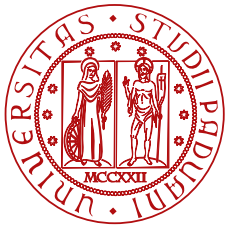


Università
degli Studi
di Padova

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Master degree in Computer Engineering

Natural Language Processing

University of Padova

Academic Year
2023/2024

Contents

1 Natural Language Processing **1**

1.1 Essentials of Linguistic 1

2 Text Normalization **5**

2.1 Tokenization 7

2.1.1 Word tokenization 8

2.1.2 Character tokenization 8

2.1.3 Subword tokenization 8

3 Words and meaning **13**

3.1 Distributional semantics 14

3.1.1 Information theory 17

3.1.2 Neural Networks Word Embeddings 19

3.1.3 Issues 23

3.1.4 Miscellanea 23

3.1.5 Evaluation 24

3.1.6 Cross-lingual word embeddings 25

4 Language modelling **29**

4.0.1 N-gram model 31

4.0.2 Evaluation 32

4.1 Sparse Data 33

4.1.1 Zero numerator: smoothing 33

4.1.2 Zero denominator: backoff and interpolation 36

4.1.3 Estimation of unknown words 37

4.2 Neural language models 38

4.2.1 Feedforward Neural Networks 39

4.2.2 Recurrent NLM 42

4.2.3 Practical issues 43

5 Large Language Modelling **45**
5.1 Review of Transformer architecture 45
5.1.1 Encoder 46
5.1.2 Decoder 52
5.2 Contextualized Embeddings 55
5.2.1 ELMo 56
5.2.2 BERT 57
5.3 Large Language Models 60
5.3.1 GPT family 60
5.3.2 Multilingual large language models 61
5.3.3 SBERT 62
5.3.4 Miscellanea 63
5.4 Fine-Tuning 63
5.4.1 Transfer learning 65
5.4.2 RAG 67

6 Part of Speech Tagging **69**
6.1 Hidden Markov model 71
6.1.1 Probability estimation 72
6.1.2 HMMs as automata 73
6.1.3 Viterbi algorithm: supervised learning 74
6.2 Forward-Backward algorithm: unsupervised learning 76
6.2.1 Forward algorithm 77
6.2.2 Backward algorithm 79
6.2.3 Baum-Welch algorithm 80
6.3 Conditional random fields 82
6.3.1 Inference 85
6.3.2 Training 85
6.4 Neural POS tagger 86
6.4.1 Sequence labelling 89

7 Dependency Parsing **91**
7.1 Syntax and Phrase-Structure 91
7.2 Dependency Grammar 94
7.3 Training a parser 96

7.3.1 Transition-Based Dependency Parsing 97
7.3.2 Alternative models for dependency parsing 102
7.4 Neural Dependency Parsing 103
7.5 Evaluation 105

8 Semantics parsing **107**
8.1 Semantic Role Labelling 110
8.2 Meaning 111

9 Machine Translation **115**
9.1 Statistical machine translation 116
9.2 Neural Machine Translation 117
9.2.1 Encoder-decoder networks using RNN 118
9.2.2 Attention 121
9.3 Encoder-decoder with Transformer 123
9.3.1 Beam search 124
9.3.2 Training on parallel corpora and evaluation 126

10 Question Answering **129**
10.1 Text-based QA 130
10.1.1 Using contextual embeddings 131
10.1.2 Stanford attentive reader 133
10.1.3 Retrieval-augmented generation 134
10.1.4 Miscellanea 135
10.2 Knowledge-based QA 135

1

Natural Language Processing

Natural Language Processing is a field of AI that allows machines to read, derive meaning from text, and produce documents automatically. It's tricky because despite text data is fundamentally discrete, new words can always be created.

Moreover, language is **ambiguous**, **compositional** (meaning of a unit depends also on other units of the sentence), **recursive** (can be repeatedly combined).

1.1 Essentials of Linguistic

Natural language is a structured system for communication, consisting of a vocabulary and a grammar.

Linguistics is the scientific study of language, and in particular the relationship between language form and language meaning.

There is a new framework for the study of language, called **generative linguistics**, which has kind of structures that seem to be **universal** across languages.

In Figure 1.1, different parts of the linguistics field of study.

Phonology studies rules that organize patterns of sounds in human languages and it is different from **phonetics** which is concerned with production, transmission and perception of sounds.

Morphology is the study of how words are composed by **morphemes** (our "sillabe"), which are our smallest units of language. A word is composed of root

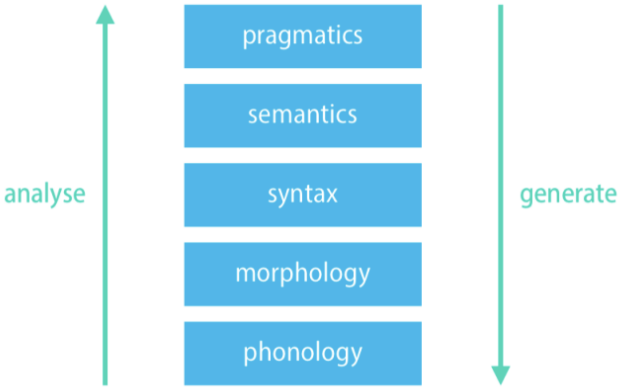


Figure 1.1: Linguistic diagram blocks

plus affixes.

It's important to distinguish between **inflectional morphology** and **derivational morphology**: the first never change the grammatical category (part of speech) of a word. If the word is a verb, it remains a verb. Instead, the second, changes in the grammatical category.

The term **morphologically rich language** refers to a language in which substantial grammatical information is expressed at word level.

Syntax studies rules and constraints that govern how words can be organized into sentences. NLP systems has to be robust to input that doesn't follow the rules of grammar.

A **part of speech** is a category for words that play similar roles within the syntactic structure of a sentence. It can be defined:

- **distributionally**: Kim saw the $elephant$, $movie$, $mountain$ before we did.
- **functionally**: verbs= predicates, nouns= arguments, adverbs = modify verbs...

We can subdivide the part of speech using **open class tags** (or content words) that are nouns, verbs, adjectives. New words of a language are usually added to these classes. Instead, **closed tags** (or function words) which are prepositions, conjunctions etc and rarely receive new members.

We can represent syntactic structure in several ways, but the most common are:

- **phrase structure**: is a tree like representation where leafs represent words and internal nodes represent groups called **sentence**. These structures use labels as shown in Figure 1.2

PoS tags	Phrase tags
—	S = Sentence
N = Noun	NP = Noun Phrase
V = Verb	VP = Verb Phrase
P = Preposition	PP = Prepositional Phrase
A = Adjective	AP = Adjectival Phrase
Det = Determiner	—
⋮	⋮

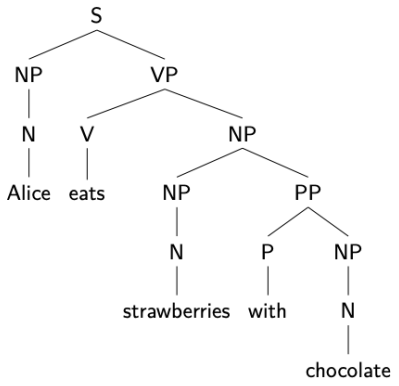


Figure 1.2: Phrase structure

- **dependency tree**: is a tree-like representation where nodes represent words and puntuaction in the sentence and arcs represent grammatical relations between a **head** and a **dependent**. Dependency trees use labels at arcs, representing **grammatical relations** and arcs are often called **dependencies**. In Figure 1.3 an example

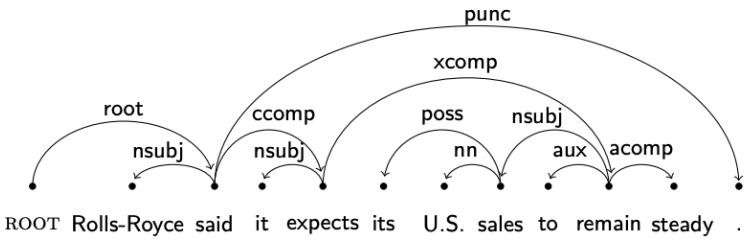


Figure 1.3: Dependency tree

Semantics is the study of **meaning** of linguistic expressions such as words, phrases, and sentences in an abstract sense, without reference to a precise context.

The study of word meaning is **lexical semantics**. In this study, we can distinguish internal and external semantic structure: internal refers to the similarity

with other words, and external refers to the allowability to combine with other words. In Figure 1.4 there is an example

Example : Contrast the following sentences
Alice eats an apple
?Alice eats a thunderstorm (internal violation for thunderstorm)
*Alice eats an apple to John (external violation for eat)
? marks weak ungrammaticality; * marks strong ungrammaticality.

Figure 1.4: Example

Lexical ambiguity arises because a word can have different meanings, called **word senses**. The meaning of a whole expression is a function of the meanings of its parts and of the way they are syntactically combined.
Pragmatics studies the way linguistic expressions with their semantic meanings are used for specific **communicative** goals or rather what an expression means in a given context.
Discourse analysis studies written and spoken language in relation to its social context where a discourse is a piece of text with multiple sub-topics and coherence relations between them.

2

Text Normalization

When we analyze a text, it is important to distinguish **word** and **punctuation marks**.

Punctuation is critical for finding sentence boundaries (period, colon, etc.) and for identifying some aspects of meaning (question mark, etc.).

Moreover, it is important to distinguish **tokens** and **types**. A token is the individual occurrences of a word in a document while a type is a distinct word in a document.

Considering the example sentence *"They picnicked by the pool, then lay back on the grass and looked at the stars"*, ignoring punctuation marks, the above sentence has 16 tokens and 14 types.

The set of all types in a corpus is the **vocabulary** V while the **vocabulary size** $|V|$ is the number of types in the corpus. The size of the corpus N is the number of tokens without considering punctuation marks. In Figure 2.1 an example of the differences between these two metrics consdierinng different English corpora.

Example : N vs. $|V|$ for a few English corpora

Corpus	Tokens = N	Types = $\|V\|$
Shakespeare	884 thousand	31 thousand
Brown corpus	1 million	38 thousand
Switchboard telephone conversations	2.4 million	20 thousand
COCA	440 million	2 million
Google n-grams	1 trillion	13 million

Figure 2.1: Example

In very large corpora, the relation between N and $|V|$ can be expressed as

$$|V| = kN^{\beta}$$

where k and β has precise value. This was the **Herdan law**.

