \\ 0, & \text{otherwise} \end{cases}$$

The training phase consists of maximizing the probability of output the correct class $y$ by using a gradient iterative scaling. That is equivalent to minimizing the negative log probability of that class:

$$-log(p(y|x)) = -\log(\frac{e^{w_c \cdot x}}{\sum_{c' \in C} e^{w_{c'} \cdot x}})$$

Using log probability, we also convert our objective function to sums, which is easier to work with on paper and implementation.

### 7.3.1 Cross entropy

Let the true probability distribution be $p$ and the computed model probability be $q$, the cross entropy is:

$$H(p,q) = \sum_c p(c)\log(q(c)) \tag{7.11}$$

Assuming a ground truth probability distribution that is 1 at the right class and 0 everywhere else:

$$p = [0, \ldots 0, 1, 0, \ldots, 0]$$

then, because of one-hot $p$, the only term left in 7.11 is the negative log probability of the true class. So maximizing the likelihood is the same as minimizing the cross entropy.

## 7.4 Again on POS tagging

In section 6.3, we saw that we could solve the POS tagging task using an HMM and the Viterbi algorithm (6.25). Another way to solve the problem is through a logistic regressor that takes as input a word and its context window of size $k$, and returns the corresponding tag (see figure 7.3):

$$\hat{t}_i \approx \arg\max_{t_i} P(t_i | w_{i-k}, \ldots, w_{i+k}) \tag{7.12}$$

A better input features are usually the **categories** of the surrounding tokens. We can use categories of either preceding or succeeding tokens by going forward or back and using previous output as shown in figure 7.4.

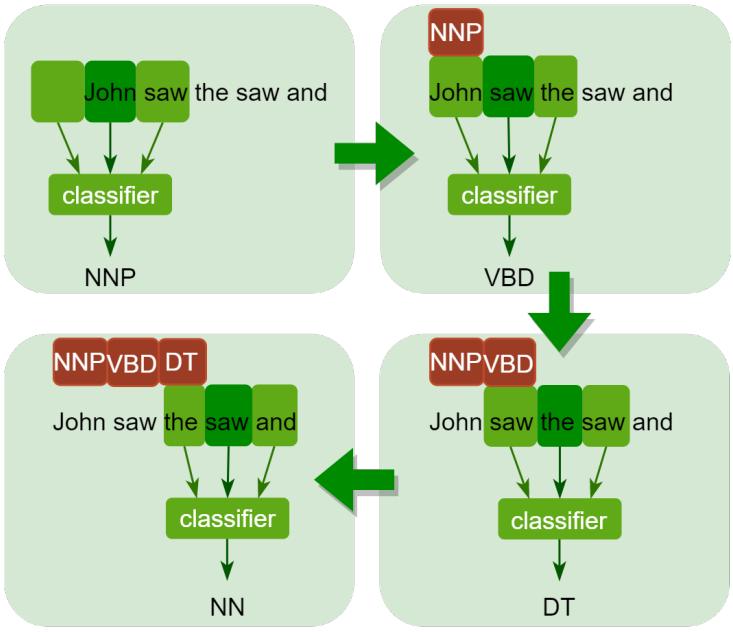



Figure 7.3: Classify each token independently but use, as input feature, information about the sorrounding tokens (sliding window)

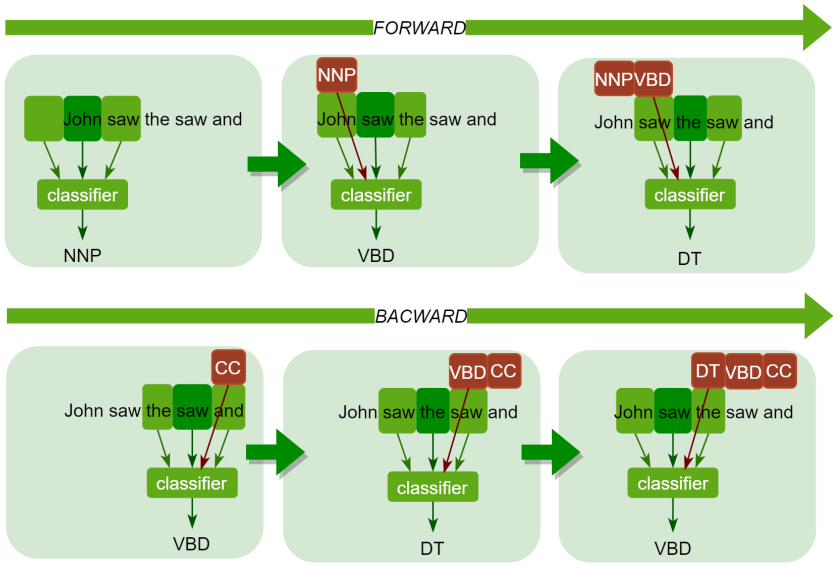



Figure 7.4: Classify each token by also considering the tag associated to previous tokens

HMM and logistic regression represents two different types of classification:

**Generative Model**: computes $P(x|y)$ instead of $P(y|x)$ employing the Bayes rule:

$$\hat{y} = \arg\max_y P(y)P(x|y)$$

$P(x|y)$ is generating because it tells us the probability of the input given the output, so we can use it to generate inputs. The Naïve bayes and the HMM are generative models.

**Discriminative Model**: directly estimates the probability of the output given the input:

$$\hat{y} = \arg\max_y P(y|x)$$

The logistic regression, the maximum entropy Markov model (see next section), or the support vector machine are discriminative models.

The most critical limit of the logistic regression is that it estimates the tag without considering the whole phrase. So, it will find the local optimum but could not find the global optimum. On the other hand, probabilistic sequence models, as the HMM, allow integrating uncertainty over multiple, interdependent, classifications and collectively determine the most likely global assignment. Besides the HMM, other models that use this approach are the Maximum Entropy Markov Models (MEMM) and the Recurrent Neural Networks (RNN). that use this approach are the Maximum Entropy Markov Models (MEMM) and the Recurrent Neural Networks (RNN).

### 7.4.1 Maximum entropy Markov model (MEMM)

A MEMM is a **discriminative model** that extends a standard **maximum entropy classifier** (logistic regressor) by assuming that the unknown values to be learned are connected in a Markov chain. As shown in Figure 7.5 the difference between an HMM and a MEMM is that while in the HMM, the hidden states produce the output, in the MEMM, that relation is inverted.

The MEMM computes the overall sequence of tags given all the input of words:

$$P(q_1, \ldots, q_N|o_1, \ldots, o_N) \tag{7.13}$$

That using the chain rule and then the Markov assumption turns into:

$$P(q_1, \ldots, q_N | o_1, \ldots, o_N) = \prod_t^N P(q_t | qt-1, o_t) \tag{7.14}$$

The probability of a certain label $s$ is modeled in the same way as a maximum entropy classifier:

$$P(q_t | qt-1, o_t) = \frac{e^{w \cdot f(o,q)}}{\sum_{o,q'} e^{w \cdot f(o,q')}} = \text{softmax}_{o,q}(w \cdot f(o,q)) \tag{7.15}$$

Where $f(o,q)$ is a vector of features and $w$ are the corresponding weights. In order to maximize 7.14, we use again the Viterbi algorithm.

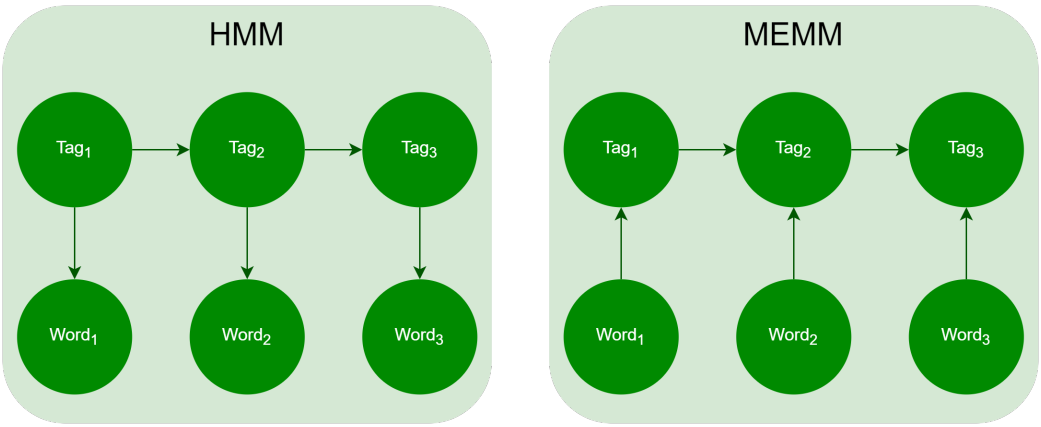



Figure 7.5: In the HMM, the hidden state depends only on the previous hidden state. In the MEMM, a hidden state $T_i$ depends on both the previous state and on the observation $w_i$: $P(T|W) = \prod P(t_i | t_{i-1}, w)$

## 7.5 Named Entity Recognition [TODO]

Chapter 8

# Tensorflow

Tensorflow is an open-source platform for machine learning published by Google in 2015. In this chapter, we will summarize some of the Tensorflow features. However, we suggest warmly reading the online documentation.

## 8.1 Execution Models

TensorFlow's **eager execution** is an imperative programming environment that evaluates operations immediately, without building a graph. The operations return concrete values instead of building a graph to run later. That means that we can inspect results using print() or a debugger, making the debugging easier. It is also easier to program since it has the same control flow in Python instead of graph control flow.

On the other hand, we have the **Graph execution** that means that tensor computations are executed as a TensorFlow graph. In this case, the operations construct a computational graph to be run later. Inside the graph, the operations represent computation units while the tensors represent the unit of data that flow between operations. Since these graphs are data structures, they can be saved, run, and restored all without the original Python code. Moreover, graphs allow Tensorflow to run fast, run in parallel, and run efficiently on multiple devices. Hence, they are more suitable for production deployment.

**Tensors** are multi-dimensional arrays with a uniform type (called a dtype). All tensors are immutable: we can never update the contents of

a tensor, only create a new one. For example, the following code builds a scalar tensor:

```
rank_0_tensor = tf.constant(4)
```

But we can increase the dimensions by building a 1-d tensor (a vector), a 2-d tensor (a matrix), and so on.

```
rank_1_tensor = tf.constant([2.0, 3.0, 4.0])
rank_2_tensor = tf.constant([[1, 2],
                             [3, 4],
                             [5, 6]], dtype=tf.float16)
```

As you can see from the code, the number of axes determines the tensor's rank: a vector has rank 1, a matrix has rank 2.

## 8.2 Automatic Differentiation

TensorFlow provides the tf.GradientTape API for automatic differentiation; that is, computing the gradient of a computation with respect to some inputs, usually tf.Variables. Here is how to define the derivative of square:

```
def square(x):
    return tf.multiply(x,x)

def grad(x):
   with tf.GradientTape () as
        return t.gradient(square(x), x)

square(3.).numpy()# x^2 = 9.
grad(3.).numpy()# 2 * x = 6.
```

Tensorflow uses the reverse automatic auto differentiation that applies several times the chain rule to the initial formula until it reaches some factor for which it knows both the output and the derivative. For instance, let us consider the following formula:

$$e = (a + b) * (b + 1)$$

The algorithm will first try to build a graph by applying the chain rule until it reaches some factors for which it has both the output and the derivative:

$$\frac{\partial e}{\partial a} = \frac{\partial e}{\partial c}\frac{\partial c}{\partial a}$$

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial d}\frac{\partial d}{\partial a}$$

Where $c = a + b$ and $d = b + 1$. The corresponding graph is the following:
From the bottom of the graph, the algorithm calculates the output of all the

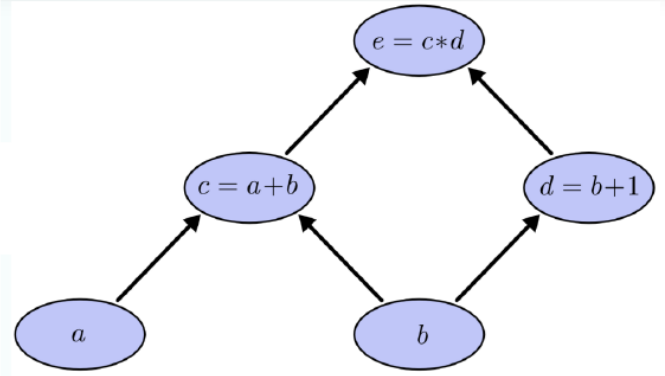



Figure 8.1: Graph used to compute the derivatives of $e = (a + b) * (b + 1)$

nodes. As shown in figure 8.2, a and b are given as input. Then we compute c by summing a and b and d computing b + 1. We terminate multiplying c and d obtaining e. Finally, the algorithm goes backward from e applying the chain rule, computing the derivatives, and multiplying them. For instance, we obtain the partial derivative of e with respect to a, multiplying the partial derivatives in the path between a and e.

To do this, every element in the graph must implement forward and backward methods that compute, in turn, the output and the derivative. The ComputationalGraph class, which represents the entire graph, uses its forward method and backward method to do what we explained in the previous example. The forward method uses the forward methods implemented by the graph elements to compute the final loss. While the backward method employs the backward methods implemented by the graph elements to compute the gradient. Figure 8.3 shows the ComputationalGraph code while an example of graph elements is presented in figure 8.4.

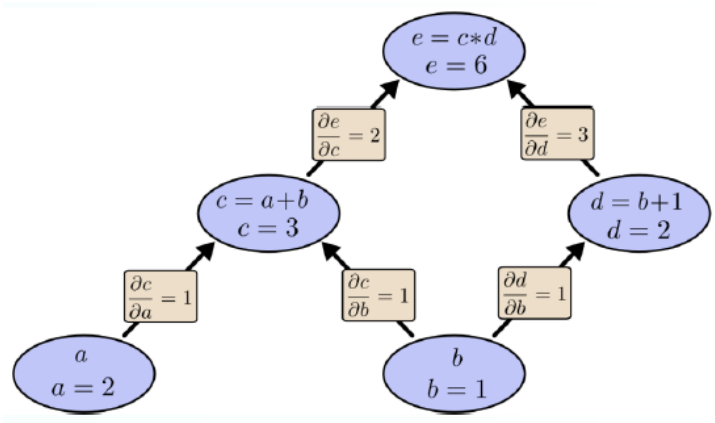



Figure 8.2: Forward and backward steps to compute the derivative of $e = (a + b) * (b + 1)$

## 8.3 Autograph

**tf.function** allows us to transform a subset of Python syntax into portable, high-performance Tensorflow graph. To transform a function into a TensorFlow graph-compatible code, we just need to annotate a function with the tf.function decorator:

```
@tf.function def f(x):
    while tf.reduce_sum(x) > 1
        x = tf.tanh(x)
    return x
```