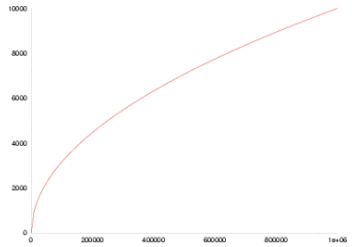


Figure 2.2: Herdan's law

Instead, **Zipf/Mandelbrot law** is referred to the frequency of the $r-th$ most used type, that is:

$$f(r) = \frac{1}{(r+\beta)^{\alpha}}$$

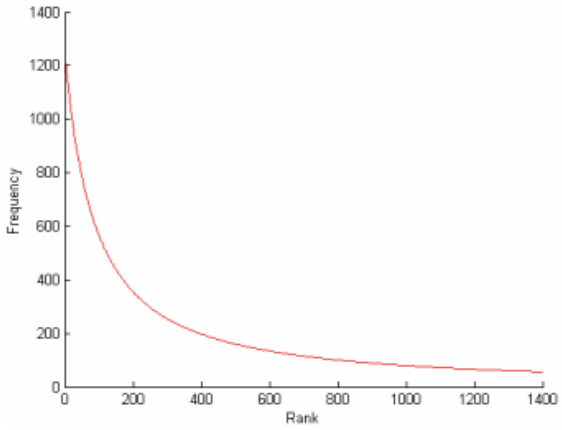


Figure 2.3: Zipf law

Before starting with the text normalization, it's important to deal with the concept of word form: **word form** is each derived form of a word. Each word form is associated to a single lemma, the citation form used in dictionaries. For example, verbs declination of a lemma (e.g **sing, sang, sung**) are word forms,

associated to a lemma (**sing**).
There is no one-to-one correspondence between tokens and word-forms.

Now we try to enter a little bit mroe in the specific in the task of **text normalization**. A **corpus**, (plural: corpora), is a large collection of text, in computer-readable form.

When building corpora for NLP tasks, it's crucial to consider different factors like language and genre (different sources of text documents) variation. Testing NLP algorithms across multiple languages is essential.

Languages with limited resources (large corpora) pose specific challenges for research and development in the field of NLP, and are called **low-resource languages**.

Text normalization is the process of transforming the text into some predefined standard form. This consists of several tasks, and there is no standard normalization procedure since it depends on what type of text is being normalized and what type of NLP task needs to be carried out.

There are several tasks involved in test normalization:

- **Language identification** is the task of detecting the source language for the input text. We need to know what is the language of the text. There are several **statistical** techniques for this task
- **Spell checkers** correct grammatical mistakes in text. It uses approximate string matching algorithms such as **Levenshtein distance** to find correct spellings. There are difficult cases like when we have a misspelled word that can be also in the language (than vs. then).
  To deal with these cases, more sophisticated algorithms analyze the context by surrounding words.
- **Contractions**: Contractions need to be managed before normalization. Contracted forms (I'm), abbreviations (Mr.) or slang (LOL) can be managed by creating a dictionary with contractions and expansions.
- **Puntuaction** marks need to be isolated and treated as if they were separate words. This is important for finding sentence boundaries and for identifying some aspects of meaning
- **Special characters** like emoticons or emoji

2.1 Tokenization

Tokenization is the process of segmenting text into units called **tokens**. Tokenization techniques can be grouped into three families that we will see.

It's important to say that all tokens then are organized into a vocabulary and they may be mapped into natural numbers, making a **token indexing**.

2.1.1 Word tokenization

Word tokenization is a very common approach for European languages. For English, most of the test is already tokenized after previous steps of normalization with exceptions for compound names (e.g *white space vs whitespace*) or city names, companies etc. For other languages this task can be more difficult (as for German or Italian).

Moreover, there are certain language-independent tokens that require specialized processing such as phone numbers, dates, email addresses, etc. The use of regular expressions is recommended in these cases.

2.1.2 Character tokenization

Major east Asian languages write text without any spaces bewteen words. For most Chinese NLP tasks, characer tokenization works better than word tokenization since each character represents a single unit of meaning.

2.1.3 Subword tokenization

Many NLP systems need to deal with **unknown words** that are words that are not in the vocabulary of the system.

For example if the corpus contains "foot" and "ball" but not football, if football appears in the test set, it would not be recognized.

To handle this problem of unknown words, modern **tokenizers** automatically induces sets of tokens that include tokens smaller than words, called **subwords**.

Subword tokenization reduces vocabulary size, that is good, and this is why it's the most common tokenization method for large language modelling.

Why **subword tokenization reduces the vocabulary number?**

With subword tokenization, we can break down any word into smaller, more frequent components (subwords). This means that we can handle rare or unknown words by splitting them into known subwords, rather than needing a unique token for every possible word.
In traditional word tokenization, every single word must be in the vocabulary.

This leads to a very large vocabulary because you need to account for all variations of words (e.g., plurals, conjugations, compound words). With subword tokenization, these variations can be handled by combining smaller units that are already in the vocabulary.

Common substrings (prefixes, suffixes, roots) are reused across different words. For instance, the words "playing", "played", "player" can all be tokenized using the common subword "play" and respective suffixes "-ing", "-ed", and "-er". This reuse of subwords means fewer unique tokens are needed overall.

Subword tokenization schemes consist of three different algorithms:

- **token learner** takes an unprocessed training corpus and induces a set of tokens, called **vocabulary**
- **token segmenter** (encoder) which takes a vocabulary and an unprocessed test sentence and segments the sentence into the tokens of the vocabulary
- **token merger** (decoder) takes a token sequence and reconstructs the original sentence

In Figure 2.4 there is an example.

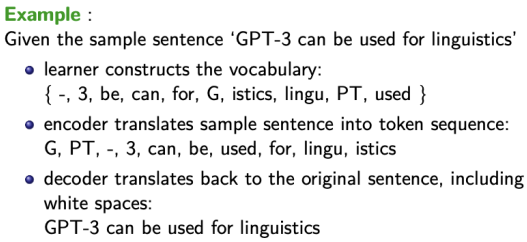


Figure 2.4: Example of subword tokenization

Three algorithms are widely used for subword tokenization.

BPE TOKENIZATION

The **BPE learner** (Byte-pair encoding) is run inside words and uses a special end-of-word marker. It focuses on finding frequent pairs of characters or sequences within single words to create tokens.
The algorithm iterates the following steps:

- begin with a vocabulary composed by all individual characters
- choose the two symbols A, B that are most frequently adjacent
- add a new merged symbol AB to the vocabulary

- replace every adjacent A, B in the corpus with AB

Stop when the vocabulary reaches size k, a hyperparameter.
In the slide there is an important example: after several iterations, BPE learns **entire words** and most frequent **units**, useful for tokenizing unknown words.
Instead there are two version of **BPE token segmenter**:

- apply merge rules in frequency order all over the data set
- for each word, left-to-right, match longest token from vocabulary (eager)

Example :
Assume training corpus contained words **newer**, **low**, but not **lower**. Typically, the test word `[lower]` will be encoded by means of tokens `[low, er_]`.

Figure 2.5: Example

Encoding is computationally expensive. Many systems use some form of **caching** which consists of:

- pre-tokenize all the words and save how a word should be tokenized in a dictionary
- when an unknown word (not in dictionary) is seen apply the encoder to tokenize the word add the tokenization to the dictionary for future reference

At the end, **BPE token merger**. To decode, we have to:

- concatenate all the tokens together to get the whole word
- use the end-of-word marker to solve possible ambiguities

Example :
The encoded sequence

```
[the_, high, est_, range_, in_, Seattle_]
```

will be decoded as

```
[the, highest, range, in, Seattle]
```

as opposed to

```
[the, high, estrange, in, Seattle]
```

Figure 2.6: Example of BPE token merger

WordPiece tokenization

WordPiece is a subword tokenization algorithm used by the large language model BERT. It starts from the initial alphabet and learns merge rules. The main difference is the way A,B is selected to be merged: considering $f(x)$ the frequency of a token x

$$\frac{f(A,B)}{f(A)\cdot f(B)}$$

The algorithm prioritizes the merging of pairs where the individual parts are less frequent in the vocabulary.

After the tokenization of our corpora, there are other important pre-process technique to apply:

- **Sentence segmentation**: breaking up a text into individual sentences and this can be done using cues like periods, question marks, or exclamation points.

- **Lower casing**: is very useful to standardize words but could be difficult in some cases as proper name or cities name. For these a specialized pre-processing is needed.

  Sometimes, it might be useful to keep both versions of the text data.

- **Stop word removal**: stop words are common words that usually carry little meaningful information for understanding the content of a text, especially when it comes to analyzing its themes or topics. These include words like "the", "is", "at", "which", and "on". Because they're so common, they can be frequent to the point of cluttering the analysis, making it harder to find and focus on the important words.
  The process of stop words removal involves identifying and removing these words from the text data before further processing. This can help in improving the performance of various NLP tasks by reducing the dataset's size and focusing the analysis on words that carry more meaning. Stop word removal heavily depends on the task at hand, since it can wipe out relevant information. This is more of a dimensionality reduction technique than normalization.

- **Stemming**: refers to the process of cutting a word with the intention of removing the affixes; it could be problematic since it could sometimes produce words that are not in the language or which could have different meanings.

- **Lemmatization**: has the objective of reducing a word to its base form, also called **lemma**, therefore grouping together different forms of the same word. For example the words *car, cars, cars'* will be associated to the same lemma **car**.

The suggestion is that there is not a perfect way to do pre-processing: a very big slice of NLP is given my trial: try to train model using different approaches and by using model selection evaluate which is the best one!

3

Words and meaning

The linguistic study of word meaning is called **lexical semantics** and a model of word meaning should allow us to relate different inference (significati) to each word, based on the context.
A **word-form** can have multiple meanings; each meaning is called a **word sense**, or sometimes a synset.
Word sense disambiguation (WSD) is the task of determining which sense of a word is being used in a particular context.
Lexical semantic relationship between words are important components of word meaning. Two words can be related as:

- synonyms (common word sense)
- similar (similar meanings, e.g. car and bicycle)
- related, if they refer to related concept (e.g. car and gasoline)
- antonyms (e.g. an opposite meanings, hot and cold)
- One word hyponym of another if the first has a more specific sense (e.g. car and vehicle)
- affective meanings (e.g. happy or sad)

3.1 Distributional semantics

It is very difficult to define the notion of word sense in a way that can be understood by computers.
Suppose you don't know the meaning of the word 'tezguino', but you have seen it in the following contexts.

a) A bottle of _____ is on the table.
b) Everybody likes _____.
c) Don't have _____ before you drive.
d) We make _____ out of corn.

What other words fit into these contexts?

	a)	b)	c)	d)
tezgüino	1	1	1	1
motor oil	1	0	0	1
tortillas	0	1	0	1
choices	0	1	0	0
wine	1	1	1	0

Based on these vectors, we conclude that 'wine' is very similar to 'tezguino'.
Distributional semantics develops methods for quantifying semantic similarities between words based on their distributional properties, meaning their neighboring words.
With distributional properties we refer to how words are distributed respect to another.
The basic idea lays in the so-called distributional hypothesis: **linguistic items with similar distributions have similar meanings.** The basic approach is to collect distributional information in high-dimensional vectors, and to define distributional/semantic similarity in terms of vector similarity.
Similar words are mapped into "close enough" vectors. In Figure an example of vectors from sentiment analysis application.

The obtained vectors are called **word embeddings**. Each discrete word is embedded in a continuous vector space. The approach is called **vector semantics**

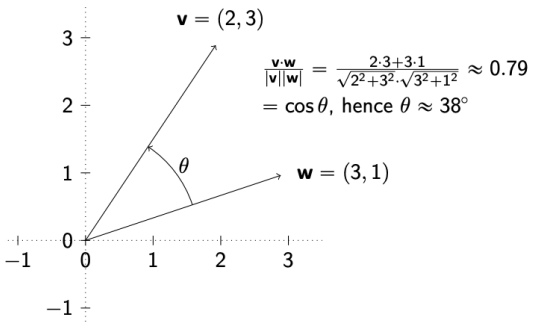
and is the standard way to represent word meaning in NLP. All the learning algorithms we report are unsupervised, so the training set is composed only by words and not by words plus the correct embedding.
We discuss two families of word embeddings:

- **Sparse vectors**: Vector components are computed through some function of the counts of nearby words.
- **Dense vectors**: Vector components are computed through some optimisation or approximation process.

Before going on, it's important to take some rows in order to remember vectors.
Assuming that we succeed in the goal of embedding a word in a vector. How can we indicate the similarity between two vectors?
Supposing two vectors, the **dot product** is provided by:

$$\vec{v} \cdot \vec{w} = |\vec{v}||\vec{w}| \cos\theta$$

where θ is the angle between these two vectors.
The **cosine similarity** between two vectors, $\cos\theta$ (obtained by inverting the formula), is a measure of how close are these vectors. If the value is 0, they are **orthogonal**, -1 are in the **opposite directions** and if it's 0, they are overlapped.



So two word embedding very close will have very similar meanings.
Let $\{w_1, ..., w_W\}$ be a set of **term** (target) words, or rather words that we want to know meanings, and let $\{c_1, ..., c_C\}$ be a set of **context words**.
For each term w_i in the corpus, we consider context words appearing in a window of size L, at the left or at the right of w_i. So, if we consider a target word w, context (words) are words which are in the window L near w.
A **term-context matrix** F is a matrix of size WxC, where each element $f_{i,j}$ is the

number of times that a context word c_j appears in the context of word w_i in the corpus.

In Figure an example

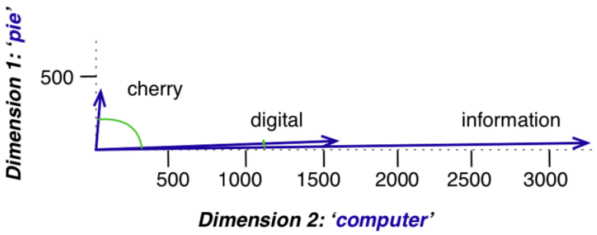
	pie	data	computer
cherry	442	8	2
digital	5	1683	1670
information	5	3982	3325

Cosine similarity between vectors associated with term words:

$$\cos(\text{cherry}, \text{information}) = \frac{442*5+8*3982+2*3325}{\sqrt{442^2+8^2+2^2}\sqrt{5^2+3982^2+3325^2}} = .018$$

$$\cos(\text{digital}, \text{information}) = \frac{5*5+1683*3982+1670*3325}{\sqrt{5^2+1683^2+1670^2}\sqrt{5^2+3982^2+3325^2}} = .996$$

For example, in the context of word "cherry", word "pie" appears 442 times. Restricting to dimensions 'pie' and 'computer' for convenience, so that can be easily represented, the model states, as shown in Figure, that 'information' is way closer to 'digital' that to 'cherry'.



3.1.1 Information theory

Pointwise mutual information (PMI) is a statistical measure used to determine the degree of association between two events, that, in our case, is a context word near a target word.
PMI measures how much more likely two events co-occur compared on what we would expect if they were independent. In other words, it quantifies the extent to which the occurrence of one event is dependent on the occurrence of another:

$$I(x,y) = \log_2 \frac{P(x,y)}{P(x)P(y)}$$

At the denominator there is the case on where two words are independent, so means that the number of times which they compare in the corpora are counted independent.
It's symmetric, can be positive and negative, and if RVs are independent, mutual

information is 0.

- If $PMI = 0$ it suggests that the events A and B are independent
- If $PMI > 0$ it indicates that the events co-occur more often than expected by chance, implying a positive association. So, talking about words, means that there is a relation between these two words.
- If $PMI < 0$ it implies that the events co-occur less frequently than expected by chance, suggesting a negative association.

So, the PMI of w_i and c_j is:

$$PMI(w_i, c_j) = \log_2 \frac{P(w_i, c_j)}{P(w_i)P(c_j)}$$

The numerator tells us how often we observed the two words together.
The denominator tells us how often we would expect the two words to co-occur assuming they each occurred independently. The ratio gives us an estimate of how much more the two words co-occur than we expect by chance.
We can also rewrite it using definition of conditional probability:

$$PMI(w_i, c_j) = \log_2 \frac{P(w_i|c_j)}{P(w_i)}$$

PMI may then be understood as quantifying how much our confidence in the outcome w_i increases after we observe c_j.
It's easy to compute probabilities: remembering that $f_{i,j}$ is the number of times that a context word c_j appears in the context of word w_i in the corpus, we can estimate probabilities:

$$P(w_i, c_j) = \frac{f_{i,j}}{\sum_{i=1}^{W} \sum_{i=1}^{C} f_{i,j}}$$

where the denominator is the total number of possible target/context pairs

$$P(w_i) = \frac{\sum_{j=1}^{C} f_{i,j}}{\sum_{i=1}^{W} \sum_{i=1}^{C} f_{i,j}}$$

so the numerator means all times there is the word w_i in the corpora

$$P(c_j) = \frac{\sum_{i=1}^{W} f_{i,j}}{\sum_{i=1}^{W} \sum_{i=1}^{C} f_{i,j}}$$

so the numerator means all times there is the word c_j in the corpora
However, it's difficult to estimate negative values of PMI (when we have very low probabilities): in this case it's useful to use positive PMI (PPMI).

$$PPMI(w_t, w_c) = max\{\log_2 \frac{P(w_t, w_c)}{P(w_t)P(w_c)}, 0\}$$

In the notebook there is an example.
Practical issues: PMI has the problem of being **biased** toward infrequent events: very rare words tend to have very high PMI values. One way to reduce this bias is to slightly change the computation for $P(w_c)$ in the PPMI:

$$PPMI_\alpha(w_t, w_c) = max\{\log_2 \frac{P(w_t, w_c)}{P(w_t)P_\alpha(w_c)}, 0\}$$

with

$$P_\alpha(w_c) = \frac{f(w_c)^\alpha}{\sum_{v \in V} f(v)^\alpha}$$

We can use the rows of the PPMI term-context matrix as word embeddings. Notice that these vectors:

- have the size of the vocabulary, which can be quite large
- when viewed as arrays, they are very sparse

Dot product between these word embeddings needs to use specialised software libraries, implementing vectors as dictionaries rather than arrays. We can obtain more **dense** word embeddings from the PPMI term-context matrix and it's done through a matrix approximation technique known as **truncated singular value decomposition**. In the notebook it's shown how to do that

3.1.2 Neural Networks Word Embeddings

Word embeddings can be done also by using **neural networks** and it works better than previous embeddings in every NLP task, since they're dense, low dimensions and component can be negative.
The first NN embedding that we will see is **static**: it means that the presented methods learn one fixed embedding for each word in the vocabulary (we will see also contextualized, which depends on the context).
Word2vec is a software package including two different algorithms for learning word embeddings:

- skip-gram with negative sampling (SGNS)
- continuous bag-of-words (CBOW)

We focus on the skip-gram algorithm.
Our idea is that instead of counting how often a context word appears near a target word, we train a classifier on the following binary prediction task: is a given context word probably to appear near a given target word? We don't care the answer but the parameters provided, that we'll use as word embeddings.
The input layer represents the target word, and the projection layer represents the word embeddings or vector representations.
Basic steps of skip-gram algorithm are the following. For each target word $w_t \in V$:

- treat w_t and any neighboring context word w_c as **positive examples**. The context is defined by a window size, which determines the number of words before and after the target word that are considered context words. Consider an example sentence: "I love to eat pizza."
  If we set the window size to 2, the context-target pairs for the word "love" would be:
  Context: [I, to, eat]
  Target: love
  Similarly, we create context-target pairs for all the words in the training data.
- randomly sample other words $w_n \in V$, called **noise words**, to produce **negative examples** for w_t

By following these steps, you construct a training set consisting of both positive examples (target word and context word pairs) and negative examples (target word and noise word pairs). This training set is then used to train the skip-gram model, which learns word embeddings that capture semantic relationships between words. An example is shown in Figure

positive examples +

w	c_{pos}
apricot	tablespoon
apricot	of
apricot	jam
apricot	a

negative examples -

w	c_{neg}	w	c_{neg}
apricot	aardvark	apricot	seven
apricot	my	apricot	forever
apricot	where	apricot	dear
apricot	coaxial	apricot	if

Then we use **logistic regression** to train a classifier that provides us the probability that an input word is positive or negative, so it aims to predict, given as input a target word and a generic word, of how much is probably that it is a context word or a noise word, using the learned weight (that we'll use as embeddings).