We can still call that function as a normal one, but it will be compiled into a graph, which means we will get the benefits of faster execution, running on GPU or TPU or exporting to SaveModel.

```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

Figure 8.3: ComputationalGraph code

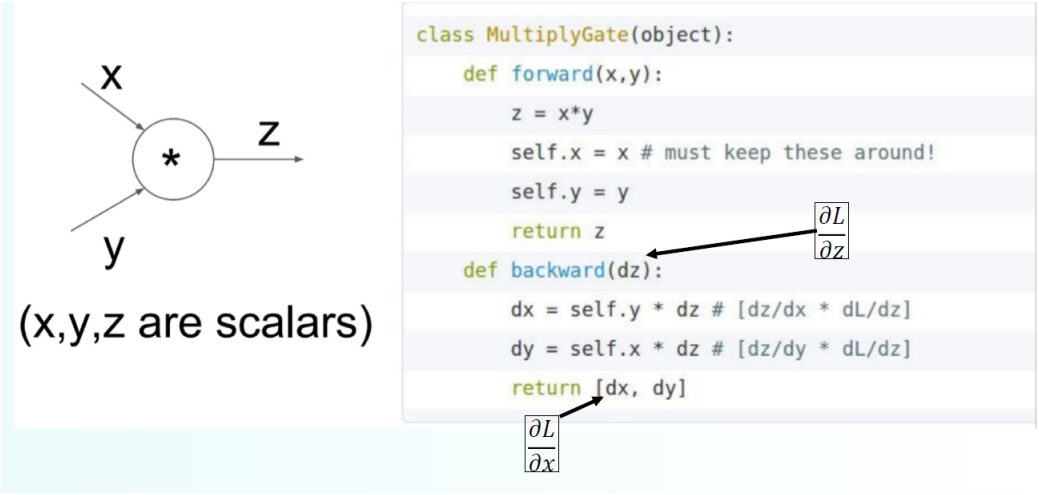



Figure 8.4: For each element of the ComputationalGraph, we must define both the forward and backward methods. As you can see, the forward method returns the output of the multiplication between x and y while the backward method returns the gradient.

Chapter 9

# Parsing

Parsing, **syntax** analysis, or **syntactic analysis** is the process of analyzing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal **grammar**.

Words are composed into bigger units, called **constituents**, to convey complex ideas and to build statements to express.
In the definition, copied by Wikipedia, we used two keywords:

**Syntax** refers to the way words are arranged together into constituents or larger units.

**Grammar** is a formalism to describe the synstax of a language.

We can use parsing to recognize structural ambiguities such as the propositional attachment or the coordination scope. For instance, let us consider the following sentence: *San Jose cops kill man with knife.* In this phrase, *with* introduces an ambiguity, we can indeed either think that San Jose cops stab a man or that San Jose cops kill a man that was using a knife. That is a propositional attachment ambiguity. Consider now this sentence: *Shuttle veteran and longtime NASA executive Fred Gregory appointed to board.* In this case, we have a coordination scope ambiguity since the sentence could either refer to two people (the veteran and Fred Gregory) or to only Fred Gregory that is also a veteran.

We can use parsing in many use cases as relation extraction, semantic relation, translation, sentiment analysis and so on.

## 9.1 Constituency Grammar approach

This approach uses a defined Context-Free Grammar to build a tree that should represent the sentence.

A Context-Free Grammar could be as follow:

$$\begin{aligned} NP &\rightarrow DetNominal \\ NP &\rightarrow ProperNoun \\ Nominal &\rightarrow Noun|NominalNoun \\ Det &\rightarrow a \\ Det &\rightarrow the \\ Noun &\rightarrow flight \end{aligned} \tag{9.1}$$

Formally, a Context-Free Grammar is a formal grammar composed of:

- N, a set of non-terminal symbols (NP and Nominal).
- $\Sigma$, a set of **terminal** symbols (Det and Noun).
- R, a set of **rules** or **productions**, each of the form $A \rightarrow \alpha$ where $A$ is a non-terminal, $\alpha$ is a string of symbols from infinite set of strings $(\Sigma \cup N)*$.
- S, a designated **start symbol** $\in N$.

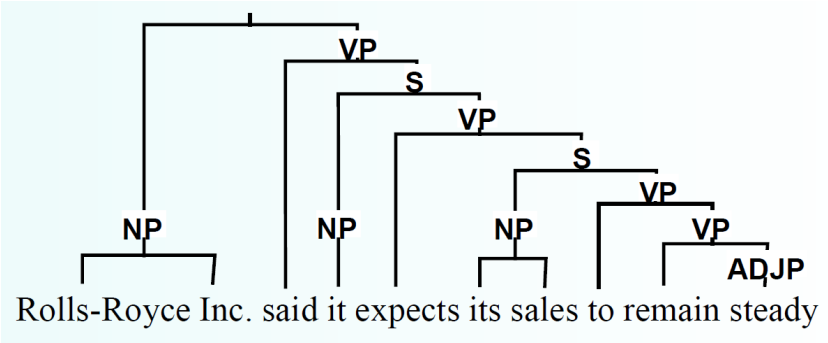



Figure 9.1: Phrase structure parse tree example

Once we have defined the grammar, the output will be a phrase structure parse tree like the one shown in figure 9.1.

A simple algorithm that given a sentence returns the most likely phrase structure parse tree consists of a pipeline composed of four components (see figure 9.2):

1. **GEN** given a sentence returns a set of candidates, which are the possible phrase structure parse trees.

2. $\Phi$ maps a candidate x to a feature vector $< h_1(x), h_2(x), \ldots, h_d(x) > \in R^d$. A feature is a function on a structure, for instance, h(x) = "Number of times *property* is seen in x".

3. $W$ is a parameter vector $\in R^d$. $\Phi(x) \times W$ maps a candidate to a real-valued score.

4. We finally return the candidate that has the maximum score.

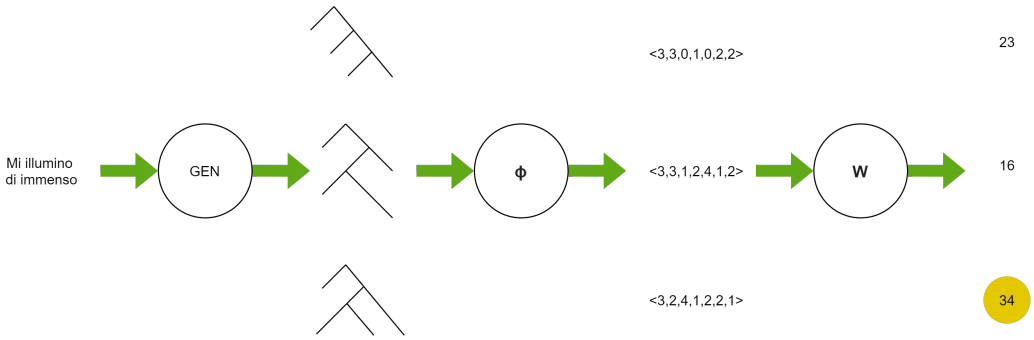



Figure 9.2: Given the sentence "Mi illumino di immenso", GEN produces a set of suitable trees that, in turn, $\Phi$ transforms into arrays of features. Finally, we return the candidate with the maximum score, which we obtain by multiplying its features vector by W. In the example, we return the third candidate since it has the maximum score.

Formally speaking, the algorithm implements a function $F : X \to Y$ with X a set of sentences and Y a set of possible outputs (e.g. trees). The four components define:

$$F(x) = \underset{y \in GEN(x)}{\arg\max}\, \Phi(y) \cdot W \tag{9.2}$$

## 9.2 Dependency Grammar

The dependency structure is a tree that shows binary dependency relations between words. If a word A depends on another word B, we draw a labeled direct link from B to A. We call B the head and A the dependant. The label of the direct link indicates the type of dependency. For instance, in figure 9.3, "prefer" is the head while "I" is the dependant. The root of the tree is always a verb. In figure 9.3, the root is indeed "prefer". As you

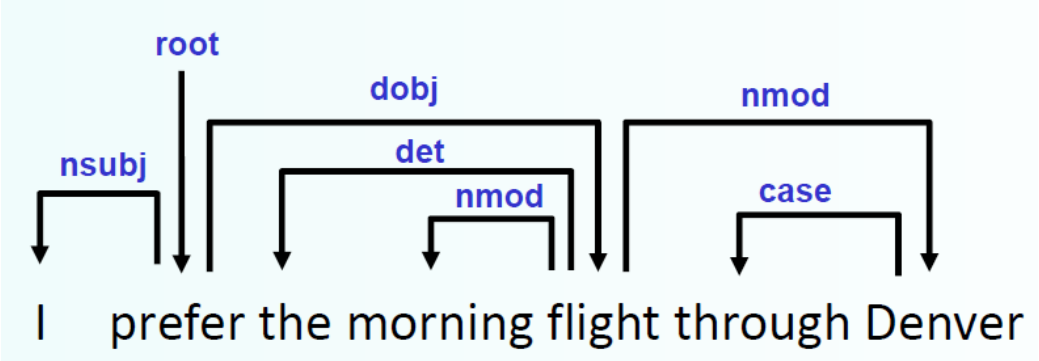



Figure 9.3: Dependency tree of the sentence "I prefer the morning flight through Denver"

can notice comparing Figures 9.3 and 9.1, the constituency tree contains more nodes than the sentence words and is deeper than a dependency tree in which the elements are the words and the arches between the words. So dependency trees are flatter than constituency trees. Therefore, they are easier to understand and to annotate than constituency trees.

Formally a dependency graph (tree) is defined as follow: let $R = \{r_1, \ldots, r_m\}$ be the set of permissible dependency types. A **dependency graph** for a sequence of words $W = w_1 \ \ldots \ w_n$ is a labeled directed graph $D = (W, A)$, where:

- $W$ is the set of nodes, i.e. word tokens in t he input sequence.
- $A$ is a set of labeled arcs $(w_i, w_j, r)$.
- $\forall w_j \in W$ there is **at most one arc** $(w_i, w_j, r) \in A$. This condition ensures that the graph will be a tree.

## 9.3 Dependency parsing - Transitional based

The transitional based dependency parsing defines a transition system that leads to a parse tree while analyzing a sentence one word at a time. A transitional-based parser processes the input by performing transitions between parser states. The parser state is a triple $< S, B, A >$ where:

- S is a stack of partially processed tokens.
- B is a buffer of (remaining) input tokens.
- A is the arc relation for the dependency graph.

At each step, the basic algorithm will check if there exist dependencies between the last element of S, which we call s, and the first element of B, which we call n. If s depends on n, the algorithm performs a left-arc action that adds an arc $< n, s, \textit{dependency name} >$ to A and deletes s from S:

$$\textbf{LEFT ARC} \quad \frac{< S|s, n|B, A >}{< S, n|B, A \cup \{(n, s, r)\}} \tag{9.3}$$

On the other hand, if n depends on s, the algorithm performs a right-arc action that adds an arc $< s, n, \textit{dependency name} >$ to A, attaches s to B, and deletes n from B:

$$\textbf{RIGHT ARC} \quad \frac{< S|s, n|B, A >}{< S, s|B, A \cup \{(s, n, r)\}} \tag{9.4}$$

Finally, if there is not any dependency, the algorithm performs a shift that pushes n into S:

$$\textbf{SHIFT} \quad \frac{< S, n|B, A >}{< S|n, B, A >} \tag{9.5}$$

The algorithm terminates when the buffer becomes empty. Let us consider an example parsing the sentence "I prefer the morning flight" (see Figures 9.3 and 9.4):

1. After a SHIFT action, the initial parser state will be <"I", "prefer the morning flight", {}>. The algorithm will start from the pair "I" and "prefer" and check if the words have a dependency. It concludes that "I" depends on "prefer", so it adds to the beginning empty dependency graph, an arc from "prefer" to "I". (LEFT ARC)

2. The parser state is: now <"", "prefer the morning flight", A>. So, the algorithm performs a shift, pushing "prefer into the stack. (SHIFT)

3. The next pair to be considered is "prefer" and "the". In this case, the algorithm does not find any dependency. Therefore it performs a shift action passing to the words "the" and "morning". (SHIFT)

4. Again, no dependencies, again a shift operation that leads the algorithm to the words "morning" and "flight". (SHIFT)

5. The algorithm finds an NMOD relation between "morning" and "flight". Therefore, it performs a Left-arc operation, which links the two words. (LEFT ARC)

6. Then, it also detects a DET dependency between "the" and "flight". (LEFT ARC)

7. The algorithm finds a DOBJ dependency from "prefer" to "flight", therefore it performs a right ARC action. (RIGHT ARC)

8. Finally, the last SHIFT makes the buffer empty.

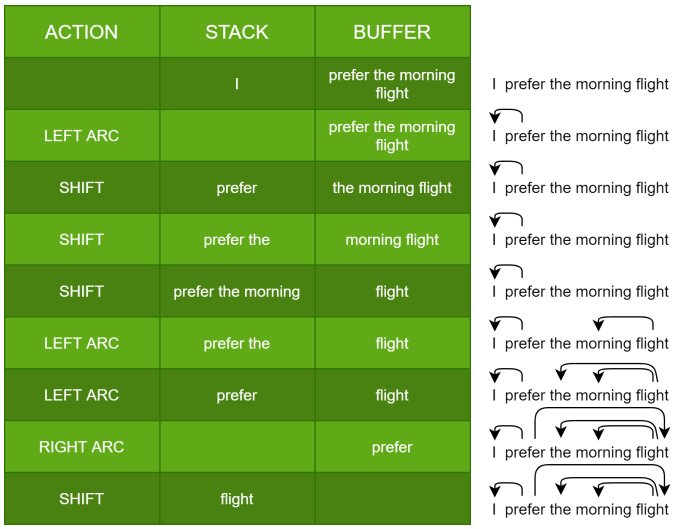



Figure 9.4: transition algorithm example. Note that the algorithm brings I into the stack though a SHIFT action.

The algorithm's output is not the dependency tree itself. But the sequence of actions that lead to that dependency tree. Thus, in our example, the output is [LEFT ARC, SHIFT, SHIFT, SHIFT, LEFT ARC, LEFT ARC, RIGHT ARC, SHIFT].

The algorithm uses a trained model to predict the next action given a representation of the context's current state defined by the triple <S, B, A>. During training, the **gold tree** of each action can be used to suggest which actions to perform in order to rebuild such gold tree. There can be more **than one possible sequence** to produce the same parse tree. An **Oracle** is an algorithm that given the **gold tree** for a sentence, produces a proper **sequence of actions** that a parser may use to obtain that gold tree from the input sequence. In other words, the training examples for our model will be <sentence, O(gold tree)> where O(gold tree) is the expected output, which is the oracle's output given the sentence gold tree. We can summarize the oracle's behavior with the following code:

```
function arc_standard_oracle(sentence)
#Input Sentence: (w1, w2, ..., wn)
S = []
B = [w1, w2, ..., wn]
while B != []:
    if has_dependency(S[0],B[0]) and all_children_attached(S[0]):
        left_arc()
    else if has_dependency([S[0],B[0]) and all_children_attached(B[0]):
        right_arc()
    else:
        shift()
```

The additional condition `has_dependency(S[0],B[0])` assures that each word will be connected to all its children. Indeed, executing the left arc action, we can not anymore add children to S[0], and therefore if there are still children not attached to S[0], those children will never be attached to S[0]. The same holds for the right arc action concerning B[0].

Figure 9.3 shows what we call a **projective** dependency tree. A dependency tree is projective if and only if every arc is projective. An arc $w_i \rightarrow w_k$ is projective if and only if:

$$\forall j, i < j < k \text{ or } i > j > k : w_i \rightarrow^* w_j \tag{9.6}$$

Intuitively, an arc is projective if we can draw it on a plane without intersection. All the arcs in Figure 9.3 do not cross with each other. On the other hand, Figure 9.5 shows a non-projective tree where for instance, the arc from "ale" to "nejen" crosses the arc from "ale" to "lze".

Unfortunately, the algorithm presented, called the arc standard algorithm,

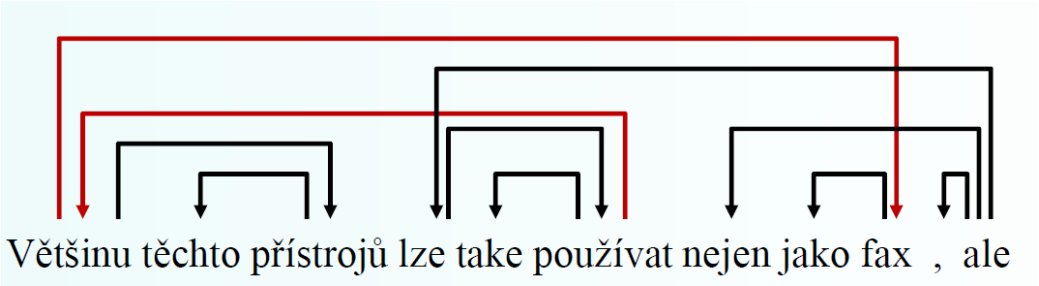



Figure 9.5: Non projective tree

does not deal with non-projectivity. However, it represents a fast deterministic linear algorithm to produce a projective dependency tree. Indeed, it parses $n$ words in $2n$ transitions.

### Arc Eager Transitions

A simple extension of the arc standard algorithm is the arc eager algorithm that helps us deal better with right arc actions. In particular, the right arc action instead of deleting n pushes n onto the stack:

$$\textbf{RIGHT ARC} \quad \frac{< S|s, n|B, A >}{< S|s|n, B, A \cup \{(s, n, r)\}} \tag{9.7}$$

Then the algorithm introduces a new command to delete the top word in the stack called reduce:

$$\textbf{REDUCE} \quad \frac{< S|s, B, A >}{< S, B, A >} \tag{9.8}$$

Figure 9.6 depicts the algorithm working on the sentence "They told him a story".

| Action | Stack | Buffer |
|---|---|---|
| | [] | They told him a story |
| Shift | They | told him a story |
| LA-subj | | told him a story |
| Shift | told | him a story |
| RA-obj | told him | a story |
| Reduce | told | a story |
| Shift | told a | story |
| LA-det | told | story |
| RA-obj | told story | |
| Reduce | told | |

Figure 9.6: Arc Eager Transitions algorithm example

### 9.3.1 Non Projective Transitions

Dealing with non-projectivity means add new actions that allow the algorithm to create arcs that cross other arcs.

#### Attardi transitions

Attardi presented a set of simple rules that generalize the right/left arc actions. For instance, he added the right-arc2 action, that is, the right-arc action where instead of connecting S[0] and B[0], we attach S[1] and B[0]:

$$\textbf{RIGHT-ARC2} \quad \frac{< S|s_2|s_1, n|B, A >}{< S|s_1, n|B, A \cup \{(s_2, n, r)\}} \tag{9.9}$$

Where S[0] = $s_1$, S[1] = $s_2$ and B[0] = $n$.
We can do the same for the left arc action:

$$\textbf{LEFT-ARC2} \quad \frac{< S|s_2|s_1, n|B, A >}{< S|s_2, s_1|B, A \cup \{(n, s_2, r)\}} \tag{9.10}$$

Note that we can now use a left-arc2 action to connect "ale" and "nejen" in Figure 9.5. However, we can not still attach "fax" to "Většinu". Thus, we need two additional rules that consider more distant words:

$$\textbf{RIGHT-ARC3} \quad \frac{< S|s_3|s_2|s_1, n|B, A >}{< S|s_2|s_1, n|B, A \cup \{(s_3, n, r)\}} \tag{9.11}$$

$$\textbf{LEFT-ARC3} \quad \frac{< S|s_3|s_2|s_1, n|B, A >}{< S|s_3|s_2, s_1|B, A \cup \{(n, s_3, r)\}} \tag{9.12}$$

### Swap transition

An alternative consists in adding, to the basic arc actions, the swap action that exchanges, at most one time, S[0] and B[0]:

$$\textbf{SWAP} \quad \frac{< S|s, n|B, A >}{< S|n, s|B, A >} \quad s < n \tag{9.13}$$

In this case, the oracle swaps the tokens whenever $T(s) < T(n)$. T(i) = k means that the ith sentence token is the kth met in the inorder traversal of the dependency gold graph. For instance, the inorder traversal of the tree pictured in Figure 9.7 is 1 2 5 6 7 3 4 8. In the following, we recall the pseudocode of the inorder traversal in a tree:

```
Algorithm Inorder(tree)
   Inorder(left-subtree) #Traverse the left subtree
   Visit the root.
   Inorder(right-subtree) #Traverse the right subtree, i.e., call
```

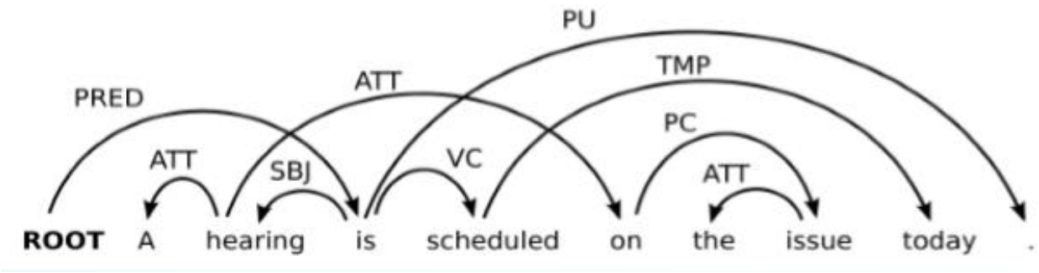



Figure 9.7: Dependency tree of the sentence "a hearing is scheduled on the issue today"

Figure 9.8 shows the parser working on the sentence "a hearing is scheduled on the issue today".

| Action | Stack | Buffer |
|---|---|---|
| | [] | A hearing is scheduled on the issue today |
| Shift | A | hearing is scheduled on the issue today |
| Shift | A hearing | is scheduled on the issue today |
| LA-ATT | hearing | Is scheduled on the issue today |
| Shift | hearing is | scheduled on the issue today |
| Shift | hearing is scheduled | on the issue today |
| Shift | hearing is scheduled on | the issue today |
| Swap | hearing is on | scheduled the issue today |
| Swap | hearing on | Is scheduled the issue today |
| Shift | hearing on is | scheduled the issue today |
| Shift | hearing on is scheduled | the issue today |
| Shift | hearing on is scheduled the | Issue today |
| Swap | hearing on is the | scheduled issue today |
| Swap | hearing on the | Is scheduled issue today |

Figure 9.8: Swap algorithm example. Note, probably image with errors

## 9.4 Learning Phase

We can see the parser problem as a classification one: at each step, the classifier, given a context representation of the state, should return the same action indicated by the oracle.

More in detail, the classifier takes into input a context representation of the state in terms of features. The features may be hand-made or automatically generated. The classifier used are:

**Maximum Entropy** that is fast but not very accurate.

**SVM** that is very accurate but slow.

**Multilayer perceptron** that is very accurate and fast.

**Depp learning** that uses word embedding as features.

The first-ever statistical transitional parser based on dependency trees was DeSR, who won the first CoNLL-X competition about parsing in 2006. The shared task of the CoNNL-X was to assign labeled dependency structures for a range of languages employing a fully automatic dependency parser. The inputs were tokenized and tagged sentences. For each token, the parser must output its head and the corresponding dependency relation. The evaluation metrics used were:

**UAS** (Unlabeled Attachment Score) proportion of tokens that are assigned the **correct head**.

**LAS** (Labeled Attachment Score) proportion of tokens that are assigned **both** the **correct head** and the correct dependency relation **label**.

**CLAS** (Context Word Labeled Attachment Score) like LAS but disregards attachments of punctuation and function words, i.e. determiners (det), classifier (clf), adpositions (case), auxiliaries (aux,cop), and conjunctions (cc, mark).

### 9.4.1 Issues

#### Single-head constraint

In CoNNL-X competition the dependency graph $D = (W, A)$ produced by the parser must be a **directed rooted tree**:

- D is (weakly) **connected**:

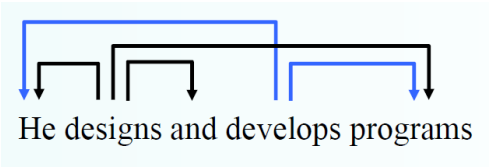
$$\text{if } i, j \in V, i \leftrightarrow^* j$$

- D is **acyclic**:

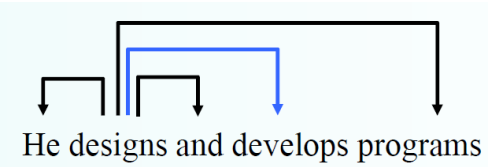
$$\text{if } i \rightarrow j, \text{ then not } j \rightarrow^* i$$

- D obeys the **single-head** constraint:

$$\text{if } i \rightarrow j, \text{ then not } i' \rightarrow j, \text{ for any } i' \neq i$$




(a) He can have only one head




(b) "Designs" is used as unique root

Figure 9.8: Single-head constraint causes problems with conjunctions

The single-head constraint causes problems in handling certain linguistic phenomena. For instance, in Figure 9.8a we have conjunction where "He" depends both on "designs" and "develops". However, due to the single-head constraint, "He" can have only one head. One way to solve this problem is to choose either "designs" or "develops" as root and make everything come from that conjunct.

#### Dynamic Oracle

The oracle suggests the correct path of actions, but if a parser makes mistakes, it will find itself in a state never seen in training without knowing how to recover. That causes error propagation because the parser will carry on its path, probably returning different values to the ones suggested by the oracle.

Things can become worse since a sentence can have multiple parsing sequences, while a static oracle will consider only one. Therefore, the parser could choose a different parsing sequence that may still be correct.

To overpass the problem, we can use a dynamic oracle that allows more than one transition sequence [**goldberg2012dynamic**].

### 9.4.2 Dependency Parser using Neural Networks

In 2014, Chen and Manning presented a dependency parser based on neural networks. The neural network was a classifier that, given a features vector representing the current parse state, returns a probability distribution that indicates how likely action is. The parser generated the features vector concatenating the word embeddings of both the sentence tokens and their POS (Part of Speech). The output layer was a Softmax.

The algorithm reaches good results, as shown in Figure 9.9.

| Parser | Penn TB | Chinese TB | Sent/sec |
|---|---|---|---|
| Standard | 89.9 | 82.7 | |
| Malt | 90.1 | 82.4 | 470 |
| MST | **92.0** | 83.0 | 10 |
| NN | **92.0** | **83.9** | **650** |

Figure 9.9: The parser reaches state-of-the-art accuracy increasing the number of sentence/sec parsed

## 9.5 Graph-based Dependency Parsing

The idea of graph-based parsing is the following: consider all possible dependency trees, score them and eventually return the tree with the highest score. We can score a dependency tree T by the score of its arcs:

$$s(T) = \sum_{i,j,k \in T} s(i, j, k) \tag{9.14}$$

Where $s(i, j, k)$ is a scoring function that takes in input the arc $< i, j >$ with label $k$.

The learning phase will consist of defining the scoring function $s(i, j, k)$.

The inference phase will consist of returning the tree with the highest score. So, it basically works as follow:

1. Choose the arc with highest score from each node (Figure 9.10 step 1).
2. If the resulting graph is not a tree, we identify the cycle and contract it. Then we recalculate arc weights into and out-of-cycle (Figure 9.10 step 2).

In Figure 9.10, we have a cycle between "John" and "saw". Therefore, we consider "John" and "saw" as a single node and recompute all the weights. For instance, the weight from the root to "saw" is 40 because we sum the score of <root, saw> and <saw, john>. Then, we again select the arc with maximum weight, and the process terminates.

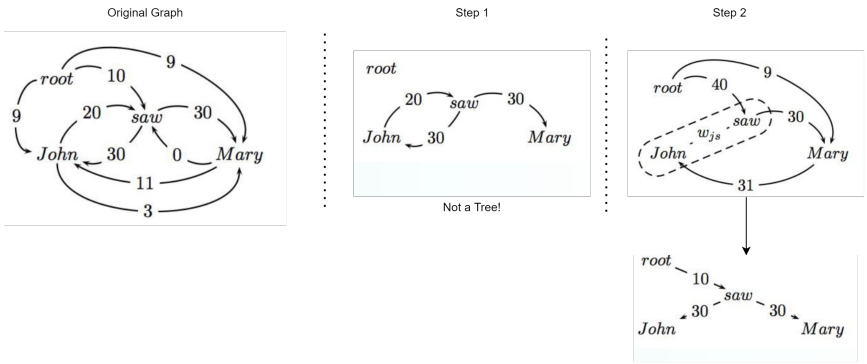



Figure 9.10: Finding Maximum Spanning Tree

## 9.6 Universal Dependencies

The CoNNL-X 2006 competition prove for the first time that it was possible to use one parser for multiple languages. However, the parser had to be trained with language-specific tree-banks to work properly with that specific language. So, people started to think about creating a parser, trained with multiple language tree-banks, and so able to deal with multiple languages. To do that, we needed to find the part of speeches, morphology, and dependencies shared across the different languages. That brought to the born of the Universal Dependencies project, whose objective was to find these similarities.

So, the goal of the project was to find a grammatical annotation that was consistent across languages. In that way, we can build tools that can work in different languages relying upon a single common annotation style for the tree-bank. Another crucial aspect was that it must be a community effort - anyone could (and can) contribute.

Researchers decided to use a **dependency grammar** instead of a constituency one. Before CoNNL-X of 2006, dependency grammars were not very popular except for some languages such as Czech that would not deal with constituency grammars because of non-projectivity. Thanks to CoNNL-X, dependency grammars became popular. So, many of the tree-banks were converted to dependency trees.

The universal annotation assumes that tokens are the units. Only clitics and contradictions are split into multiple tokens. Therefore, the dependency tree will have as many nodes as the tokens. Moreover, each token will be associated to a **Morphological annotation** composed of three elements:

- **Lemma** represents the semantic content of a word.
- **Part-of-Speech** is the tag representing its grammatical class.
- **Features** represents lexical and grammatical properties of the lemma of the particular word form.

Figure 9.11 shows an example of a dependency tree built through universal annotation. Tokens are connected using universal dependencies. Each token has associated its Part-Of-Speech and its morphology features. For instance, "Restaurant" is a NOUN, the case is dative, the number is singular, and the gender is mutual

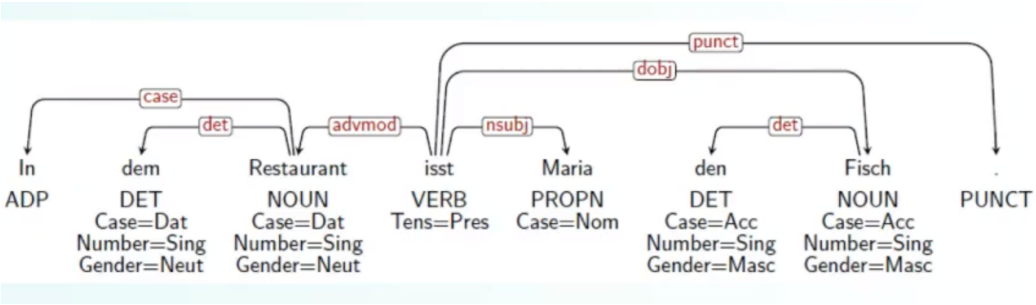



Figure 9.11: Universal dependency tree

Besides, the universal annotation adopts the following rules regarding the connections between the tokens:

- **Content words**[1] are related by dependency relations.
- **Function words**[2] attach to the content word they modify.
- **Punctuation** attach to head of phrase or clause.

For example, in Figure 9.12 we connect "chased" and "street", which are content words, and we attach "the" and "down", which are function words, to "street", which is the content word they modify. Due to the third rule, we attach the period to the head of phrase that is "chased".

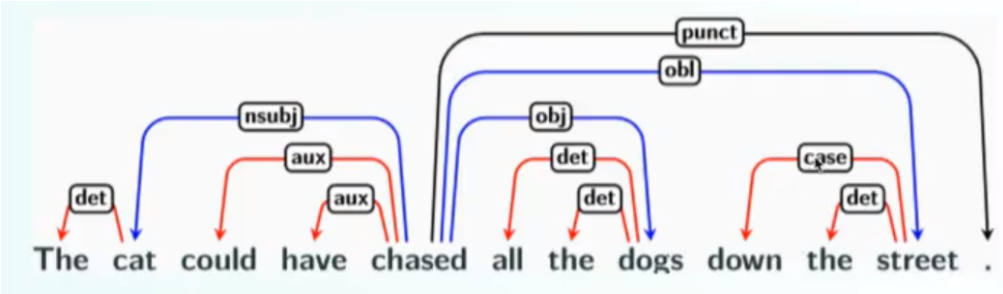



Figure 9.12: Another example of universal dependency tree

[1]Content words, in linguistics, are words that possess semantic content and contribute to the meaning of the sentence in which they occur.

[2]function words are words that have little lexical meaning or have ambiguous meaning and express grammatical relationships among other words within a sentence, or specify the attitude or mood of the speaker. Examples are "the", "He/She", "down", "up", "on".

Keeping **content words as head** promotes parallelism across languages. In Figure 9.13, we can notice similarities between the two parse trees, in particular concerning the dependencies between content words (blue arrows). For instance, between subject and verb, we have the same relation (passive subject), and it is irrelevant that English has an auxiliary and a determiner. That means we can more easily extract information because we can find the correct connection between content words.

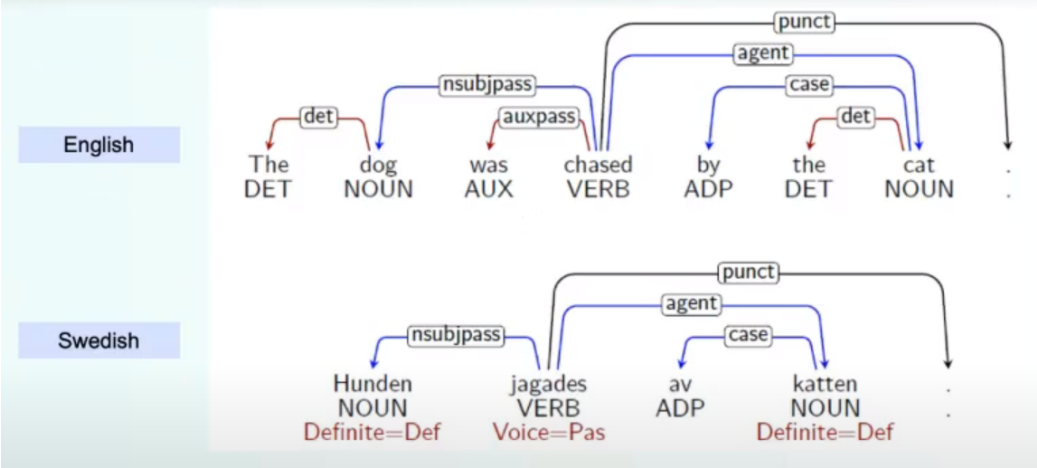



Figure 9.13: Dependency trees about the same sentence written in different languages. The dependencies between the content words (blue arrows) are the same.

Marneffe et al. publish the first set of universal dependency relations in 2014 [8]. Figure 9.14 shows those universal dependencies.

Figure 9.16 shows the **17 universal part-of-speech** tags, based on the Google Universal Tagset [11]. Finally, Figure 9.17 lists the standardized inventory of morphological features, based on the interset system.

Udify represents an example of a single model for multiple languages [5]. Indeed, It is a single model that has been trained in 75 languages. To work, it must rely on some input representation that is consistent across languages. It extracts that representation using BERT. BERT produces several layers: each word will have a hidden representation at multiple layers, typically 8-12. Then Udify takes a combination of those and specializes several neural networks to produce various annotations: there will be a neural network for dependency tags, another for lemmas, and so on. As shown in Figure

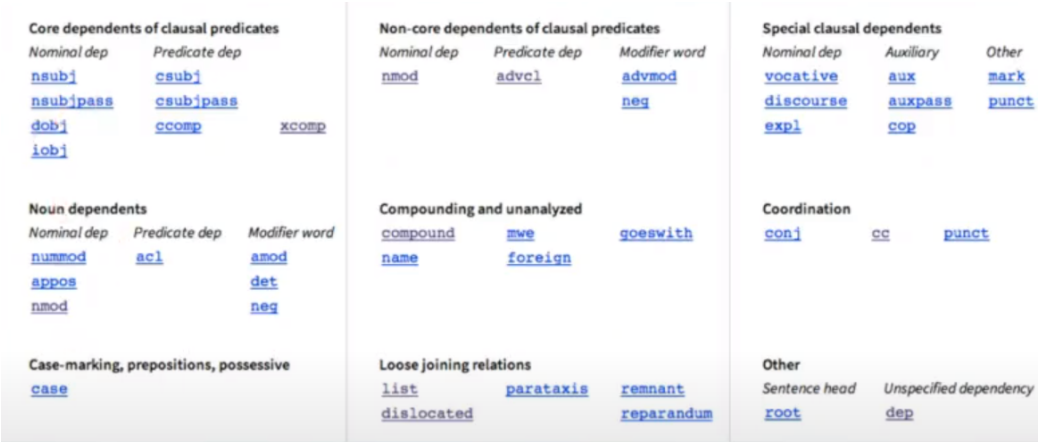



Figure 9.14: The dependencies are subdivided into categories. For instance, in the top left-most category we have the subject (nsubj), the passive subject (nsubjpass), the direct object (dobj), or the indirect object (iobj). An interesting dependency is an apposition (appos) that gives us an ontological relationship. For example, in "Mattarella president of republic went to Bergamo yesterday", between "Mattarella" and "president of the republic" we have an apposition. Another pretty dependency are multi-words such as "New York".

9.15, the algorithm can work well on languages for which it was not trained (zero-shot learning).

| Treebank | | UPOS | FEATS | LEM | UAS | LAS |
|---|---|---|---|---|---|---|
| Breton KEB | br keb | 63.67 | 46.75 | 53.15 | 63.97 | 63.97 |
| Tagalog TRG | tl trg | 61.64 | 35.27 | 75.00 | 64.73 | 39.38 |
| Faroese OFT | fo oft | 77.86 | 35.71 | 53.82 | 69.28 | 61.03 |
| Naija NSC | pcm nsc | 56.59 | 52.75 | 97.52 | 47.13 | 33.43 |
| Sanskrit UFAL | sa ufal | 40.21 | 18.45 | 37.60 | 41.73 | 19.80 |

Figure 9.15: Zero-shot learning. The algorithm can work pretty well with languages such as Breton or Faroese which are not included in the training set

| Open class words | Closed class words | Other |
|---|---|---|
| ADJ | ADP | PUNCT |
| ADV | AUX | SYM |
| INTJ | CONJ | X |
| NOUN | DET | |
| PROPN | NUM | |
| VERB | PART | |
| | PRON | |
| | SCONJ | |

Figure 9.16: Set of universal POS

| Lexical features | Inflectional features | |
|---|---|---|
| | *Nominal** | *Verbal** |
| PronType | Gender | VerbForm |
| NumType | Animacy | Mood |
| Poss | Number | Tense |
| Reflex | Case | Aspect |
| Foreign | Definite | Voice |
| Abbr | Degree | Evident |
| | | Polarity |
| | | Person |
| | | Polite |

Figure 9.17: Set of universal POS

Chapter 10

# Convolutional Neural Networks for NLP

The name convolution indicates that the network employs a mathematical operation called **convolution**:

$$\textbf{CONVOLUTION} \quad s(t) = (x * w)(t) = \sum_{m=-M}^{M} x(m)w(t-m) \qquad (10.1)$$

x represents the convolution input while w is the function, called **kernel** or **filter**, that we apply to x. The kernel output is called **feature space**.

Convolution is classically used to extract features from images, which we can represent as matrices, as in Figure 10.1, where we perform a convolution using a $3 \times 3$ kernel.

Practically, a CNN works taking only a portion of the input (sliding window) and using every time the same weights (weight sharing). For instance, performing a convolution C over an array A through a kernel K of size 3 can mean executing the following calculations:

1. $C[0] = K[0] * A[0] + K[1] * A[1] + K[2] * A[2]$
2. $C[1] = K[0] * A[1] + K[1] * A[2] + K[2] * A[3]$
3. $C[2] = K[0] * A[2] + K[1] * A[3] + K[2] * A[4]$
4. That, in general, is: $C[i] = K[0] * A[i-1] + K[1] * A[i] + K[2] * A[i+1]$

In NN terms, we **traverse the data** with the same neuron whose weights are the kernel values.

Due to weight sharing, the model is position-invariant. That is, eyes must not necessarily be in humans' heads, but they can appear in humans' hands.

The convolution can be multi-dimensional. For instance, in Figure 10.1, we perform a 2D convolution since we move both left-to-right and top-to-bottom, while the convolution in the previous array example is one-dimensional.

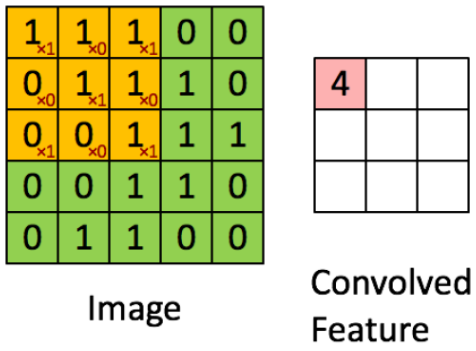



Figure 10.1: Convolution over an image. Red numbers show filter weights. We get the feature space of the yellow portion by multiplying element-wise it by kernel weights.

## 10.1 Convolution for text

The question is now how to apply convolution to text. First, we can turn sequences of words into a matrix by exploiting the fact that we can represent each word by its embedding. Then we can perform a one-dimensional convolution using one or more kernels with as many columns as the size of the embeddings and with as many rows as we want. We will say that the kernel size is the number of its rows, that is, the number of words we consider for each convolution.

Let us take the sentence "Not going to the beach tomorrow :-(" over which applying a kernel of size 3. Suppose that the embeddings and the kernel weights are the ones shown in Figure 10.2. We perform the first convolution

over the embeddings of "Not going to", that is:

$$\begin{aligned} C[0] =& 0.2 * 3 + 0.1 * 1 + (-0.3) * 2 + 0.4 * (-3) + 0.5 * (-1) + \ldots \\ & + (-0.2) * (-1) + 0.4 * 1 = -1 \end{aligned}$$

Then we sum C[0] by a bias, and then we feed a non-linear function that will return the final output. Once we have done, we slide the window and compute the convolution for the words "going to the":

$$\begin{aligned} C[1] =& 0.5 * 3 + 0.2 * 1 + (-0.3) * 2 + (-0.1) * (-3) + (-0.1) * (-1) + \ldots \\ & + 0.1 * (-1) + 0.1 * 1 = -0.5 \end{aligned}$$

And again, we sum the bias and apply the non-linear function. The process goes on until we reach the last 3-gram to consider: "beach tomorrow :-(". The final output is shown in Figure 10.3.

INPUT

| | | | | |
|---|---|---|---|---|
| **Not** | 0.2 | 0.1 | −0.3 | 0.4 |
| **going** | 0.5 | 0.2 | −0.3 | −0.1 |
| **to** | −0.1 | −0.3 | −0.2 | 0.4 |
| **the** | 0.3 | −0.3 | 0.1 | 0.1 |
| **beach** | 0.2 | −0.3 | 0.4 | 0.2 |
| **tomorrow** | 0.1 | 0.2 | −0.1 | −0.1 |
| **:-(** | −0.4 | −0.4 | 0.2 | 0.3 |

KERNEL

| | | | |
|---|---|---|---|
| 3 | 1 | 2 | −3 |
| −1 | 2 | 1 | −3 |
| 1 | 1 | −1 | 1 |

Figure 10.2: Word embeddings and kernel. Notice that the kernel has as many columns as the size of the embeddings.

| | | SUM | + BIAS | NON-LINEAR |
|---|---|---|---|---|
| C[0] | **w1,w2,w3** | −1.0 | 0.0 | 0.50 |
| C[1] | **w2,w3,w4** | −0.5 | 0.5 | 0.38 |
| C[2] | **w3,w4,w5** | −3.6 | -2.6 | 0.93 |
| C[3] | **w4,w5,w6** | −0.2 | 0.8 | 0.31 |
| C[4] | **w5,w6,w7** | 0.3 | 1.3 | 0.21 |

Figure 10.3: Convolution output using input and kernel shown in Figure 10.2. For instance, the output for C[0] is non-linear-function(-1 + bias) = 0.5

From Figure 2, we can see that the output will be smaller than the original input. Indeed, the number of outputs is 5, while the input has 7 words. If we want to avoid that, a simple approach consists of adding some padding values before the input start and after the input end. In Figure 10.4, we add one row before "Not" and one row after ":-(" so that the output will contain 7 elements.

| | | | | |
|---|---|---|---|---|
| **0** | 0.0 | 0.0 | 0.0 | 0.0 |
| **Not** | 0.2 | 0.1 | −0.3 | 0.4 |
| **going** | 0.5 | 0.2 | −0.3 | −0.1 |
| **to** | −0.1 | −0.3 | −0.2 | 0.4 |
| **the** | 0.3 | −0.3 | 0.1 | 0.1 |
| **beach** | 0.2 | −0.3 | 0.4 | 0.2 |
| **tomorrow** | 0.1 | 0.2 | −0.1 | −0.1 |
| **:-(** | −0.4 | −0.4 | 0.2 | 0.3 |
| **0** | 0.0 | 0.0 | 0.0 | 0.0 |

| | |
|---|---|
| **0,w1,w2** | −0.6 |
| **w1,w2,w3** | −1.0 |
| **w2,w3,w4** | −0.5 |
| **w3,w4,w5** | −0.1 |
| **w4,w5,w6** | −0.2 |
| **w5,w6,w7** | 0.3 |
| **w6,w7,0** | −0.5 |

Figure 10.4: Extra padding values example

Usually, we employ more than one filter, as shown in Figure 10.5. In this case, we will have an input column for each kernel.

| | | | | |
|---|---|---|---|---|
| **0** | 0.0 | 0.0 | 0.0 | 0.0 |
| **Not** | 0.2 | 0.1 | −0.3 | 0.4 |
| **going** | 0.5 | 0.2 | −0.3 | −0.1 |
| **to** | −0.1 | −0.3 | −0.2 | 0.4 |
| **the** | 0.3 | −0.3 | 0.1 | 0.1 |
| **beach** | 0.2 | −0.3 | 0.4 | 0.2 |
| **tomorrow** | 0.1 | 0.2 | −0.1 | −0.1 |
| **:-(** | −0.4 | −0.4 | 0.2 | 0.3 |
| **0** | 0.0 | 0.0 | 0.0 | 0.0 |

| | | | |
|---|---|---|---|
| **0,w1,w2** | −0.6 | 0.2 | 1.4 |
| **w1,w2,w3** | −1.0 | 1.6 | −1.0 |
| **w2,w3,w4** | −0.5 | −0.1 | 0.8 |
| **w3,w4,w5** | −3.6 | 0.3 | 0.3 |
| **w4,w5,w6** | −0.2 | 0.1 | 1.2 |
| **w5,w6,w7** | 0.3 | 0.6 | 0.9 |
| **w6,w7,0** | −0.5 | −0.9 | 0.1 |

Apply 3 **filters** (or **kernel**) of size 3

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 2 | −3 | 1 | 0 | 0 | 1 | 1 | −1 | 2 | −1 |
| −1 | 2 | 1 | −3 | 1 | 0 | −1 | −1 | 1 | 0 | −1 | 3 |
| 1 | 1 | −1 | 1 | 0 | 1 | 0 | 1 | 0 | 2 | 2 | 1 |

Figure 10.5: 3D channel convolution with padding

After the convolution, we can then combine all the convolutions by a pooling operation. For instance, the max-pooling takes the maximum of

each column. Thus, it returns a vector of size the number of kernels. There are other pooling types. For example, the average-pooling, instead of taking the maximum, takes the average.


Figure 10.6: one-dimensional convolution, padded, with max pooling over time

Until now, we have always considered a slide of one position. In these cases, we say that the convolutional neural network has stride 1. More in general, the stride is a CNN's hyperparameter that modifies the amount of movement over the input data. In NLP, the stride is usually one, but of course, there are some cases where having a higher stride is better. Figure 10.7 shows the usual example with stride 2.

## 10.2 Sentiment analysis on Twitter

SemEval Shared Task Competition makes sentiment analysis on Twitter relevant. In the first year of the competition, in 2013, the top system was a very complex system based on an SVM with sentiment lexicons and many lexical features. The years after have seen the predominance of convolutional neural networks.

The main problem of sentiment analysis of Twitter was the lack of available classified data. Indeed, Twitter does not give stars to comments to say

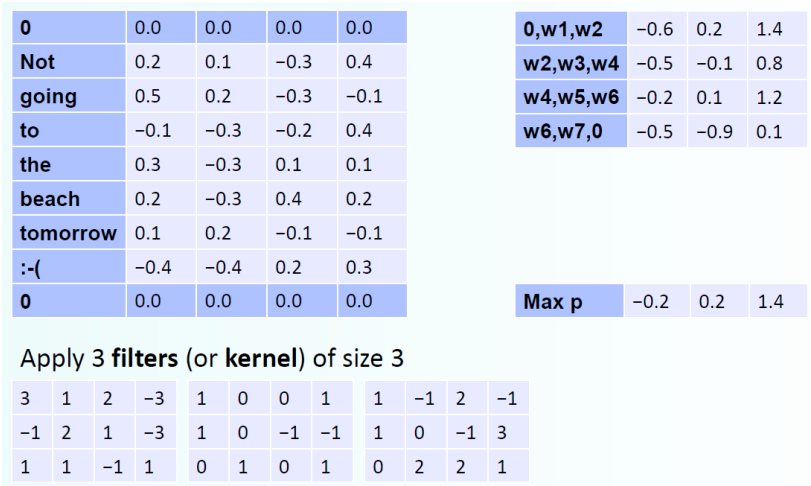



Figure 10.7: one-dimensional convolution, padded, with max pooling over time and stride = 2

if they are positive or negative, as on the contrary happens in IMDB. A solution goes under the name of **Distant Supervision**: pre-train the model over training data that is not 100% accurate. In this case, researchers assumed that tweets with positive emojis were positive, while tweets with negative emojis were negative. Therefore, they collect 10 million tweets treating tweets containing positive emojis as positive and tweets containing negative emojis as negative.

Another approach consists of generating word embeddings specific for sentiment analysis. The architecture used was similar to the Collobert one, but the neural network also outputs the context polarity.

The team SwissCheese won the 2016 SemEval competition presenting a convolutional neural network organized into the following procedures:

1. creation of word embeddings for initialization of the first layer. Word2vec on an unlabeled corpus of 200M tweets.

2. distant supervised phase, where the network weights and word embeddings are trained to capture aspects related to sentiment. Emoticons used to infer the polarity of a balanced set of 90M tweets. (Distant Supervision)

3. supervised phase, where the network is trained on the provided supervised training data.

Actually, the model was an ensemble of two convolutional neural networks having similar architectures but differing in the choice of certain parameters (such as the number of convolutional filters). networks were also initialized using different word embeddings and used slightly different training data for the distant supervised phase.

### Sentiment Classification from a single neuron

In 2017, Rafford, Jozefowicz, and Sutskever trained a model over **82 millions** reviews from Amazon [13]. The curious aspect was that, after training **one of the units had a very high correlation with sentiment**, resulting in state-of-the-art accuracy when used as classifier. Besides, the model can also be used to **generate text**, and by setting the value of the sentiment unit, one can **control the sentiment of the resulting text**.

Chapter 11

# Machine Translation

Machine translation (MT) consists of translating a text from one language to another automatically. At the moment, we can use MT whenever we do not need a flawless translation, such as recipes, web pages, or email translations. Or we can use MT as a pre-processing stage that is then revisioned by a human. However it is not still a reliable translation when the smallest detail is essential - medical translation in hospitals or emergency phone calls.

The idea of machine translation comes for the first time in 1946 when Booth and Weaver discussed MT at Rockefeller foundation in New York. The key idea of the time was that *there are certain invariant properties which are... to some statistically useful degree, common to all languages*[1]. After years of study, the MT field suffers a pessimism period in which various scientists stated fully automatic MT was impossible. Bar-Hillel argued that MT was too hard and should work semi-automatic instead of automatic (1959/1960)[2]. However, studies continued, especially in Europe, where due to the babel of languages, an automatic translation would have been a godsend. The first commercial systems were born in the 90s, in 2000's statistical MT takes off, and in 2015 neural MT becomes state-of-the-art.

[1] http://www.stanford.edu/class/linguist289/weaver001.pdf
[2] Bar-Hillel "Report on the state of MT in US and GB

## 11.1 Example of language similarities and divergences

It is true that languages have similarities, but it is also true that they also have lots of dissimilarities. And that is one of the main issues in MT. In this section, we try to list some of those similarities/dissimilarities. **Typology** is the study of systematic cross-linguistic similarities and differences.

A morpheme is the minimal meaningful unit of language. The field of linguistic study dedicated to morphemes is called **morphology**. When a morpheme can stand alone, it is considered a **lemma** (or root) because it has a meaning of its own (such as hope or cat). When it depends on another morpheme to express an idea, it is an **affix**.

Regarding morphology, there exist the following taxonomies among languages:

- **Isolating languages**: each word has **one morpheme** (Cantonese or Vietnamese).
- **Polysynthetic languages**: single word may have very many morphemes (Eskimo).

And:

- **Agglutinative languages**: morphemes have clear boundaries (Turkish).
- **Fusion languages**: single affix may have many morphemes (Russian).

Sentences from different languages can have changed sentence structures:

- **Subject–verb–object (SVO)**: a sentence structure where the subject comes first, the verb second, and the object third (English, Italian, German, French). For example, *William eats apples*, is an SVO sentence.
- **subject–object–verb (SOV)**: a sentence structure where the subject comes first, the object second, and the verb third (Japanese, Turkish, Hindi).
- **verb–subject–object (VSO)**: a sentence structure where the verb comes first, the subject second, and the object third (Irish, Classical Arabic), as in *eats William oranges*.

SOV is the most common order among the known languages. SVO and SOV together account for more than 75% of the world's languages.

Moreover, we can have segmentation variations, that is, not every writing system has word boundaries marked, such as Chinese or Japanese. Some languages tend to have sentences that are quite long, closer to English paragraphs than sentences, as Chiense or Modern Standard Arabic.

There are also differences concerning the inferential load:

- Some **cold** languages require the hearer to do more "figuring out" of who the various actors in the various events are (Japanese or Chinese).
- Other **hot** languages are pretty explicit about saying who did what to whom. (English).

Languages can also have different grammatical or semantic constraints. For instance, English has gender on pronouns, Mandarin not. Or in English, we always say brother, while in Chinese, if the brother is older, we call him gege, otherwise we call him didi.

There are also words not present in some languages (lexical gaps). For example, in Japanese, there are words to express privacy.

We can also distinguish languages in **Verb-framed** languages, or **Satellite-framed** ones:

- **Verb-framed**: mark direction of **motion on verb**. Spanish, for example, makes heavy use of verbs of motion like entrar, salir, subir, bajar ("go in", "go out", "go up", "go down"), which directly encode motion path.
- **Satellite-framed**: mark direction of **motion on satellite**. English verbs use particles to show the path of motion ("run into", "go out", "fall down"), and its verbs usually show manner of motion; thus, English is a satellite-framed language.

## 11.2 Classical MT methods

### 11.2.1 Direct

The direct translation method consists of four steps:

1. Do some morphological analysis.

2. Translate each word. So, we need a huge bilingual dictionary containing word-to-word translation information.

3. Do simple **local** reordering.

4. Do morphological generation to get the final result.

Figure 11.1 shows an example of translation using the direct method.

| | |
|---|---|
| Input: | Mary didn't slap the green witch |
| After 1: Morphology | Mary DO-PAST not slap the green witch |
| After 2: Lexical Transfer | Maria PAST no dar una bofetada a la verde bruja |
| After 3: Local reordering | Maria no dar PAST una bofetada a la bruja verde |
| After 4: Morphology | Maria no dió una bofetada a la bruja verde |

Figure 11.1: Translation of *Mary didn't slap the green witch* using the direct method

As one could imagine, the direct method is not so powerful. One of the problems that the next method, we will present, tries to solve is that the DM can only handle local reordering, and so it does not work whenever we need to do large movements.

### 11.2.2 Transfer

The idea is apply **contrastive knowledge**, that is knowledge about the difference between two languages. The transfer model consists of three steps:

1. **Analysis**: syntactically parse the Source language. The output of this step is a parse tree.

2. **Transfer**: use some **tranfer rules** to turn this parse tree into a parse tree for Target language.

3. **Generation**: generate the Target sentence from the resulting parse tree.

One of the oldest machine translation companies is Systran. Founded in 1968, Systran has been the principal automatic translator for almost forty years and combines direct and transfer methods.


Figure 11.2: Example of transfer rule from English to French. Note that this rule is not always true

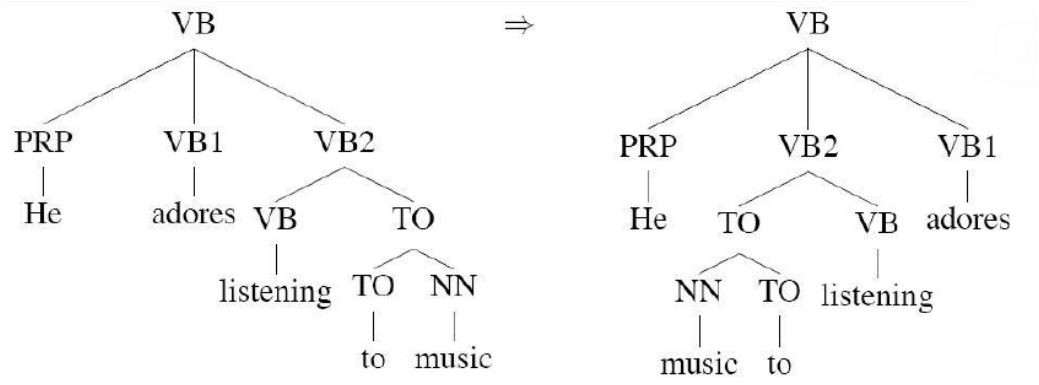



Figure 11.3: Second step of the transfer model. We use the transfer rules to transform the original English parse tree into a Japanese parse tree. Then, we use a word-to-word translation to get the final result.

Some problems of the transfer model were that grammar and lexicon are full of language-specific stuff. Besides, it is hard to build and maintain. However, the main problem of the transfer model is that it needs $N^2$ sets of transfer rules (for $N$ languages). The next model tries to put a patch on this.

### 11.2.3 Interlingua

Both the direct and the transfer methods were based on the syntactic structure of the sentence. The key idea of the interlingua method is to translate a sentence based on its meaning. The method consists of two steps:

1. Translate the Source sentence into **meaning representation**.
2. Generate the Target sentence from meaning.

We initially translate the sentence into a universal representation, such as the one shown in Figure 11.4. Then we do a translation from this universal

representation into the Target language. Therefore, if we want to provide translation for $N$ languages, the number of pairs to consider is just $N + 1$.

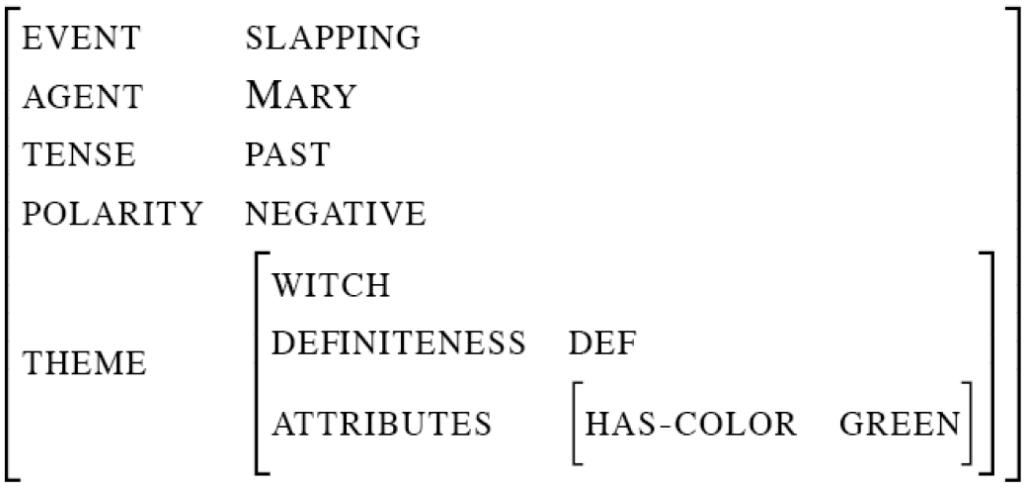



Figure 11.4: Meaning representation of the sentence *Mary did not slap the green witch*

We conclude the section listing the pros and cons of the direct and the interlingua methods.

Concerning the pros of the direct method:

✓ Fast.

✓ Simple.

✓ Cheap.

✓ No translation rules hidden in lexicon.

On the other hand, the cons are:

× Unreliable.

× Not powerful.

× Rule proliferation.

× Requires lots of context.

× Major restructuring after lexical substitution.

The pros of the interlingua method are:

✓ Avoids the $N^2$ problem.

✓ Easier to write rules.

While the cons are:

× Semantic is hard.

× Useful information lost (paraphrase).

## 11.3 Statistical MT

Translators often talk about two factors to maximize to get a good translation:

**Faithfulness** or fidelity: how close is the meaning of the translation to the meaning of the original. In other words, a translation has high faithfulness if it causes the reader to draw the same inferences as the original would have.

**Fluency** or naturalness: how natural the translation is, just considering its fluency in the target language.

In statistical MT, we try to maximize both the factors:

$$\overline{T} = \arg\max_T fluency(T) faithfulness(T, S) \tag{11.1}$$

That is the Bayes rule. Indeed, the task is translating from a foreign language sentence F (the source language) to an English sentence E (the target language). We want to find the best English sentence given the foreign one:

$$\begin{aligned} \overline{E} &= \arg\max_E P(E|F) \\ &= \arg\max_E \frac{P(F|E)P(E)}{P(F)} \end{aligned}$$

That, since P(F) is constant, becomes:

$$\overline{E} = \arg\max_{E} P(F|E)P(E) \tag{11.2}$$

We call the first factor **Translation Model**, and the second factor **Language Model**.

We can get the language model - the fluency - using a standard n-gram language model. We can train it on a large, unsupervised monolingual corpus for the target language E. We could also use a more sophisticated PCFG language model to capture long-distance dependencies. Terabytes of web data have been used to build an extensive 5-gram model of English.

Regarding the translation model - the faithfulness - we need to know, for every target language word, the probability of its mapping to every source language word. We will see that we can do this through parallel texts, which are sets of sentences of words for which we have the corresponding translated version in another language.

The job of the faithfulness model $P(F|E)$ is just to model a "bag of words", that is, the order of the words does not matter. Indeed, it is the language model $P(E)$ that puts words in order.

### 11.3.1 Phrase Based Machine Translation

In this method, we move the translation level from words to phrases, which are sequences of words. The algorithm to compute the faithfulness consists of three steps:

1. Segment E into a sequence of phrases $\overline{e_1}, \ldots, \overline{e_I}$.

2. Translate each phrase $\overline{e_i}$, into $\overline{f_i}$, based on **translation probability** $\phi(\overline{f_i}|\overline{e_i})$.

3. Then reorder translated phrases based on **distortion probability** $d(i)$ for the $i$-th phrase.

4. Compute:

$$P(F|E) = \prod_{I=1}^{I} \phi(\overline{f}_i, \overline{e}_i) d(i) \tag{11.3}$$

Assuming a **phrase aligned** parallel corpus is available or constructed that shows matching between phrases in E and F. Then we can approximate the translation probability with the maximum likelihood that is, we merely count frequencies:

$$\phi(\overline{f}|\overline{e}) = \frac{count(\overline{f}, \overline{e})}{\sum_{\overline{f}} count(\overline{f}, \overline{e})} \tag{11.4}$$

Where $count(\overline{f}, \overline{e})$ counts how many times, in the corpus, $\overline{e}$ is translated into $\overline{f}$.

The **distortion** is a measure of distance between positions of a corresponding phrase in the two languages. The distortion of the $i$-th phrase is the distance between the start of the phrase obtained translating $\overline{e}_i$ ($a_i$), and the end of the phrase obtained translating $\overline{e}_{i-1}$ ($b_{i-1}$) (see Figure 11.5). Typically we assume the probability of a distortion decreases exponentially with the distance of the movement:

$$d(i) = c \cdot \alpha^{a_i - b_{i-1}} \tag{11.5}$$

Where $0 < \alpha < 1$, is based on fit to phrase-aligned training data, while we use $c$ to normalize $d(i)$ so it sums to 1.

| Position | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| English | Mary | did not | slap | the | green | witch |
| Spanish | Maria | no | dió una bofetada a | la | bruja | verde |

| $a_i - b_{i-1}$ | 1 | 1 | 1 | 1 | 2 | -1 |
|---|---|---|---|---|---|---|

verde - la

bruja - verde

Figure 11.5: The top table shows a parallel text between an English and a Spanish sentence. The bottom table indicates $a_i - b_{i-1}$

Summarizing, we have:

- A language model $P(E)$
- A translation model $P(F|E)$

The decoder task is finding the most probable sentence E. However, it is not so easy. We have said we need a phrase-aligned parallel corpus to compute the translation model. But, unfortunately, we have only word alignments such as the one in Figure 11.6. A word alignment is a mapping between the

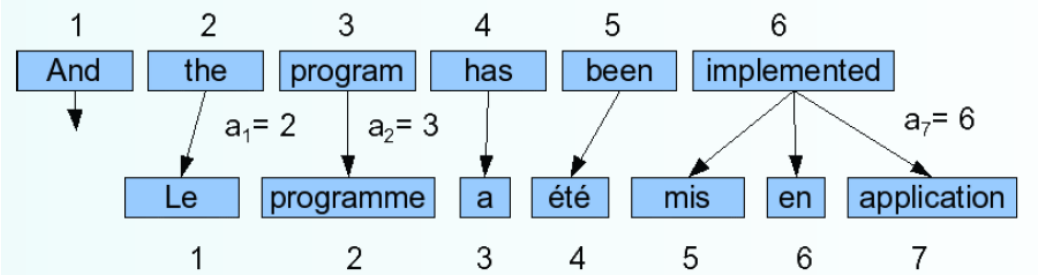



Figure 11.6: Word alignment between a sentence in English and the corresponding translated phrase in French

source words and the target words in a set of parallel sentences. We simplify the computations assuming that each foreign word comes from exactly one English word. The advantage is that we can represent an alignment by the index of the English word that the French word comes from. For example, in Figure 11.6, the alignment is 2,3,4,5,6,6,6.

A word in the foreign sentence that does not align with any word in the English sentence is called **spurious word**. We model these by pretending they are generated by English word $e_0$ or NULL (see Figure 11.7). We will also refer to word alignment as one-to-many alignment since each word in $F$ aligns to 1 word in $E$, but each word in $E$ may generate more than one word in $F$.

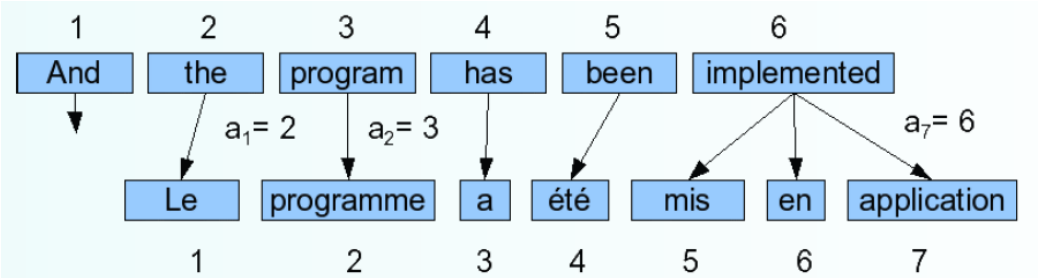



Figure 11.7: "a" does not align with any word in English. Thus, we connect "a" with NULL

A word alignment model gives us $P(F, E)$. We want this to train our phrase probabilities $\phi(\overline{f}_i, \overline{e}_i)$ as part of $P(F|E)$. A word-alignment model allows to compute the **translation probability** $P(F|E)$ by summing the probabilities of all possible 'hidden' alignments A between F and E:

$$P(F|E) = \sum_A P(F, A|E) \tag{11.6}$$

**Word Alignment Translation - IBM model 1**

What we will describe now is the IBM Model 1. It was proposed in a seminal paper by Brown et al. in 1993 as part of CANDIDE, the first complete SMT system. The model assumes following simple generative model of producing F from E = $e_1, e_2, \ldots e_I$:

1. Choose length $J$ of sentence $F = f_1, f_2, \ldots f_J$. Note that $J$ can be different from $I$.

2. Choose a one-to-many alignment $A = a_1, a_2, \ldots a_J$.

3. For each position in $F$, generate a word $f_i$ from the aligned word in $E : e_{aj}$.

To compute $P(F|E)$, we assume to have some length distribution $P(J|E)$ and that all alignments are equally likely. Since there are $(I + 1)^J$ possible alignments, the probability of have an alignment A given a sentence E is:

$$P(A|E) = P(A|E, J)P(J|E) = \frac{P(J|E)}{((I + 1)^J} \tag{11.7}$$

Assuming $t(f_x, e_y)$ is the probability of translating $e_y$ as $f_x$, the probability to obtain the sentence $F = f_1, f_2, \ldots f_J$, from the sentence $E$ with alignment $A$ is:

$$P(F|E, A) = \prod_{j=1}^{J} t(f_j, e_{a_j}) \tag{11.8}$$

Eventually, we can compute $P(F|E)$ by marginalizing with respect all the possible alignments $A$:

$$P(F|E) = \sum_A P(F|E, A)P(A|E) = \sum_A \frac{P(J|E)}{(I + 1)^J} t(f_j, e_{a_j}) \tag{11.9}$$

The goal of the model is to find the most probable alignment given the parameterized model:

$$\hat{A} = \arg\max_{A} P(F, A|E) = \arg\max_{A} \sum_{A} \frac{P(J|E)}{(I+1)^J} t(f_j, e_{a_j}) \tag{11.10}$$

That, since $\frac{P(J|E)}{(I+1)^J}$ is constant becomes:

$$\hat{A} = \arg\max_{A} P(F, A|E) = \arg\max_{A} \sum_{A} t(f_j, e_{a_j})$$

Since translation choice for each position j is independent, the product is maximized by maximizing each term of A = $a_1, a_2, \ldots, a_J$:

$$a_j = \arg\max_{o \leq i \leq I} t(f_j, e_i) 1 \leq j \leq J \tag{11.11}$$

We can train the alignment probabilities by taking a parallel corpus, then splitting each document into sentences, and each sentence into word alignments, and finally using EM (Expectation-Maximization) to train the word alignments.

Initially, all translation probabilities will be equally likely. Then, according to the data set we have, we will update the probabilities. For instance, if we frequently observe "la" and "the" co-occur, we can increase P(la—the). However, increasing a probability will decrease others.

**Example 11.3.1** Sample EM trace for Alignment

We have a small corpus with only two parallel sentences: "green house - casa verde" and "the house - la casa". We initially assume that all translation probabilities are equally likely:

| | **verde** | **casa** | **la** |
|---|---|---|---|
| **green** | 1/3 | 1/3 | 1/3 |
| **house** | 1/3 | 1/3 | 1/3 |
| **the** | 1/3 | 1/3 | 1/3 |

For the first sentence, the possible alignments are $a_{1,1}$ : green $\rightarrow$ casa, house $\rightarrow$ verde, and $a_{1,2}$ : green $\rightarrow$ verde, house $\rightarrow$ casa. On the other hand, the possible aligments for the second sentence are

$a_{2,1}$ : the $\rightarrow$ la, house $\rightarrow$ casa, and $a_{2,2}$ : the $\rightarrow$ casa, house $\rightarrow$ la. According to the translation probabilities shown in the above table, we have that:

$$P(a_{1,1}, f|e) = 1/3 \times 1/3 = 1/9$$
$$P(a_{1,2}, f|e) = 1/3 \times 1/3 = 1/9$$
$$P(a_{2,1}, f|e) = 1/3 \times 1/3 = 1/9$$
$$P(a_{2,2}, f|e) = 1/3 \times 1/3 = 1/9$$

That normalizing (11.10) become:

$$P(a_{1,1}, f|e) = \frac{1/9}{2/9} = 1/2$$
$$P(a_{1,2}, f|e) = \frac{1/9}{2/9} = 1/2$$
$$P(a_{2,1}, f|e) = \frac{1/9}{2/9} = 1/2$$
$$P(a_{2,2}, f|e) = \frac{1/9}{2/9} = 1/2$$

At this point, we recompute the translation probabilities according to the frequency each word co-occur in the alignments:

- "green" and "casa" co-occur 1 time.
- "green" and "verde" co-occur 1 time.
- "green" and "la" co-occur 0 times.
- "house" and "verde" co-occur 1 time.
- "house" and "casa" co-occur 2 times.
- "house" and "la" co-occur 1 time.
- "the" and "verde" co-occur 0 times.
- "the" and "casa" co-occur 1 time.
- "the" and "la" co-occur 1 time.

Then, we sum the translation probabilities of co-occurring words obtaining the following table:

| | **verde** | **casa** | **la** |
|---|---|---|---|
| **green** | 1/2 | 1/2 | 0 |
| **house** | 1/2 | 1/2 + 1/2 | 1/2 |
| **the** | 0 | 1/2 | 1/2 |

Note that now if we sum the second row, we get $2 > 1$. As before, we have to normalize so that all rows sum to one:

| | **verde** | **casa** | **la** |
|---|---|---|---|
| **green** | 1/2 | 1/2 | 0 |
| **house** | 1/4 | 1/2 | 1/4 |
| **the** | 0 | 1/2 | 1/2 |

Then we restart from the beginning again, computing the alignment probabilities and re-weighting the translation probabilities. The process continues until convergence.

The IBM model is the simplest one. There exist also IBM model 2, 3 and 4 (See Figure 11.8).

| | |
|---|---|
| **IBM Model 1** | lexical translation |
| **IBM Model 2** | adds absolute reordering model |
| **IBM Model 3** | adds fertility model |
| **IBM Model 4** | relative reordering model |

Figure 11.8: IBM models

### Phrase Alignment Translation

Now that we know how to compute word alignments, we would like to find a way to obtain phrase[3] alignments from word alignments. The advantages

[3]remember, phrase = sequence of words

of using a phrase alignment approach are many, for instance:

- Many-to-may translations (phrase alignment translations) can handle non-compositional phrases, which are phrases whose meaning does not come from their constituent words. For instance, "it is raining cats and dogs" in Italian would be laterally translated into "piovono cani e gatti", even though the right translation is "piove a dirotto".
- Use of local context in translation.
- The more data, the longer phrases can be learned.

The idea to obtain phrase alignments from word alignments is to combine both the word alignments from the English to the foreign language ($E \rightarrow F$) and the word alignments from the foreign to the English language ($F \rightarrow E$).

So, we take the word alignments as in the top of Figure 11.9, we compute the intersection, and finally, we add alignments from union to intersection. Finally, we split the result into all possible phrase alignments that are consistent. Intuitively, we have consistency when phrase alignment contains all alignment points for all covered words. Figure 11.10 should clarify the concept.

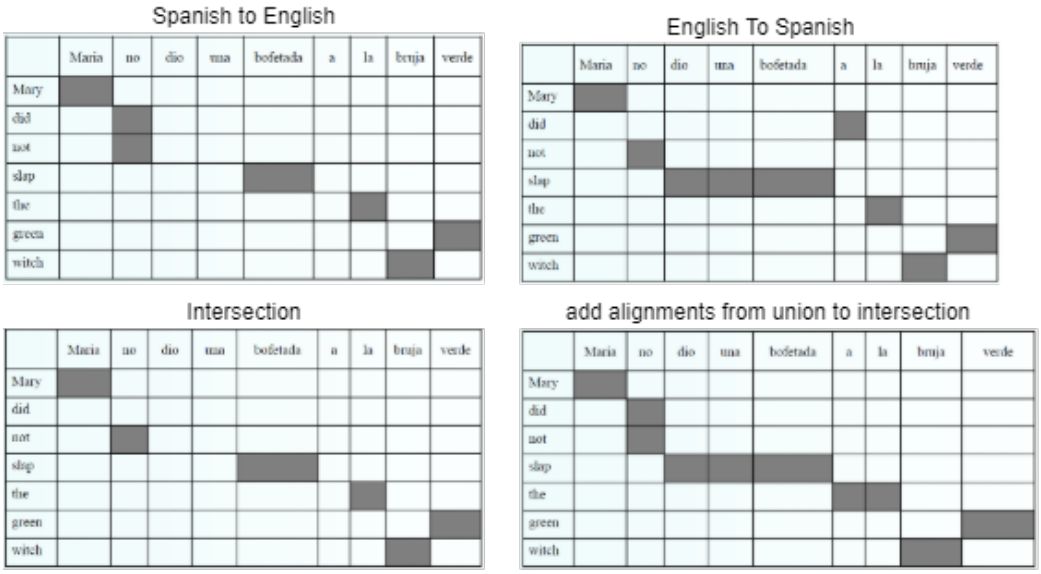



Figure 11.9: Word alignments on both directions

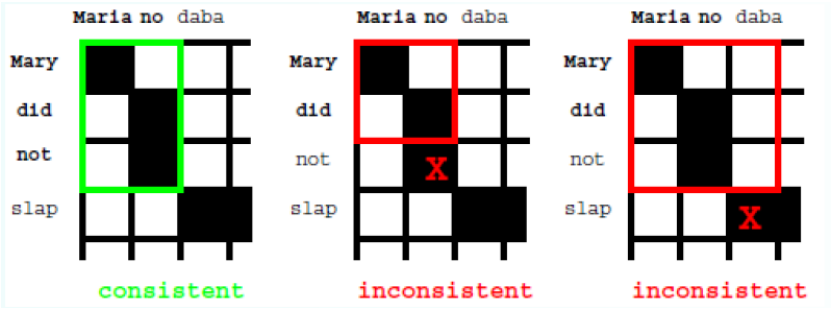



Figure 11.10: In the first example, the groups produced are not broken: the green rectangle encloses the translation of "Maria" and "no" completely. In the second example, that is not true because "not" is off the red rectangle.

Once we have the phrase alignment, we can proceed with the decoding. We have already mentioned how to calculate the translation model in Formula 11.3.

One possible way to translate the sentence is to consider all the possible phrase translations: try all possible translation hypotheses and return the one with the highest translation probability. We can see the process as a search in a tree of hypotheses (see Figure 11.11).

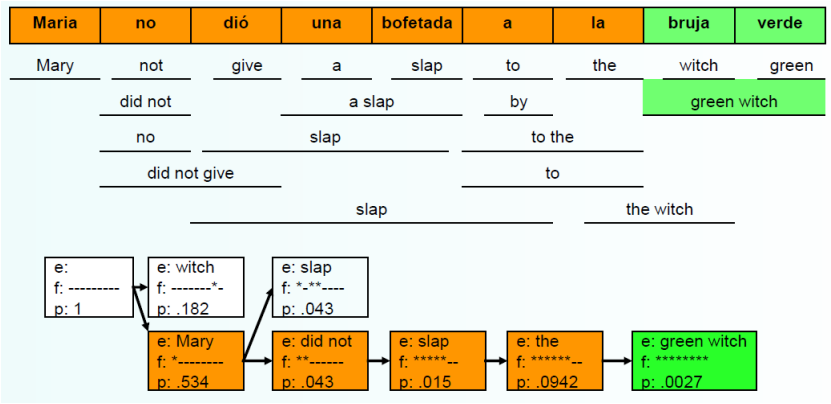



Figure 11.11: Hypotheses expansion example

However, this method has an enormous problem: the number of hypotheses is exponential with respect to sentence length. So, decoding is an NP-complete problem. Therefore, we need to reduce the search space, for instance using:

- **hypothesis recombination**: we combine those paths that bring to the same hypothesis. Since those paths can have different overall probabilities, when we combine them, we drop the weaker path (the one with the lowest probability).
- **Pruning**: organize hypothesis in **stacks**, e.g. by the same foreign words covered, the same number of foreign words covered, or the same number of English words produces. We compare hypotheses in stacks, and we discard the bad ones.

### 11.3.2 Evaluation MT

Human subjective evaluation is the best, but it is time-consuming and expensive. An automated evaluation comparing the output to multiple human reference translations is cheaper and correlates with human judgments.

A classic measure is the **edit cost**, which is the number of changes that a human translator must make to correct the MT output. The changes can be in terms of the number of words changes, amount of time taken to edit the text or number of keystrokes needed to edit.

In automatic evaluation, we collect one or more human **reference translations** of the source. Then, we compare MT output to these reference translations. The score result is based on similarity to the reference translation. One of the most used scores is the BLEU, which determines the number of n-grams of various sizes that the MT output shares with the reference translations. More in detail:

1. We compute the n-gram precision overall n-grams up to size N (typically 4) (Figure 11.12). The precision is the number of shared n-grams between the translated sentence and the reference human translations divided by the number of n-grams:

$$p_n = \frac{\text{shared n-grams}}{\text{total n-grams}} \tag{11.12}$$

2. Average n-gram precision using geometric mean:

$$p = \sqrt[N]{\prod_{n=1}^{N} p_n} \tag{11.13}$$

Concerning the example in Figure 11.12, applying 11.13, we obtain:

$$\text{Cand1} : p = \sqrt[w]{\frac{5}{6}\frac{1}{5}}$$
$$\text{Cand2} : p = \sqrt[w]{\frac{7}{10}\frac{1}{3}}$$

3. Note that we could have a decoder that translates only the part of the sentence in which, certainly, the translation is right. So, we would get a high score for a partial translation. Therefore, we introduce a penalty for translations that are shorter than the reference translations. Let $c$ be the candidate sentence length and $r$ the effective reference length:

$$BP = \begin{cases} 1, & \text{if } c > r \\ e^{1-r/c}, & \text{if } c \leq r \end{cases} \tag{11.14}$$

In our example:

$$\text{Cand1} : BP = e^{1-7/6} = 0.846 \quad (c = 6,\ r = 7)$$
$$\text{Cand2} : BP = 1 \quad (c = 10,\ r = 7)$$

4. We compute the final BLEU score as:

$$BLEU = BP \times p \tag{11.15}$$

So, in our example:

$$\text{Cand1} : BLEU = 0.846 \times 0.408 = 0.345$$
$$\text{Cand2} : BLEU = 1 \times 0.483 = 0.483$$

BLEU has been shown to correlate with human evaluation when comparing outputs from **different SMT systems**. However, it does not correlate with human judgements when comparing SMT systems with manually developed MT or MT with human translations. OTher MT evaluation metrics have been proposed that claim to overcome some of the limitations of BLEU.

## 11.4 Neural MT [TO DO]

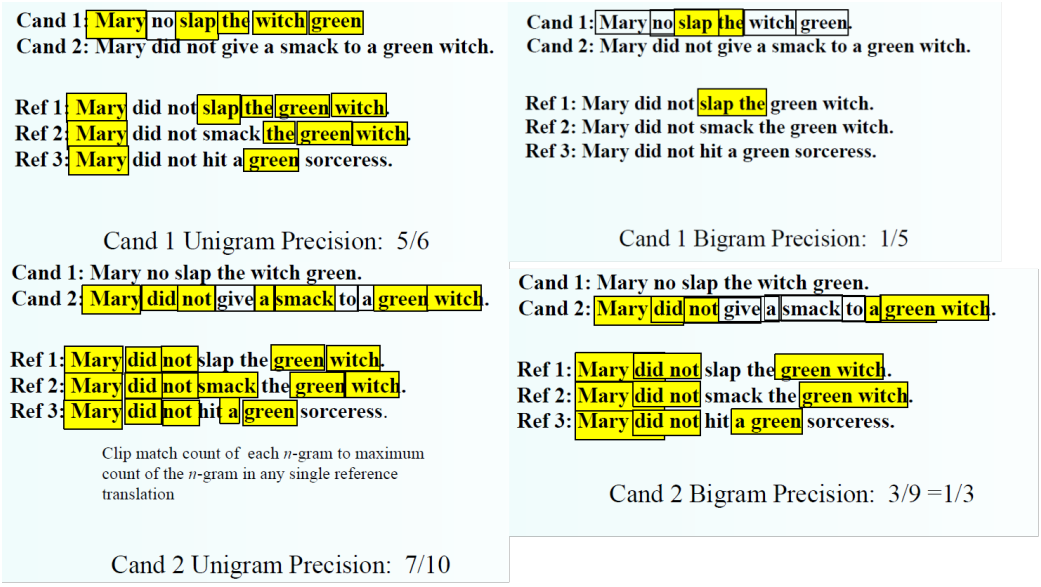



Figure 11.12: Precision calculation. The first sentences are the candidate translations, while the bottom sentences are the human translations.

Chapter 12

# Transformer and Attention

We have seen that sequence to sequence models with **attention** (Neural MT [TO DO]) are quite effective in transduction tasks, which are those where we take a sequence in input, and we generate another sequence in output. Their **sequential nature** limits though parallelism. The transformer transduction model relies entirely on self-attention to compute representations of its input and output. Thus, it does not use neither sequence aligned RNNs nor convolutions. The training costs are **reduced** by 1-2 orders of magnitude.

## 12.1 From RNN to Self-Attention

Suppose to have a sequence such as the one shown in Figure 12.1, where "chef" and "was" are related, but they are separated by N words. An RNN will take O(N) steps for this pair to interact. Moreover, in the meantime, the RNN could lose information about "chef", making long-distance dependencies learning very hard.
Another problem is that the linear order of words is "baked in". However, we know that linear order is not the right way to think about sentences because we may have a different way of saying the same thing, and so we would have a bias toward a specific word order rather than another one.

Finally, probably the worst problem is that forward and backward passes have O(N) unparallelizable operations. Still, GPUs can perform a bunch of independent computations at once. But we can not compute future RNN hidden states in full before we have not computed past RNN hidden states.

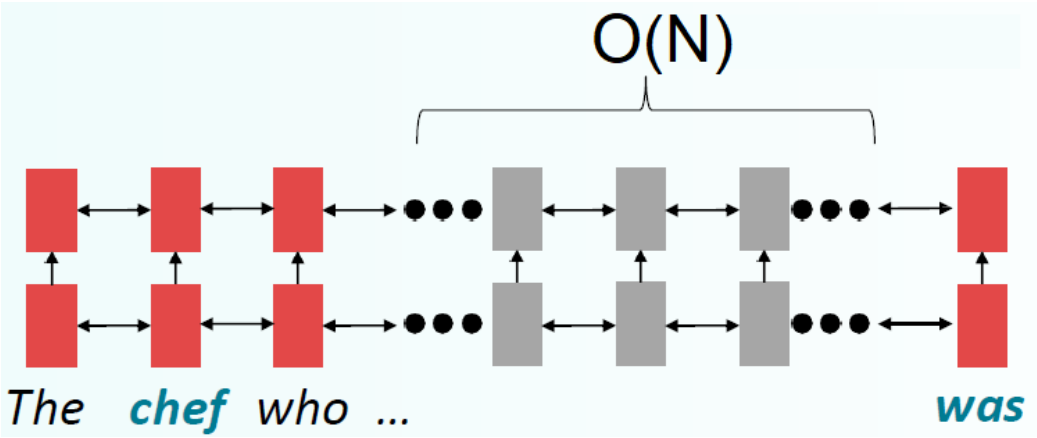



Figure 12.1: RNNs take O(N) steps for distant word pairs to interact.

In alternative to RNNs, we could use word window models, which work with a sliding window aggregating local contexts. In this way, the number of unparallelizable operations does not increase with the sequence length. However, it may happen that "chef" and "was" are so distant that the sliding window can not consider both together, and therefore we ignore the dependency.

The final alternative we present is by using the (self-)attention.

### 12.1.1 Attention

Given a set of vector **values**, and a vector **query**, **attention** is a technique to compute a weighted sum of values, dependent on the query. The weighted sum is a **selective summary** of the information contained in the values, where the query determined which values to focus on. So, attention is a way to obtain a **fixed-size representation of an arbitrary set of representations** (the values), dependent on some other representation (the query).

Attention operates on :

- A set of **queries** $q_1, q_2, \ldots, q_T$: Each query is $q_i \in \mathbb{R}^{d_2}$.
- A set of **keys** $k_1, k_2, \ldots, k_T$: Each key is $k_i \in \mathbb{R}^{d_1}$.
- A set of **values** $v_1, v_2, \ldots, v_T$: Each value is $v_i \in \mathbb{R}^{d_1}$. In NMT, values and keys coincide ($k_i = v_i$).

**keys**, and **values**. Computing the attentions for a given query $s \in \{q_1, q_2, \ldots, q_T\}$ requires three steps:

1. Compute the attention scores $e \in \mathbb{R}^N$. There are several ways to compute $e$. Usually, we use the basic dot-product:

$$e_i = s^T \cdot k_i \tag{12.1}$$

   Note that we can apply 12.1 only if $d_1 = d_2 = d$. Note also that when dealing with normalized vectors, the dot product measures the cosine similarity between $s^T$ and $k_i$ (3.1).

2. Take softmax to get an **attention distribution** $\alpha$:

$$\alpha = \text{softmax}(e) \in R^T \tag{12.2}$$

3. Use attention distribution to take weighted sum of values:

$$a = \sum_{i=1}^{N} \alpha_i \cdot v_i \in R^{d_1} \tag{12.3}$$

   Thus obtaining the **attention output a** (sometimes called the **context vector**).

In transformers what we actually use is the **self-attention**. In **self-attention**, the queries, keys, and values are drawn from the same source. For example, if the output of the previous layer is $x_1, \ldots, x_T$, (one vector per word) we could let $v_i = k_i = q_i = x_i$, that is use the same vector for all of them. Following the steps written above, we compute the self-attention by:

1. Computing **key-query** affinities:

$$e_{ij} = q_i^T \cdot k_j$$

2. Computing attention weights from affinities (softmax):

$$\alpha_{ij} = \frac{e^{e_{ij}}}{\sum_j e^{e_{ij}}}$$

3. Computing outputs as weighted sum of **values**:

$$output_i = \sum_j \alpha_{ij} v_j$$

Using attention allow us to solve the problems of the previous solutions (RNNs and word embedding models):

- We can compute the attentions in parallel.
- All words interact at every time. Therefore we do not have to think about non considered dependencies as in word embeddings.

However, using attention also bring some problem:

- Self-attention does not build in order information. So, we drop information about the positions of the words.
- There are no elementwise nonlinearities in self-attention. So, if we stack more self-attention layers, we merely re-averages value vectors.
- We need to ensure we do not look at the future when predicting a sequence. That is, if we are computing $e_{ij}$, we do not want to consider the keys $k_z$ such that $j \geq j$.

Let us try to find a solution for all these points:

- We can encode the order of the sentence in our keys, queries, and values. So, we can consider representing each word position as a vector:

$$p_i \in \mathbb{R}, \text{for } i \in \{1, 2, \ldots, T\} \text{ are position vectors} \tag{12.4}$$

And then incorporate this info adding the $p_i$ to our inputs. Let $\hat{v}_i, \hat{q}_i$, and $\hat{k}_i$ be our old values, keys, and queries:

$$\begin{aligned} v_i &= \hat{v}_i + p_i \\ q_i &= \hat{q}_i + p_i \\ k_i &= \hat{k}_i + p_i \end{aligned}$$

In deep self-attention networks, we do this at the first layer. We could also concatenate them as well, but people mostly just add $p_i$.

A way to compute $p_i$ is by using a **sinusoidal position representation**, which concatenate sinusoidal functions of varying periods:

$$p_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \cdots \end{pmatrix} \tag{12.5}$$

Most of the modern system uses a **learned absolute position representation** that let all pi be learnable parameters.

- We can add a **feed-forward network** to apply a non-linear function to each self-attention output. Note that to maintain the parallelization, we need a network for each self-attention output, as shown in Figure 12.2.

- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$:

$$e_{ij} = \begin{cases} q_i^T \cdot k_j, & \text{if } j < i \\ -\infty, & \text{if } j \geq i \end{cases} \tag{12.6}$$

  So, we mask the future by artificially setting attention weights to 0.

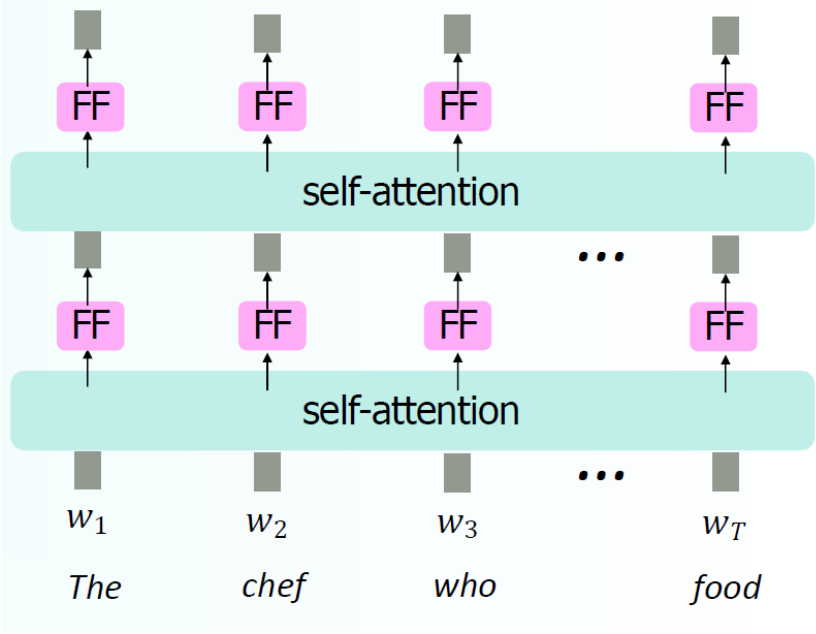



Figure 12.2: Solution for the problem that is described in the second point. Several Feed-forward networks work in parallel to maintain the parallelization pro of employing attention.

## 12.2 Transformers

A transformer consists of two parts:

- The **encoder** mpas an input sequence og symbol representations $(x_1, \ldots, x_n)$ ti a sequence of continuous representations $z = (z_1, \ldots, z_n)$.
- Given $z$, the **decoder** then generates an output sequence $(y_1, \ldots, y_m)$ of symbols one element at a time.

At each step the model is **auto-regressive**, consuming the previously generated symbols as additional input when generating the next. Actually, as shown in Figure 12.3, the transformer has a stack of encoders. And the last encoder sends its output to a stack of decoders. The number of encoders and decoders is the same.

The encoder consists of two layers: a self-attention layer and several Feed-Forward Neural Networks (with ReLu as activation) - one for each self-attention layer output. The decoder has both those layers, but between them there is an attention layer that helps the decoder focus on relevant parts of the input sentence.

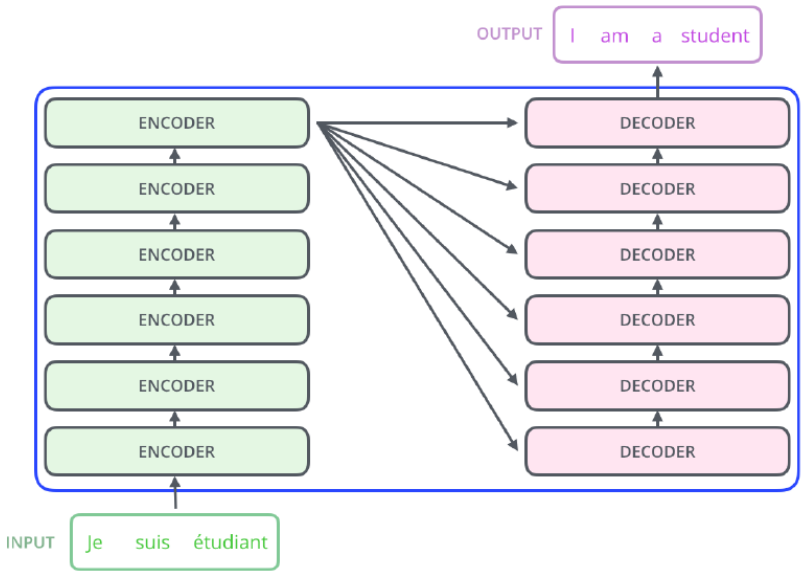



Figure 12.3: A transformer has a stack of encoders and decoders

Each input word is turned into a **word embedding** vector. The vectors flow through each of the two layers of the encoder. The word in each position flows through its own path in the encoder. There are **dependencies** between these paths in the self-attention layer. On the other hand, the **feed-forward**

**layers are independent** and thus the various paths can be executed in **parallel** (see Figure 12.4).

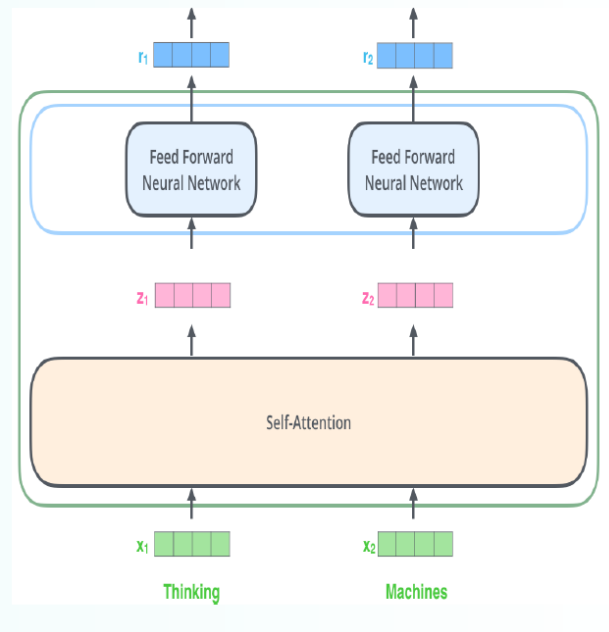



Figure 12.4: The encoder more in detail

### 12.2.1 Self-Attention

As the model processes each word, self-attention allows it to look at **other words** in the input sequence for clues that can help lead to a better encoding for this word. For instance, when in Figure 12.5, the model is processing the word "it", self-attention allows it to associate "it" with "animal". So, self-attention allows the Transformer to bake into one word hidden vector the "understanding" of other relevant words.

For each word, we create a **query** vector, a **key** vector, and a **value** vector. These vectors are created by multiplying the embeddings by three matrices that were trained during the training phase:

$$q_i = W^Q X_i$$
$$k_i = W^K X_i$$
$$v_i = W^V X_i$$

The computations are then the same described in the previous section:

1. Calculate the score: $E = QK^T$
2. Apply the softmax: $S = \text{softmax}(E)$
3. Finally, compute the weighted sum: $Z = SV$

That summarizing all the computations in one step is like calculating:

$$Z = \text{softmax}(QK^T)V \tag{12.7}$$

Note that with only one self-attention, a word has only one way to interact with others. We can expand the model's ability to focus on different positions. So, instead of using one self-attention layer, we can use several self-attention layers and then concatenate all the attention heads.

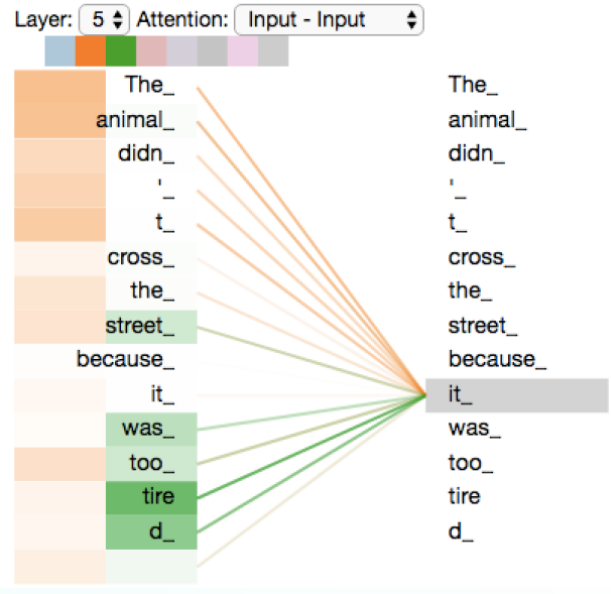



Figure 12.5: Self-attention example. Different attention heads focus on different aspect. One attention head is focusing most on "the animal", another is focusing on "tired"

The model described so far has no way to account for the order of the words in the input sequence. So, as already suggested, we add a vector to each input, learned though training, which helps to determine the position of each word, or the distance between different words in the sequence.

## 12.3 Pretraining

We usually develop transformers through two steps:

1. **Pretraining**: **semi-supervised** training on large amount of texts (books, Wikipedia, etc.). The model is trained on a certain task that enables it to grasp patterns in language. For instance, we can pretrain a transformer through language modelling. By the end of the training process, the transformer has already language-processing abilities capable of empowering many models we later need to build and train in a supervised way.

2. **Supervised** training on a specific task with a labeled data-set.

Pretraining can improve NLP applications by serving as parameter initialization. Thus, pretraining a language model provides base parameters $\hat{\theta}$. The pretraining may also help because stochastic gradient descent sticks (relatively) close to $\hat{\theta}$. So, since the gradient descent will start from a good position, the changes will be small.

The neural architecture influences the type of pretraining, and natural use cases:

- **Decoders** (get unidirectional context, that is, the output depends only on the preceding words): we pretrain on a language model.

- **Encoders** (get bidirectional context, that is, the output depends both on future words and past words).

- **Encoder-Decoders**.

### 12.3.1 Pretraining decoders

When using language model pretrained decoders, we can ignore that they were trained to model $p(w_t|w_{1:t-1}$. We can fine-tune them by training a linear classifier, which tries to guess whether the last word generated is correct or not. Then the weights are fine-tuned by back-propagation.

2018's GPT was a big success in pretrainin a decoder.

### 12.3.2 Pretraining encoders

Encoders get bidirectional context, so we can not do language modeling. The idea is to replace some fraction of words in the input with a special [MASK] token; fine-tune the encoder to predict these words. The method is called Masked LM.

### 12.3.3 Pretraining encoders-decoders

For encoder-decoders, we could do something like language modeling, but where a prefix of every input is provided to the encoder and is not predicted. The encoder portion benefits from bidirectional context; the decoder portion is used to train the whole model through language modeling.

## 12.4 BERT

BERT is the first, deeply, bidirectional, unsupervised language representation, pre-trained using only a plain text corpus. As already seen in section 12.3.2, BERT masks out $k\%$ (typically $k = 15\%$) of the input words and then predicts the masked words. Note that a high k would bring too much masking, and therefore not enough context. On the other hand, too little masking would have made the training too expensive.

Moreover, BERT does not mask all the words to predict, instead:

- 80% of the time, replace with [MASK].
  For instance, *went to the* ***store*** $-z >$ *went to the [MASK].*
- 10% of the time, replace with **random word**.
  For instance, *went to the* ***store*** $- >$ *went to the running.*
- 10% of the time, keep the **same** word.
  For instance, *went to the* ***store*** $- >$ *went to the store.*

BERT uses a variant of the wordpiece model[1]. The idea is to put relatively common words (e.g. at, Fairfax 1910s) in the vocabulary. All the other words

[1] WordPiece is a subword segmentation algorithm used in natural language processing. The vocabulary is initialized with individual characters in the language, then the most frequent combinations of symbols in the vocabulary are iteratively added to the vocabulary.

are built from wordpieces. For instance, hypatia is built combining: h ##yp ##ati and ##a.

Another task used by BERT in the pretraining phase to train a model to understand sentence relationships is the **Next Sentence Prediction**. The task is, given two sentences A and B, to predict whether Sentence B is the actual sentence that follows Sentence A, or a random sentence. 50% of the time B is the actual next sentence that follows A and 50% of the time it is a random sentence.

There are two versions of BERT:

- BERT BASE - 12 layer model Comparable in size to the OpenAI Transformer in order to compare performance.
- BERT LARGE - A huge 24 layer model which achieved the state of the art results.

BERT reached state-of-the-art results in the General Language Understanding Evaluation (GLUE) benchmarks. Figure 12.6 shows the results in several tasks, such as QQP (Quora Question Pairs), QNLI (Question Natural Language Inference), or MultiNLI. In MultiNLI the model must say given a premise and a hypothesis, whether the premise entails the hypothesis or the premise contradicts the hypothesis or the premise is neutral concerning the hypothesis(see Figure 13.1).

| System | MNLI-(m/mm) 392k | QQP 363k | QNLI 108k | SST-2 67k | CoLA 8.5k | STS-B 5.7k | MRPC 3.5k | RTE 2.5k | **Average** - |
|---|---|---|---|---|---|---|---|---|---|
| Pre-OpenAI SOTA | 80.6/80.1 | 66.1 | 82.3 | 93.2 | 35.0 | 81.0 | 86.0 | 61.7 | 74.0 |
| BiLSTM+ELMo+Attn | 76.4/76.1 | 64.8 | 79.9 | 90.4 | 36.0 | 73.3 | 84.9 | 56.8 | 71.0 |
| OpenAI GPT | 82.1/81.4 | 70.3 | 88.1 | 91.3 | 45.4 | 80.0 | 82.3 | 56.0 | 75.2 |
| $\text{BERT}_{\text{BASE}}$ | 84.6/83.4 | 71.2 | 90.1 | 93.5 | 52.1 | 85.8 | 88.9 | 66.4 | 79.6 |
| $\text{BERT}_{\text{LARGE}}$ | **86.7/85.9** | **72.1** | **91.1** | **94.9** | **60.5** | **86.5** | **89.3** | **70.1** | **81.9** |

Figure 12.6: BERT results in GLUE benchmarks

Have been shown that increasing the size of the model leads to better results.

Chapter 13

# Analysis of Language models

In this chapter, we will try to analyze what can be learned via language model (LM) pretraining. For instance, we can do unsupervised Named Entity Recognition using the BERT masked language model.