In order to apply logistic regression, for any pair of words $w_t, u \in V$ our training set, we need to define the probability that u is or is not a context word for target w_t

$$P(+|w_t, u)$$

$$P(-|w_t, u) = 1 - P(+|w_t, u)$$

For each $w \in V$, we construct two **complementary embeddings**:

- a target embedding $\vec{e_t}(w) \in \mathbb{R}^d$
- a context embedding $\vec{e_c}(w) \in \mathbb{R}^d$

where d is the size of embedding. The first one is always assigned to target words, the second one is always assigned to context word (so also noise words). We compute the dot product between these two vectors:

$$\vec{e_t}(w) \cdot \vec{e_c}(u) = |\vec{e_t}(w)||\vec{e_c}(w)| \cos\theta$$

where θ is the angle between those vectors. Then we use sigmoid function σ to assign probabilities.

$$P(+|w_t, u) = \sigma(\vec{e_t}(w) \cdot \vec{e_c}(u)) = \frac{1}{1 + e^{-\vec{e_t}(w) \cdot \vec{e_c}(u)}}$$

So if the two vectors have a small angle, the probability that these vectors are positive sample is close to one. Remember that the probability refers to the fact that these two words are related, so to be a positive sample. Instead, if they have a large angle, probability is close to zero.
Obviously, as flip of the coin, in this context, the probability of being a negative example is the opposite

$$P(-|w_t, u) = \sigma(\vec{e_t}(w) \cdot \vec{e_c}(u)) = \frac{1}{1 + e^{\vec{e_t}(w) \cdot \vec{e_c}(u)}}$$

so when $P(+|w_t, u)$ is close to 0, $P(-|w_t, u)$ is close to 1.
Now we need to talk about the **training** of this model: for simplicity, assume a dataset with only a target/context pair (w, u), along with k noise words $v_1, v_2, ...v_k$ negative examples.
Skip-gram assume that all context words are **independent**, so we can compute the log-likelihood of the data.

After some steps of computation, the result is:

$$LL_w = \log(P(+|w,u)) + \sum_{i=1}^{k} log(P(-|w,v_k))$$

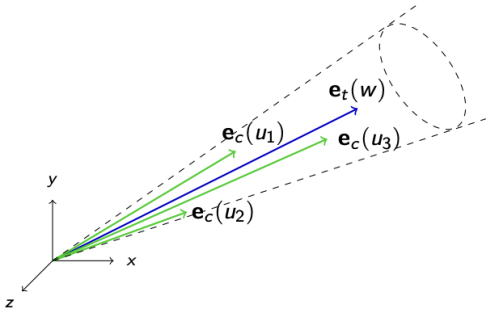
and we search for word embeddings which maximize this function, so maximize the this probability.
We can equivalently minimize the inverse of this function, that is the **cross entropy** loss function:

$$L_{CE} = -\log(P(+|w,u)) - \sum_{i=1}^{k} log(P(-|w,v_k))$$

So the model parameters are two matrices $|V|xd$ with the target and context embeddings $\vec{e_t}(w)$, $\vec{e_c}(w)$ randomly initialised.
By making an update to minimise the function, we:

- force an increase in similarity between $\vec{e_t}(w)$, $\vec{e_c}(w)$
- force a decrease in similarity between $\vec{e_t}(w)$, $\vec{e_c}(v_k)$ for all the noise words

We train the model with **stochastic gradient descent**, as usual for logistic regression. At the end, retain only $\vec{e_t}(w)$, ignoring $\vec{e_c}(w)$.
We can give a geometric interpretation: the target vector is surrounded by its context vectors, as shown in Figure.



Let w a target word with context words u_j. Let also w' be a target word similar to w. Then we have that $\vec{e_c}(u_j)$ form a thin cone surrounding vector $\vec{e_t}(w)$, and vector $\vec{e_t}(w')$ mus be placed within that cone, since w and w' share their context words. At the end, $\vec{e_t}(w)$ and $\vec{e_t}(w')$ are forced to be at a small angle. For the same reason, different words repel each other.

3.1.3 Issues

There are some practical issues to solve: it's needed to **clean up the text**.
We need to:

- convert any **puntuaction** into token $< period >$
- remove all words w such that $f(w) \leq 5$ in order to reduce noise and to improve the quality of the vector representation
- **subsampling**: discard occurrences of words that are showed up very often as "of, the, for". We discard an occurence of w_i with probability

$$P(w_i) = 1 - \sqrt{\frac{t}{z(w_i)}}$$

  where $z(w_i)$ is the normalized frequency of w_i and t is a threshold parameter depending on dataset size.
- **hyperparameter** k, ratio bewteen positive and negative examples, must be adjusted
- **hyperparameter** L, size of the window, has to be fixed

An important question is born: Should we estimate one embedding per word form or else estimate distinct embeddings for each word sense? Intuitively, if word representations are able to capture the meaning of individual words, then words with multiple meanings should have multiple embeddings.

3.1.4 Miscellanea

FastText

FastText is an embedding that deals with unknown words and sparsity in languages with rich morphology, by using **subword models**.
Each word in fastText is represented by itself plus a bag of character $N - grams$, for bounded values of N. A character $N - gram$ is a sequence of N characters. We will introduce N-grams in some later lecture on language models. In Figure there is an example

Example : With $N = 3$ the word 'where' would be represented by the following strings (_ is a boundary marker)

where, _wh, whe, her, ere, re_

It's learned the embedding for each N-gram and the entire word embedding is obtained by summing all embeddings. It's useful with MRLs where words have a regular internal structure.

GloVe

Global Vectors, or GloVe for short, is an embedding that accounts for global corpus statistics, which was somehow disregarded by word2vec. GloVe combines the intuitions of count-based models like PPMI, while also capturing the linear structures used by methods like word2vec.

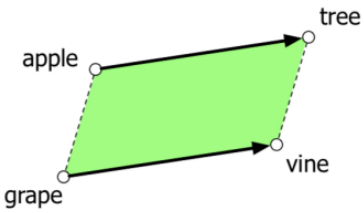
Word embedding can be used to visualize the meaning of a word w. Most commonly used methods are:

- listing words in V with highest cosine similarity
- project the d dimensions of a word embedding down into 2 dimensions

The choice of the L hyperparameters depends on the goals of the representation:

- small values of L provide semantically similar words with the same parts of speech
- larger values of L provide words that are topically related but not similar

Another semantic property of embeddings is their ability to capture relational meanings. The parallelogram model can be used for solving problems such **analogy**, as shown in Figure



Unfortunately, word embeddings also reproduce the implicit biases and stereotypes that are latent in the text.

3.1.5 Evaluation

There are different ways in order to evaluate a word embeddings model.

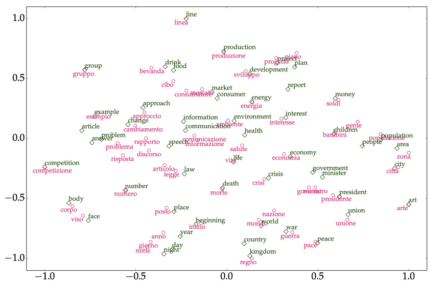
- **Extrinsic evaluation**: use the model to be evaluated in some end-to-end application (machine translation, sentiment analysis) and measure performance

- **Instrinsic evaluation**: look at performance of model in isolation, w.r.t. a given evaluation measure.
  There are three types of instrinsic evaluation:
  - word similarity: compute numerical score for the semantic similarity between two words
  - word relatedness: compute the degree of how much one word is related to another
  - word analogy: comparison of two things to show their similarities

Several dataset are available for intrinsic evaluation task.

3.1.6 Cross-lingual word embeddings

It encodes words from two or more different languages in a shared high-dimensional space. Vectors representing words with similar meaning are closely located, regardless of language, as shown in Figure.



It's very useful in comparing the meaning of words across languages and are key to applications such as machine translation.
The class of **objective function** minimized by most cross-lingual word embedding methods can be formulated as:

$$J = L^1 + L^2 + ... + L^l + \Omega$$

where each L^i is the monolingual loss of the $i - th$ language.
Joint optimization of multiple losses is difficult since it's not guaranteed to reach a even a local optimum: most approaches optimize one loss at a time, while keeping certain variables fixed.
However, in multilingual problems, the choice of multilingual parallel data sources is more important for the model performance than the actual underlying architecture.

Extending the learning process from bilingual to multilingual settings improves results, as reported on many standard tasks.
It's interesting that ambiguity for word w in one language does not transfer to the translation of w in another language: variation in ambiguity acts as a form of naturally occurring supervision.

Resume on Static Word Embedding using Skip-Gram Algorithm

- **Objective:** The goal of the Skip-Gram model is to learn word embeddings by predicting the context words (nearby words) for a given target word in the corpus. The essence is to capture the semantic relationships between words through their vector representations.
- **Corpus:** We start with a large text corpus. The corpus serves as the dataset for training our model, allowing it to learn embeddings that capture the linguistic context of words.
- **Target and Context Words:** For each word in the corpus (considered as the target word), we define its context as the words surrounding it within a specified window size. The model then tries to predict these context words based on the target word.
- **Positive and Negative Samples:** In addition to the actual context words (positive samples), the Skip-Gram model also uses negative sampling. Negative samples (noise words) are randomly selected words from the vocabulary that are not in the target word's context. This approach helps improve training efficiency and quality of the word embeddings.
- **Objective Function:** The model optimizes a function that maximizes the probability of correctly classifying a word as a context word (positive sample) or not (negative sample) given the target word. This is typically achieved using logistic regression. The optimization process adjusts the embeddings to reduce the prediction error.
- **Resulting Embeddings:** Upon completion of training, each word in the corpus is represented by a dense vector (embedding). These embeddings capture semantic similarities among words; words with similar meanings tend to be closer in the vector space.
- **Usage of Embeddings:** While embeddings are learned for both target and context positions, typically only the target word embeddings are used in subsequent tasks. These embeddings can be utilized in various natural language processing (NLP) tasks, such as text classification, sentiment analysis, and more, to enhance model performance by providing a rich representation of word semantics.

4

Language modelling

Language modelling is the task of predicting which word comes next in a sequence of words. More formally, given a sequence of words $w_1 w_2 ... w_t$, we want to know the probability of the next word w_{t+1}:

$$P(w_{t+1}|w_1 w_2 ... w_t)$$

where we can write also $w_{1:t}$
This allows language modelling to be treated as a classification task.
Rather than as **predictive** models, language models can be also be viewed as **generative** models that assign probability to a piece of text

$$P(w_1 \cdots w_t)$$

Unlike predictive models that focus on the next word, generative models are concerned with estimating the probability of entire sequences of words, or pieces of text, being coherent, grammatically correct, or contextually relevant. This means that a generative model looks at a sequence of words (e.g., a sentence) and assigns a probability to it that reflects how likely this sequence is to occur in the language as per the model's training. So it answers to the question: *How likely is it that the given text belongs to the modeled language?* These two views are **equivalent** since this probability can be expressed as a product of conditional

probability

$$P(w_{1:n}) = \prod_{t=1}^{n} P(w_t|w_{1:t-1})$$

and a conditional probability can be expressed as:

$$P(w_{t+1}|w_{1:t}) = \frac{P(w_{1:t+1})}{P(w_{1:t})}$$

For example, in machine translation system, when received a sentence, there will be different possible meanings: we use a language model to select the most likely candidate.

Probably estimation: Assume the text is its water is so transparent that. We want to know the probability that the next word is the.

We denote with $C(\cdot)$ the number of times the string is seen in a large corpus. In order to estimate the above probability, we have:

$$P(the|its\ water\ is\ so\ transparent\ that) = \frac{C(its\ water\ is\ so\ transparent\ that\ the)}{C(its\ water\ is\ so\ transparent\ that)}$$

In order that these probabilities are summable, we add to the vocabulary a sentence end marker.

More generally, given a large corpus, we can set:

$$P(w_t|w_{1:t-1}) = \frac{C(w_{1:t})}{C(w_{1:t-1})}$$

with

$$C(w_{1:t-1}) = \sum_{u \in V} C(w_{1:t-1}u)$$

that means occurrences of the sentence $w_{1:t-1}$ inside the corpora.

Quantities at numerator are **frequencies**, the ratio is called **relative frequency** and the estimator above is called **relative frequency estimator**.

The relative frequency estimator will be correct in the limit, that is, with infinite data.

This estimator is extremely data-hungry, and suffers from high variance: depending on what data happens to be in the corpus, we could get very different probability estimations.

4.0.1 N-GRAM MODEL

A string $w_{t-N+1:t}$ of N words is called **N-gram**. The **N-gram model** approximates the probability of a word given the entire sentence history by *conditioning only on the past N-1 words*.
N represents the number of preceding words considered when predicting the next word in a sequence.
In the context of language modeling, an N-gram is a contiguous sequence of N items (which can be characters, words, or other linguistic units) from a given text or corpus. For instance, in a trigram model (n=3), each n-gram consists of three consecutive words.
The assumption that the probability of a word depends only on a few previous words is called a **Markov assumption**.
Taking as example the 2-gram model, it's valid the approximation:

$$P(w_t|w_{1:t-1}) \approx P(w_t|w_{t-1})$$

so the general equation for N-gram model is:

$$P(w_t|w_{1:t-1}) \approx P(w_t|w_{t-N+1:t-1})$$

and so the relative frequency estimator is:

$$P(w_t|w_{t-N+1:t-1}) = \frac{C(w_{t-N+1:t})}{C(w_{t-N+1:t-1})}$$

The model requires estimating and storing the probability of only $|V|^N$ events, which is exponential in N.
In Figure an example.

Consider a mini-corpus of three sentences. We augment each sentence with start and end markers <s> and </s>

```
<s> I am Sam </s>
<s> Sam I am </s>
<s> I do not like green eggs and ham </s>
```

We get the following 2-gram probabilities, among others:

$P(\texttt{I}\vert\texttt{<s>}) = \frac{2}{3} = .67$	$P(\texttt{Sam}\vert\texttt{<s>}) = \frac{1}{3} = .33$
$P(\texttt{am}\vert\texttt{I}) = \frac{2}{3} = .67$	$P(\texttt{</s>}\vert\texttt{Sam}) = \frac{1}{2} = .50$
$P(\texttt{Sam}\vert\texttt{am}) = \frac{1}{2} = .50$	$P(\texttt{do}\vert\texttt{I}) = \frac{1}{3} = .33$

Talking about **learning**, N is the hyperparameter that we want to decide. Setting its value, we face the **bias-variance tradeoff**.
When N is too small there is high bias, with model that fails to recover long-distance relations between word. When N is too large, it means we are considering a longer history of words to predict the next word. We could get **data sparsity** (high variance): as N increases, the number of unique n-grams in the training data tends to decrease.
When we refer to "unique N-grams" in the training data, we're talking about the distinct combinations of N items that occur in the dataset. For example, consider the sentence:

$$"The\ quick\ brown\ fox\ jumps\ over\ the\ lazy\ dog."$$

In these sentence there are unique 3-grams as "The quick brown" or as "fox jumps over". As N, the length of n-grams, increases, the number of unique n-grams tends to increase. This is because longer n-grams capture more specific sequences of words, leading to a greater variety of unique combinations in the training data.
The relative frequency estimator can be mathematically obtained by **maximizing likelihood of the dataset**. This can be done by solving **a constrained optimization problem**, using Lagrange multipliers. More precisely, we maximize the log of the product of all sentence probabilities, under the constrain that probabilities sum up to one.
So the relative frequency estimator is also called **maximum likelihood estimator** MLE.
However, there are some practical issues: multiplying many small probabilities results in **underflow** (total result is near to 0). It's safer use **negative log probabilities** $-\log(p)$ and add them. Moreover, for large vocabulary, the model has **huge space requirements**.

4.0.2 Evaluation

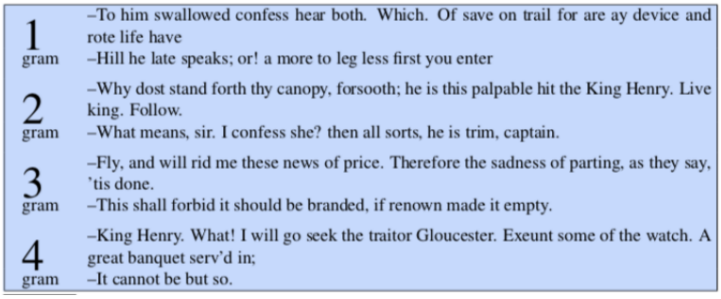
There are two types of evaluation:

- **Extrinsic evaluation**: Use the model in some application and measure performance

- **Intrinsic evaluation**: Look at performance of model in isolation, w.r.t a given evaluation measure.
  It's based on the inverse probability of the test set, normalized by number of words. For a test set $W = w_1 w_2 ... w_n$ we define **perplexity** as:

$$PP(W) = P(w_{1:n}^{-\frac{1}{n}}) = \sqrt[n]{\prod_{j=1}^{n} \frac{1}{P(w_j | w_{1:j-1})}}$$

The factor of produttory can be seen as a measure of how surprising the next word is. The degree of the root averages over all words providing **average surprising per word**.
The perplexity of two language models is only comparable if they use identical vocabularies. The lower the perplexity, the better the model.
An (intrinsic) improvement in perplexity does not guarantee an (extrinsic) improvement in the performance of a language processing task like speech recognition or machine translation.
In Figure random sentences generated by different N-gram models trained on Shakespeare's words.



4.1 Sparse Data

As said before, we could get the problem of sparse data. If there is not enough data in training set, counts will be zero for some grammatical sequences. Then some of the N-gram probabilities will be zero or undefined and the perplexity on the test set will also be undefined. There are three scenario we need to consider.

4.1.1 Zero numerator: smoothing

Smoothing or **discounting** techniques deal with words that are in our vocabulary V but were never seen before in the given context (zero numerator).

Smoothing prevents to assign zero probability to these events.
The idea is to cut a bit of probability from more frequent event and give it to events we have never seen in the training set.
Laplace smoothing doesn't perform well enough but provides a useful baseline.
The idea is to pretend that everything occurs once more than the actual count: for example, if an event appears twice, the counts will be 3.
In order to apply smoothing, let us rewrite the relative frequency estimator as:

$$P(w_t|w_{1:t-1}) = \frac{C(w_{1:t})}{\sum_{u \in V} C(w_{1:t-1}u)}$$

With u a generic word from the vocabulary so means the occurrences of encountering word $w_{1:t-1}$ followed by something.
Let n the number of tokens (length of training set) and $|V|$ is the number of types, fixed.
The relative frequency estimator of the 1-gram model is:

$$P(w_t) = \frac{C(w_t)}{n}$$

since we have not other words, so it's the probability of encounter w_t in the entire training set.
The **adjusted estimate** of the probability is then:

$$P_L(w_t) = \frac{C(w_t) + 1}{n + |V|}$$

Adding $|V|$ to the denominator, we effectively distribute the "extra count" given to each word across all possible words in the vocabulary.
Alternatively, we can think of P_L as applying an **adjusted count** C^* to the n actual observations.

$$C^*(w_t) = (C(w_t) + 1) \cdot \frac{n}{n + |V|}$$

that means that we add one count to each word and we subdivide for the number of words + the dimension of vocabulary. By subdividing it by n, we obtain P_L.
Under this view, the smoothing algorithm amounts to lowering counts for high frequency words and redistributing, since the total number of occurrences of words is equal:

$$\sum_{uinV} C(u) = \sum_{uinV} C^*(u) = n$$

High-frequency words initially have larger counts in the training data. When Laplace smoothing adds a constant value to these counts, the proportional increase for high-frequency words is relatively smaller compared to low-frequency words. This is because adding 1 to a high count has less impact in proportion to the original count than adding 1 to a low count.
Remaining in this topics, we can consider the **relative discount** $d(w_t)$:

$$d(w_t) = \frac{C^*(u)}{C(u)}$$

defined as the ratio of the discounted counts to the original counts.
By solving $d(w_t) < 1$, we find that discounting (riduzione di occorrenze) happens for high frequency types u, like $C(u) > \frac{n}{|V|}$.
A similar result also with Laplace for 2-gram. The 2-gram model relative frequency estimator is:

$$P(w_t|w_{t-1}) = \frac{C(w_{t-1}w_t)}{\sum_u C(w_{t-1}u)} = \frac{C(w_{t-1}w_t)}{C(w_{t-1})}$$

The adjusted estimate of the probability of 2-gram $w_{t-1}w_t$ is then:

$$P_L(w_t|w_{t-1}) = \frac{C(w_{t-1}w_t) + 1}{\sum_u [C(w_{t-1}u) + 1]} = \frac{C(w_{t-1}w_t) + 1}{C(w_{t-1}) + |V|}$$

The adjusted count is:

$$C^*(w_t|w_{t-1}) = \frac{[C(w_{t-1}w_t) + 1]C(w_{t-1})}{C(w_{t-1}) + |V|}$$

$$P_L(w_t|w_{t-1})$$

is larger than

$$P(w_t|w_{t-1})$$

for 2-gram sequences that occur zero or few times in the training set.