### 13.0.1 LM Limits

LM exhibits surprising abilities in several language tasks. But do they understand language? Consider, for instance, the natural language inference task, as encoded in the Multi-NLI dataset. In MultiNLI the model must say given a premise and a hypothesis, whether:

- the premise entails the hypothesis or
- the premise contradicts the hypothesis or
- the premise is neutral concerning the hypothesis.

An LM model can reach 95% accuracy. But, does that model understand the language, or is it using simple heuristics to get that good accuracy? A **diagnostic test set** is carefully constructed to test for a specific skill or capacity of our neural model. For example, HANS (Heuristic Analysis for NLI Systems) tests syntactic heuristics in NLI. Some heuristics are shown in figure 13.2.

McCoy et al. in 2019 took four strong MNLI models, with the accuracies on the original test set (in a domain) shown in figure 13.3. Looking at figure

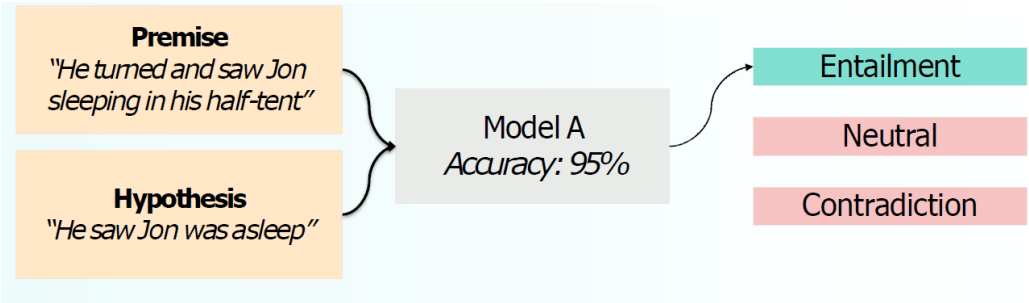



Figure 13.1: Given that premise and that hypothesis, the model concludes that there is an entailment relationship between the two sentences

| Heuristic | Definition | Example |
|---|---|---|
| Lexical overlap | Assume that a premise entails all hypotheses constructed from words in the premise | **The doctor** was **paid** by **the actor**. $\xrightarrow[\text{WRONG}]{}$ The doctor paid the actor. |
| Subsequence | Assume that a premise entails all of its contiguous subsequences. | The doctor near **the actor danced**. $\xrightarrow[\text{WRONG}]{}$ The actor danced. |
| Constituent | Assume that a premise entails all complete subtrees in its parse tree. | If **the artist slept**, the actor ran. $\xrightarrow[\text{WRONG}]{}$ The artist slept. |

Figure 13.2: Some heuristic models could use to reach high accuracies.

13.4, we can notice that accuracy is high when heuristics work, but where syntactic heuristics fail, accuracy is very low.

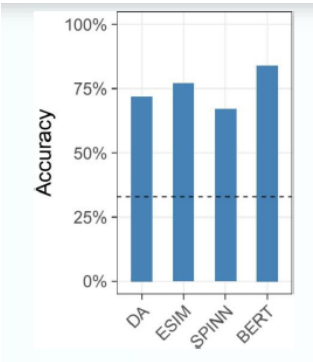



Figure 13.3: Accuray in original corpus

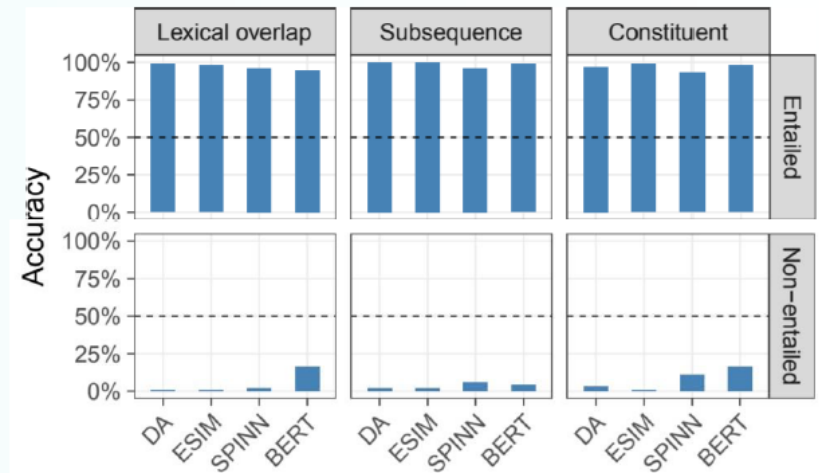



Figure 13.4: Accuracy when heuristics work and do not work

Chapter 14

# Reading comprehension

In Question Answering (QA) tasks, the model must answer questions. We can divide the QA tasks into four categories:

- **Open Domain Question Answering**: Relies on external memory consisting of large corpora of documents, that may be pre-proceesed to build IR systems (inverted index, ranking).
- **Answer Selection**: Concentrates on choosing on the fly most likely passage containing the answer from a small given list.
- **Dialog System**: Must provide answers to questions and also keep memory of dialog context, remembering previous statements.
- **Reading Comprehension**: Find the answer within a given paragraph (no pre-processing, no memory)

In this chapter, we will discuss reading comprehension. More specifically, in reading comprehension, given a passage and a question, the model must find the answer in the passage. Reading comprehension is interesting because:

- It is useful for many practical applications.
- It is an important testbed for evaluating how well computer systems understand human language.
- Many other NLP tasks can be reduced to a reading comprehension problem.

## 14.1 SQuAD

The dataset used to evaluate models attempting QA tasks is the Stanford Question Answering Dataset (SQuAD) [1]. In SQuAD 1, each sample has the question, the passage and a set of three gold answers. We score a system by two metrics:

Exact match (EM): 1/0 Accuracy on whether the system match one of the three answers.

F1: Take system and each gold answer as bag of words and compute the F1 score. Then score is the (macro-)average of per-question F1 scores.

F1 measure is sees as more reliable and taken as primary. Indeed, it is less based on choosing exactly the same span that human chose, which is susceptible to various effects, including line breaks. Both the metrics ignore punctuation and articles (a, an, the).

A defect of SQuAD 1.0 is that all questions have an answer in the paragraph. So, systems (implicitly) rank candidates and choose the best one. In other words, they do not know if they are actually answering the question. In SQuAD 2.0, 1/3 of the training questions have no answer, and about 1/2 of the dev/test questions have no answer. For NoAnswer examples, NoAnswer receives a score of 1, and any other response gets 0 for both exact match and F1.

SQuAD has a number of other key limitations too:

- Only span-based answers (counting, implicit why, yes/no answers are not present).
- Question were constructed looking at the passages.
- Barely any multi-fact/sentence inference beyond co-reference.

Nevertheless, it is a well-targeted, well-structured, clean data-set.

## 14.2 Neural Models

The problem formulation to solve SQuAD is as follow:

[1] leaderboard at https://rajpurkar.github.io/SQuAD-explorer/

Given an input passage $C = (c_1, c_2, \ldots, c_N)$ and a question $Q = (q_1, q_2, \ldots, _M)$ such that $\forall i \; c_i, q_i \in V$, the model returns as output the first and the last positions in the passage where the answer is present. Formally, the output are $1 \leq start \leq end \leq N$.

In this chapter, we will see a solution that uses LSTM (BiDAF) and another that employees BERT.

### 14.2.1 BiDAF

The general architecture of BiDAF is shown in Figure 14.1. The system works as follow [14]:

1. **Character and Word embed layer**: use a concatenation of word embeddings (GLoVe) and character embedding (CNNs over character embeddings) for each word in context (passage) and query.

2. **Phrase embed layer**: use two **bidirectional** LSTMs separately to produce contextual embeddings for both context and query.

3. **Attention flow layer**: returns three outputs. First, compute a similarity score for evert pair $(c_i, q_j)$. Then, it computes the **context-to-query attention**, that is, for each context word, choose the most relevant words from the query words. Eventually, it computes the **query-to-context attention**, that is, choose the context words that are more relevant to one of query words.

4. **Modeling layer**: two layers of **bi-directional** LSTMs model the interactions within context words. Note, on the other hand, the attention layer models interactions between query and context.

5. **Output layer**: two classifiers predict the start and end positions.

In 2017, BiDAF was the state-of-the-art system reaching an EM of 77.9 and an F1 of 85.3 on SQuAD 1.

### 14.2.2 BERT

The architecture is much simpler than BiDAF. Indeed, it has a BERT layer that returns hidden vectors of the paragraph. Then, a softmax layer returns

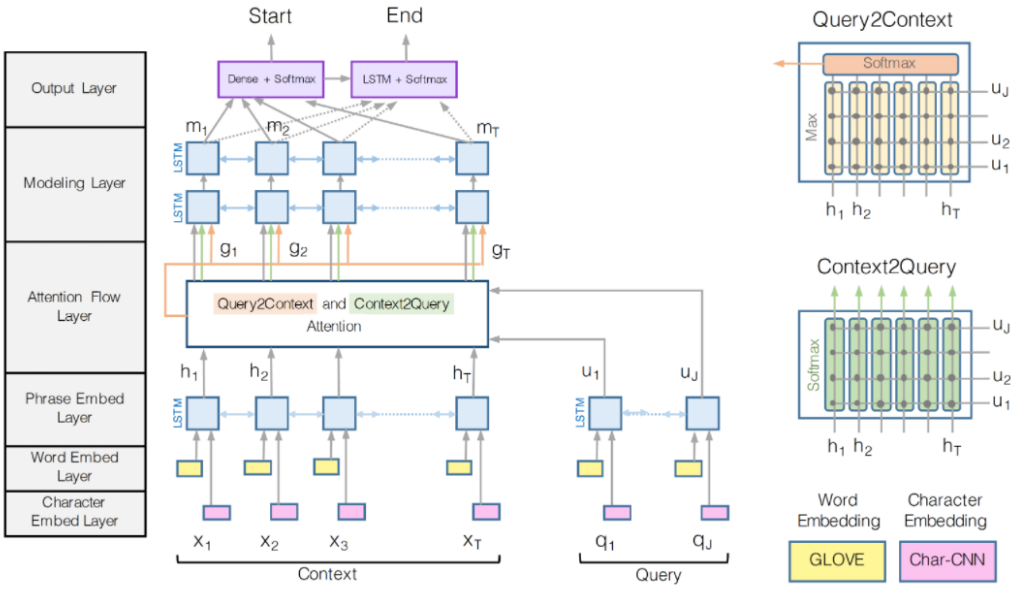



Figure 14.1: BiDAF architecture

the probability distribution of the starting position of the answer. Another softmax layer delivers the probability distribution for the end position. In formulas:

$$p_{start} = \text{softmax}_i(W_{start}^T H) \tag{14.1}$$

$$p_{end} = \text{softmax}_i(W_{end}^T H) \tag{14.2}$$

Where $H = [h_1, h_2, \ldots, h_N]$ are the hidden vectors of the paragraph, returned by BERT.

All the BERT parameters (i.e., 110 milions) as well as the newly introduced parameters are optimized together for the following loss:

$$L = -\log p_{start}(s^*) - \log p_e nd(e^*) \tag{14.3}$$

The system works amazingly well, reaching an EM of 89.3 and an F1 of 94.8 on SQuAD 1.

### 14.2.3 Differences between BiDAF and BERT

BERT model has many more parameters (110M or 330M) than BiDAF has ($\approx$ 2.5M parameters). BiDAF is built on top of several bidirectional LSTMs

while BERT is built on top of Transformers (no recurrence architecture and easier to parallelize).

BERT is pre-trained trained while BiDAF is only built on top of GloVe (and all the remaining parameters need to be learned from the supervision datasets).

Are they really fundamentally different? Probably not:

- BiDAF and other models aim to model the interactions between questions and passage.
- BERT uses self-attention between the **concatenation** of question and passage.

Clark and Gardnwer showed that adding a self-attention layer for the passage attention(P,P) to BiDAF also improves performances.

Chapter 15

# Open Domain Question Answering

In Question Answering (QA) tasks, the model must answer questions. We can divide the QA tasks into four categories:

- **Open Domain Question Answering**: Relies on external memory consisting of large corpora of documents, that may be pre-proceesed to build IR systems (inverted index, ranking).
- **Answer Selection**: Concentrates on choosing on the fly most likely passage containing the answer from a small given list.
- **Dialog System**: Must provide answers to questions and also keep memory of dialog context, remembering previous statements.
- **Reading Comprehension**: Find the answer within a given paragraph (no pre-processing, no memory)

In this chapter, we will discuss open domain question answering.

Vanilla neural networks only store memory in their parameters. Therefore, we can not extend them through usage. In open domain QA we must combine both access to external memory (**Document Retriever**), and the reading comprehension (**Document Reader**):

1. A document retriever selects several candidate documents that may contain the answer from a collection, such as Wikipedia.
2. A document reader extracts the answer from the candidates.

For instance, in DrQA (Doctor QA), the retriever is a standard TF-IDF information-retrieval sparse model. The reader is a neural reading comprehension model trained on SQuAD and other distantly-supervised QA datasets.

## 15.1 Alternative Approaches

Each text passage can be encoded as a vector using BERT and the retriever **score** can be measured as the **dot product** between the question representation and passage representation (cosine similarity). However, it is not easy to model as there are huge number of passages.

Another approach consists in also use the language model to generate answers instead of extracting answers.

Finally, it is possible to encode all the phrases (60 billion phrases in Wikipedia) using **dense** vectors and only do **nearest neighbor search without a BERT model** at inference time.

# Bibliography


[1] Sanjeev Arora et al. "Linear algebraic structure of word senses, with applications to polysemy". In: *Transactions of the Association for Computational Linguistics* 6 (2018), pp. 483–495.

[2] Ronan Collobert et al. "Natural language processing (almost) from scratch". In: *Journal of machine learning research* 12.ARTICLE (2011), pp. 2493–2537.

[3] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[4] Mark D. Kemighan, Kenneth W. Church, and William A. Gale. "A Spelling Correction Program Based on a Noisy Channel Model". In: *COLING 1990 Volume 2: Papers presented to the 13th International Conference on Computational Linguistics*. 1990. URL: https://www.aclweb.org/anthology/C90-2036.

[5] Dan Kondratyuk and Milan Straka. "75 languages, 1 model: Parsing universal dependencies universally". In: *arXiv preprint arXiv:1904.02099* (2019).

[6] Omer Levy and Yoav Goldberg. "Neural Word Embedding as Implicit Matrix Factorization". In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani et al. Vol. 27. Curran Associates, Inc., 2014. URL: https://proceedings.neurips.cc/paper/2014/file/feab05aa91085b7a8012516bc3533958-Paper.pdf.

[7] Omer Levy, Yoav Goldberg, and Ido Dagan. "Improving distributional similarity with lessons learned from word embeddings". In: *Transactions of the Association for Computational Linguistics* 3 (2015), pp. 211–225.

[8] Joakim Nivre et al. "Universal dependencies v1: A multilingual treebank collection". In: *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*. 2016, pp. 1659–1666.

[9] Jeffrey Pennington, Richard Socher, and Christopher D Manning. "Glove: Global vectors for word representation". In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.

[10] Matthew E Peters et al. "Deep contextualized word representations". In: *arXiv preprint arXiv:1802.05365* (2018).

[11] Slav Petrov, Dipanjan Das, and Ryan McDonald. "A universal part-of-speech tagset". In: *arXiv preprint arXiv:1104.2086* (2011).

[12] David MW Powers. "Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation". In: *arXiv preprint arXiv:2010.16061* (2020).

[13] Alec Radford, Rafal Jozefowicz, and Ilya Sutskever. *Learning to Generate Reviews and Discovering Sentiment*. 2017. arXiv: 1704.01444 [cs.LG].

[14] Minjoon Seo et al. "Bidirectional attention flow for machine comprehension". In: *arXiv preprint arXiv:1611.01603* (2016).

[15] Yirong Shen and Jing Jiang. "Improving the performance of Naive Bayes for text classification". In: *CS224N Spring* (2003).

[16] Duyu Tang et al. "Learning sentiment-specific word embedding for twitter sentiment classification". In: *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2014, pp. 1555–1565.