However,

$$P_L(w_t|w_{t-1})$$

will be much lower (too low) for 2-gram sequences that occur often. So Laplace smoothing is too crude in practice.

Add-k smoothing is a generalization of add-one smoothing. For some $0 \leq k \leq 1$:

$$P_{add-k}(w_t|w_{t-1}) = \frac{C(w_{t-1}w_t) + k}{C(w_{t-1}) + k|V|}$$

with $k = 0.5$ as common value.
When smoothing a language model, we are redistributing probability mass to outcomes we have never observed. This leaves a smaller fraction of the probability mass to the outcomes that we actually did observe during training. Thus, the more probability we are taking away from observed outcomes, the higher the perplexity on the training data.

4.1.2 Zero denominator: backoff and interpolation

Backoff and interpolation techniques deal with words that are in our vocabulary, but in the test set combine to form previously unseen contexts.
A significant challenge in language modeling is dealing with contexts (word sequences) that were not observed during training. This is a problem because traditional language models, like n-gram models, rely on historical frequency counts to predict the next word. If a specific word sequence (context) has never been seen before, the model might incorrectly assign it a probability of zero, which isn't ideal for making accurate predictions, especially in natural language, where new combinations of words are always possible.
These techniques prevent LLM from creating **undefined probabilities** for these events, or rather zero denominator. In fact, if there is a seequence of word never seen, the frequency at denominator would be around 0.
Backoff combines fine grained models (large N) with rough grained models (low N).
The idea is:

- if you have trigrams, use trigrams
- if you don't have trigrams, use bigrams
- if you don't have bigrams, use unigrams

Katz backoff is a popular but complex algorithm for backoff. However, if we have a large text collections, a rough approximation called **stupid backoff** is sufficient.
In Figure the equation.

For some small λ:

$$P_S(w_t \mid w_{t-N+1:t-1}) = \begin{cases} P(w_t \mid w_{t-N+1:t-1}) = \frac{C(w_{t-N+1:t})}{C(w_{t-N+1:t-1})}, \\ \quad \text{if } C(w_{t-N+1:t}) > 0 \\ \lambda P_S(w_t \mid w_{t-N+2:t-1}), \\ \quad \text{otherwise} \end{cases}$$

P_S is not a probability distribution but it's effective.
Linear interpolation means of combining different order of N-grams by linearly interpolating all the models. For example, for $N = 3$, we have:

$$P_L(w_t|w_{t-2}w_{t-1)} = \lambda_1 P(w_t|w_{t-2}w_{t-1}) + \lambda_2 P(w_t|w_{t-1}) + \lambda_3 P(w_t)$$

with values of λ_i provided in order to optimise likelihood of training data. What are good choices for the λ_j's? Algorithms exist that attempt to optimise likelihood of training data.
We might have different values for the λ_j's, depending on sequences $w_{t-2}w_{t-1}$, subject to

$$\sum_j \lambda_j P(w_{t-2}w_{t-1}) = 1$$

so more occurrences means more probabilities and higher values of λ_j

4.1.3 Estimation of unknown words

Unknown words, also called **out of vocabulary** (OOV) words, are words we haven't seen before.
Replace by new word token $< UNK >$ all words that occur fewer than d times in the training set, d some small number.
Proceed to train LM as before, treating $< UNK >$ as a regular word. At test time, replace all unknown words by $< UNK >$ and run the model.

4.2 Neural language models

N-gram language models have several limitations:

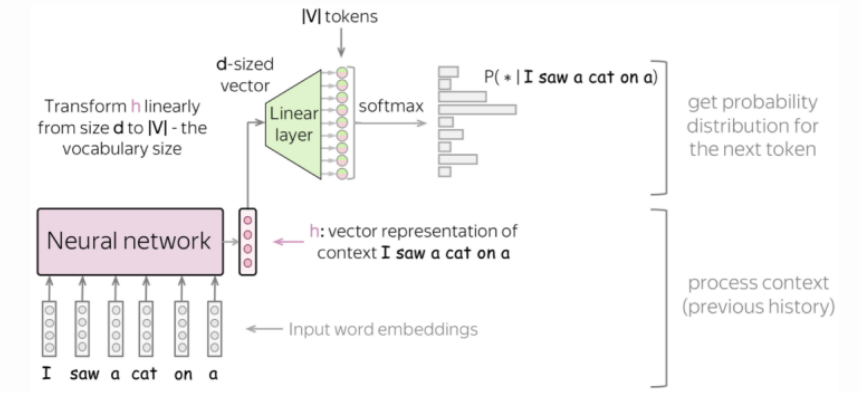
- Scaling to larger N-gram sizes is problematic, both for computational reasons and because of increased sparsity.
- Smoothing techniques are intricate and require careful engineering to retain a well-defined probabilistic interpretation.
- Without additional effort, N-gram models are unable to share statistical strength across similar words.

N-gram language models have been largely supplanted by **neural language models** (NLM). We don't focus on the limitations of LM but we concentrate on the main advantages of NLM that are:

- can incorporate arbitrarily distant contextual information but remaining computationally tractable
- can generalize better over context of similar words

However, NLM are much more complex, slower and less interpretable.
The idea is to get a vector representation for the previous context (of N-1 words) and generate a probability distribution for the next token.
The first task is done by NN architectures like recurrent NN, feedforward NN and convolutional NN.
The general architecture is provided by Figure

General architecture for NLM



4.2.1 Feedforward Neural Networks

A feedforward network is a multilayer feedforward network in which the units are connected with no cycles; the outputs from units in each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers.
When we train it, we want to find the best matrix of weights W that achieve a task with the smallest error.
A feedforward neural language model (LM) is a feedforward network that takes as input at time t a representation of some number of previous words ($w_{t-1}, w_{t-2}, w_{t-3}$ in our example) and outputs a probability distribution over possible next words.
Like the N-gram LM, NLM uses the following approximation:

$$P(w_t|w_{1:t-1}) \approx P(w_t|w_{t-N+1:t-1})$$

and a moving window that can see $N-1$ words into the past.
INFERENCE
For $w \in V$, let $ind(w)$ be the **index** associated with w and we represent each input word w_t as a **one-hot vector** $\vec{x}_t$ of size $|V|$ defined as follows:

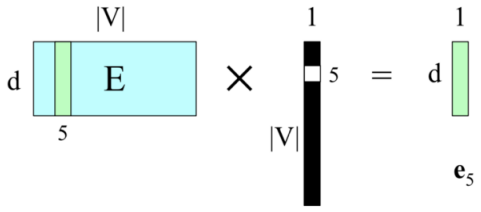
- element of $\vec{x}_t$ at position $ind(w_t)$ is set to one
- all others elements to zero

In the following examples we'll use a 4-gram example, so we'll show a neural net to estimate the probability$P(w_t|w_{t-3}w_{t-2}w_{t-1})$.
At the first layer we convert one-hot vectors for the $N-1$ words into word embeddings of size d and we concatenate embeddings, obtaining;

$$\vec{e_t} = [E\vec{x_{t-3}}, E\vec{x_{t-2}}, E\vec{x_{t-1}}]$$

E is a learnable embedding matrix of dimension $dx|V|$ so, for each column i there is th word embedding of the one hot vector of word w_t at position $i = ind(w_t)$ of dimension d for each word.
We'll see that is a parameter trained, in order to guarantee better result, initialized random and step by step tuned.
By multiplying it with an one-hot vector that has only one non-zero element $x_{ind(w_t)} = 1$, we simply select out the relevant column vector (embedding) for

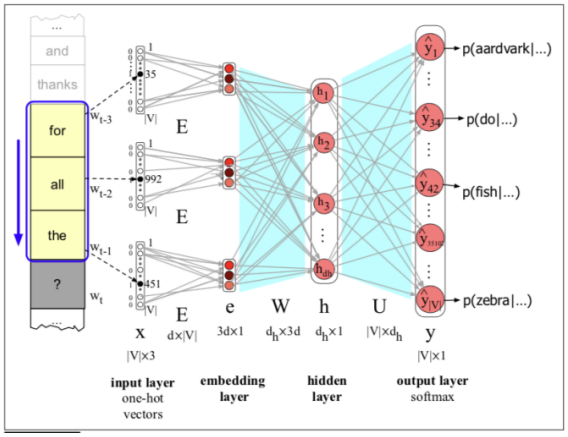
Conversion from 1-hot word representation into word embedding.



word w_t, resulting in the embedding for word w_t, as shown in Figure.
These three embedded vectors are concatenated in order to produce $\vec{e}_t$, the **embedded layer**.
Then, in the remaining layers, we multiply this embedded layer for the matrix of weight W of dimension $d_h x 3d$, with d_h the size of the second hidden layer.
Then we pass the result $(WE_t + \vec{b})$ into an activation function g, obtaining $\vec{h}_t$.
Then we multiply for a matrix U, obtaining the vector $\vec{z}_t$, of dimension $|V|$ and this vector is passed into the **softmax equation**.
Softmax equation normalizes results into a probability distribution (which can be sum up to 1), obtaining at the end $\vec{y}_t$, our probabilities for various word w_t, one for each cell: the dimension of the vector is $|V|$ and for each type of the vocabulary there is the probability of encountering that type as next word, in the cell $ind(w_t)$. The vector $\vec{z}_t$ can be thought as a set of scores called **logits** non normalized. The elements with index $ind(w_t)$ of $\vec{y}_t$ is the following probability:

$$P(w_t|w_{t-3:t-1})$$

In Figure there is the graphical representation of this concept.



TRAINING

We don't talk about **training**: the parameters of the model that we want to learn are $E, W, U, \vec{b}$.

Let w_t be the word at position t in the training data (so known at priori). Then the true distribution $\vec{y}_t$ for the word at position t is a 1-hot vector of size $|V|$ with 1 only at position $ind(w_t)$, meaning that the probability of encountering that word w_t is 1 since it's known.

So we use cross entropy or equivalently negative log likelihood for training the model as loss function:

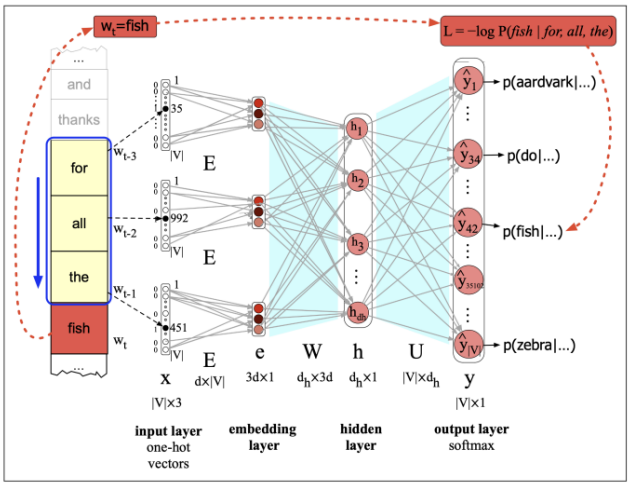
$$L_{CE}(\hat{\vec{y}}_t, \vec{y}_t) = -\sum_{k=1}^{|V|} \vec{y}_t[k] \cdot \log(\hat{\vec{y}}_t[k]) = -\log(\hat{\vec{y}}_t[\text{ind}(w_t)])$$

That is the expected information of $\hat{\vec{y}}_t$ computed w.r.t. $\vec{y}_t$. That is equal to

$$L_{CE}(\hat{\vec{y}}_t, \vec{y}_t) = -\log P(w_t \mid w_{t-N+1:t-1})$$

so we want to minimize that (it's equal to maximizing the negative of that, or rather maximizing the likelihood).

In Figure the graphical representation of training: we know that the next word is fish, we train it in order to get higher probability for that word in the modeled $\vec{y}_t$.

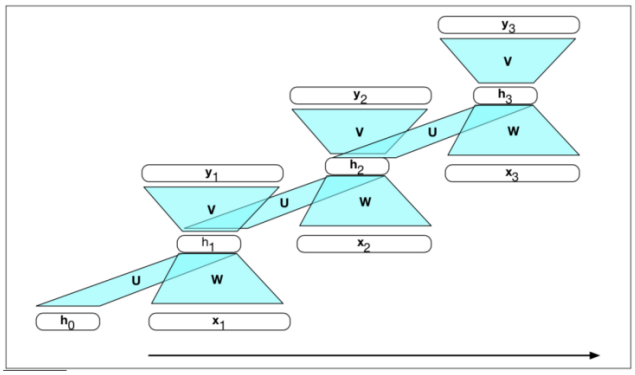


4.2.2 Recurrent NLM

A recurrent neural network (RNN) is any network that contains a cycle within its network connections, meaning that the value of some unit is directly, or indirectly, dependent on its own earlier outputs as an input. While powerful, such networks are difficult to reason about and to train.
Talking about our task or rather try to predict the next word, RNN LM process the input one word at a time, predicting the next word by using the current word and the previous **hidden state**.
It can model probability distribution $P(w_t|w_{1:t-1})$ without the window of $N-1$ words of NLM.
The hidden state represents information about all preceding words.
INFERENCE
To compute an output $\vec{y}_t$ for an input $\vec{x}_t$, we use the activation value for the hidden layer $\vec{h}_t$.
To calculate this, we multiply the input $\vec{x}_t$ (one hot vector encoding) with the usual embedding matrix $E : d_h x|V|$, obtaining $\vec{e}_t$, then for the weight matrix W and sum up to the hidden layer from the previous time step $\vec{h_{t-1}}$ multiplied by the weight matrix U. At the end is passed to an activaction function:

$$\vec{h_t} = g(U\vec{h_{t-1}} + W\vec{e_t})$$

Once we have obtained the hidden layer, we compute the output layer by using softmax equation of the product between $\vec{h_t}$ and V.
The fact that the computation at time t requires the value of the hidden layer from time $t1$ mandates an incremental inference algorithm that proceeds from the start of the sequence to the end, as shown by Figure.



For each word $w \in V$, the element of $\vec{y}_t$ with index $ind(w)$ estimates the probability that next word is w:

$$y_t[\vec{ind}(w)] = P(w_{t+1} = w|w_{1:t})$$

TRAINING

Dealing with **training**, the number of parameters of the model is $O(|V|)$, since d is a constant.

Let $\vec{y}_t$ be the true distribution at step t: this is a 1-hot vector over V, obtained from the training set.

We train the model to **minimize** the error predicting the true next word w_{t+1} in the training sequence, using the cross-entropy as loss function (remember that the cross entropy is the negative of the log likelihood).

$$L_{CE}(\hat{\vec{y}}_t, \vec{y}_t) = -\sum_{w \in V} \vec{y}_t[ind(w)] \cdot \log \hat{\vec{y}}_t[ind(w)] = -log(\hat{\vec{y}}_t[ind(w_{t+1})])$$

So at each step t during training there is a prediction on the correct sequence of tokens $w_{1:t}$ and we ignore what the model predicted at previous step.

The idea that we always give the model the correct history sequence to predict the next word is called **teacher forcing**.

Until now we have talk about word-level NLM but exists also **Character-NLM**, which improves modelling of uncommon and unknown words and reduce training parameters due to the small softmax. It has usually worse performance, since longer history is needed to predict the next word correctly.

4.2.3 Practical issues

Both feedforward and recurrent NLM learn word embeddings E simultaneously with training the network. Alternatively, we can resort to **freezing**: we use a pretrained word embeddings and hold E constant while training, modifying the reamining parameters in θ.

In the recurrent NLM model, the columns of E provide the learned word embeddings while the rows of V provide a second set of learned word embeddings, that capture relevant aspects of word meaning and function.

Weight tying, also known as **parameter sharing**, means that we impose:

$$E^T = V$$

Had an effect similar to **regularization**, preventing overfitting.
RNNs suffer from the **vanishing gradient problem**: past events have weights that decrease exponentially with the distance from actual word w_t.
The last step in NLMs, involving softmax over the entire vocabulary, dominates the computation both at training and at test time. An effective alternative is **hierarchical softmax**, based on word clustering, or **Adaptive softmax**.
The text generated by sampling our NLM should be **coherent** and **diverse** (model has to be able to produce very different samples) and there is a trade off between this two.
A very popular method for modifying language model behavior is to change the **softmax temperature** τ, as shown by Figure

$$\frac{\exp(\frac{\mathbf{V}_i \mathbf{h}_t}{\tau})}{\sum_j \exp(\frac{\mathbf{V}_j \mathbf{h}_t}{\tau})}$$

where $\mathbf{V}_i$ denotes the i-th row of $\mathbf{V}$.

Low τ produces peaky distribution (high coherence); large τ produces flat distribution (high diversity).

Contrastive evaluation is used to test specific linguistic constructions in NLM, as shown in Figure

Example : Subj/Obj agreement, compare $P(\text{is} \mid w_{1:t-1})$ to $P(\text{are} \mid w_{1:t-1})$

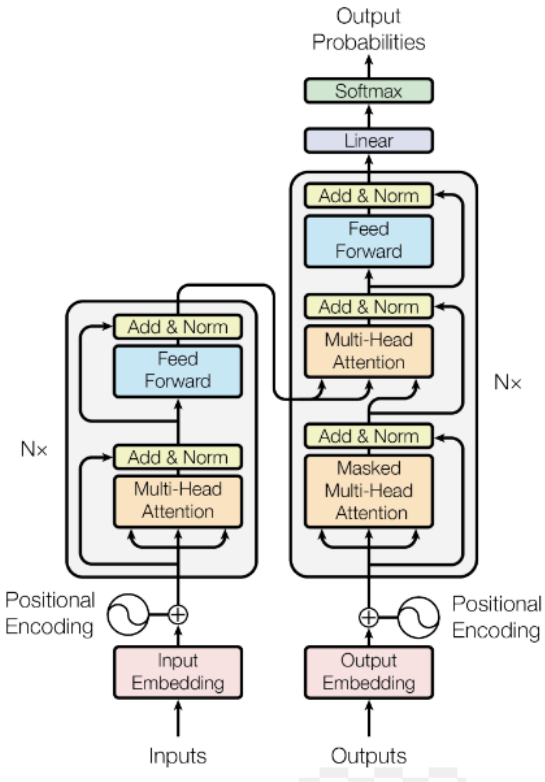
- The roses ___
- The roses in the vase ___
- The roses in the vase by the door ___

5

Large Language Modelling

5.1 Review of Transformer architecture

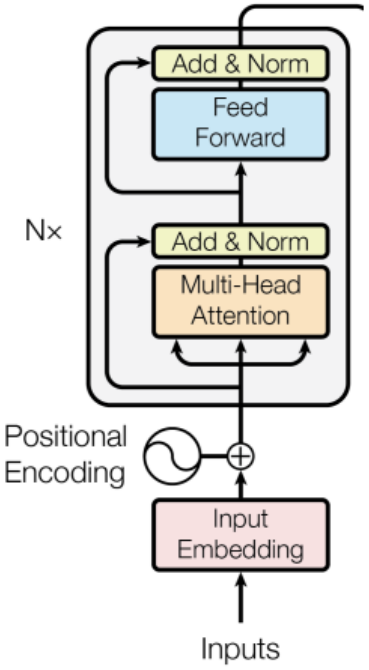
The **transformer** is a NN architecture modelling sequence to sequence: it turns one sequence into another sequence. The original transformer model uses an **encoder-decoder** architecture, as shown in Figure.



As we can see, it is composed by an encoder (sx) and a decoder (dx) blocks; each block is composed by N blocks of many layers.
Transformers differ in their ability to capture long-range dependencies and contextual information.

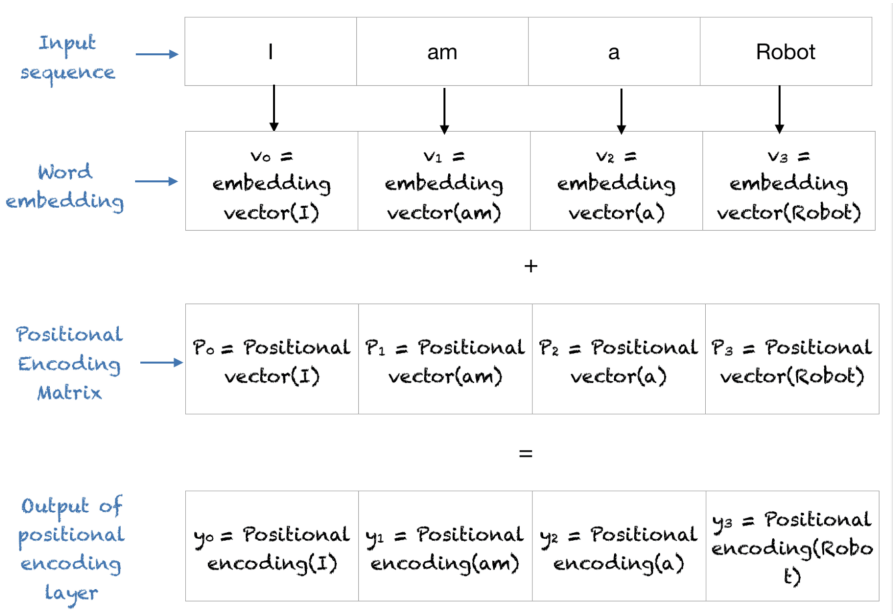
5.1.1 Encoder

In Figure the first part of a Transformer: the encoding part.



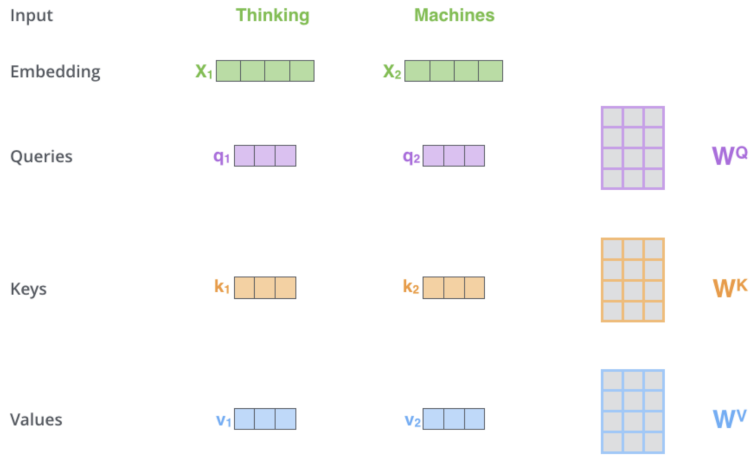
EMBEDDINGS IN ENCODER: The first step in transformer operations is understanding the input data. It takes a sentence — or a sequence of data — and turns each word or element into numerical representation known as **vector embeddings**. The sequence model's embeddings simply capture the meanings of the words or elements, not referred to a context. Since the embedding is taken from some pretrained model (e.g Word2Vec), if there are two sentences with a word in common, this word has initially the same embedding.
Transformers don't naturally understand the order of words, so they use **positional encoding** to give the model information about the order. It's a technique that transformers use to keep track of word order. There are a bunch of way to do it, and we focus on the most popular method. For example, assuming a sentence of four words, we have our single embeddings for each word. We sum

to it a **positional encoding**, provided by a sinusoidal function, obtaining so the **positional embedding**, which know the position of words in the sentence. In Figure this first part visualized



Talking about the example where two sentences have a word in common, in this case, the embeddings for these two words would be different.
Then this sequence is passed through multiple (N) encoder layers, a sort of stack. Each layer consists of two sub-layers called the **self-attention mechanism** and **the feed-forward neural network**, with both an add and normalization layer.
SELF-ATTENTION MECHANISM: The self-attention mechanism allows the model to focus on different parts of the input sequence and capture dependencies. It allows each token to "talk" to each other token and modify embedding based on it.
For example, if I have a sentence like "I love pizza, it's so tasty", self-attention mechanism allows to understand that "so tasty" is referred to pizza.
It calculates attention scores for each element based on its relationships with other elements in the sequence.
For the sentence above, **self-attention** calculates the similarity between the first word, "I", with all the other words of the sentence. The score depends on how frequents words are associated.
The first step in calculating self-attention is to create three vectors from each token of the context window, so of the encoder's input. So, for each token, we create a **Query vector, a Key vector, and a Value vector**.

These vectors are created by multiplying the embedding of each token by three matrices that we trained during the training process, as shown in Figure



Remembering that values are numerical, and supposing that we are referring to token with embedding $\vec{E}_i$, a query $\vec{Q}_i$ asks, "Given the current word I'm focusing on, which other words should I pay attention to?". Key vectors for each word in the context window represent what those words have to offer to any other word that is seeking information. Each key helps other words in the sentence decide how much attention to pay to it.

The second step in calculating self-attention is to calculate a **attention score**. Supposing that we're calculating the self-attention for the first word in this example, "Thinking", we need to score each word of the context window against this word.

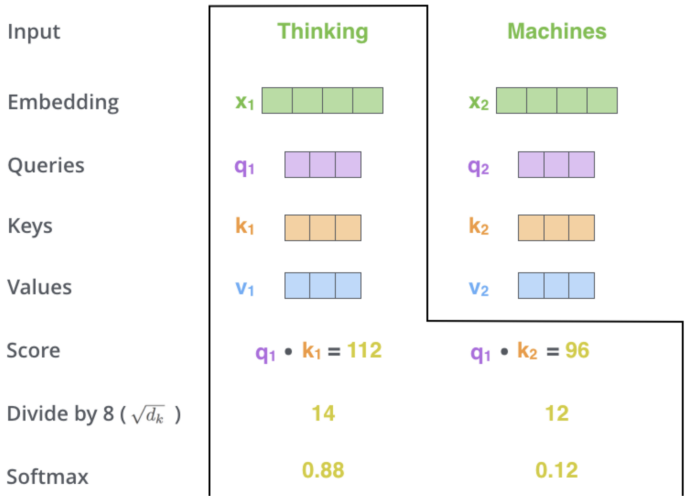
The model uses the match between query and key to determine the relevance or importance of the other words in the context relative to the current word the model is focusing on.

The score is calculated by taking the dot product of the query vector of the considering word with the key vector of all other words. So if we're processing the self-attention for the word in position 1, the first score would be the dot product of q_1 and k_1 and the second score would be the dot product of q_1 and k_2.

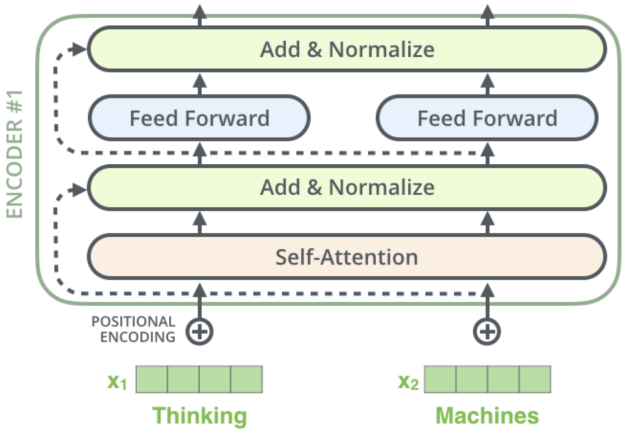
So, for each word, we obtain a vector containing how much is related this word with others.

After, the vector of product is passed through a **softmax function** (is a function that turns a vector of real values into a vector of real values that sum to 1, that can be interpreted as probabilities. If one of the inputs is small or negative, the softmax turns it into a small probability, and if an input is large, then it turns it

into a large probability, but it will always remain between 0 and 1.) in order to obtain a sort of vector of weight: if the dot product between a specific key and the query is high, the word associated with that key is very important in the context of the word associated with that query, so the weight will be high.
Always considering a single word with embedding $\vec{E}_i$, we multiply the attention score vector just computed for values vector of all other words in the context. Remember that the attention score is used as weight, in order to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).
So, resuming, for each word in the context of a word w_i, we have a value of how much each word gives contribute to the new embedding of w_i. This value is weighted on the attention score, which capture how two words are related. So, as output of the attention-mechanism, we have, for each word i in the context, a vector $\vec{\Delta}_i$ which contains this weighted sum already computed. So, these vectors contain information about the context for a single word w_i.
This process is done for all word in the context window and in parallel (**multi-head attention mechanism**), in order to relate the input word to each other words in many different ways simultaneously.
By using different attention-mechanism, we allow the model to learn all possible ways how context change the meanings. By each self-attention layer we will have the value to sum to the embeddings in order to capture the correct context: we will concatenate all the vectors of output for a single input embedding $\vec{E}_i$.
This process is done for many multi-head attention layer, in order to guarantee that the context is well incorporate.
These steps are well represented in the Figure
This output is then used in further processing layers of the model. **RESIDUAL CONNECTIONS + NORMALIZATION** As said before, each sub-layer (self-attention, FFNN) in each encoder has a residual connection around it, and is followed by a layer-normalization step; these are critical components that help in stabilizing training and enhancing the ability of deep networks to learn effectively.
Residual connections help mitigate the vanishing gradient problem (sum of weight that progressively goes to 0) that can occur in deep networks, making it difficult for lower layers to learn effectively.
A residual connection in a transformer takes the output from an earlier layer, adds it to the output of a stacked layer (like a self-attention or feed-forward



layer), and then passes this sum to the next layer.
So, if the output of self-attention referred to input word x is $A(x)$, then the output of residual connection is $(x + A(x))$.
In our case, we sum, for each vector, the initial embedding $\vec{E}_i$ with the vector containing information about the context $\vec{\Delta}_i$, obtaining so the updating of the vector.
Layer normalization normalizes the input across the features instead of across the batch. For each input vector (corresponding to each token in the context of transformers), it normalizes the vector based on its mean and variance. Layer normalization improves performance by keeping the values of a hidden layer in a range that facilitates gradient-based training.
In Figure what happens in the encoder block



FEED FORWARD NETWORK: The feed-forward neural network applies a

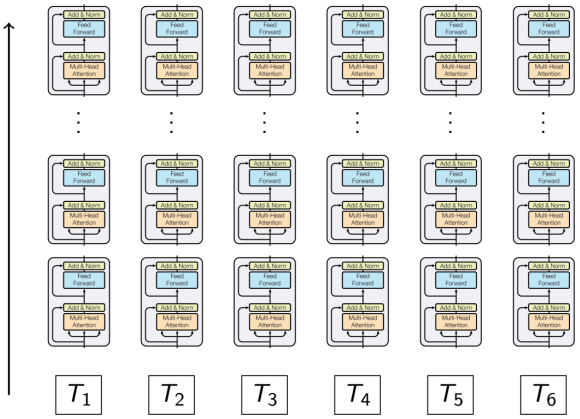
non-linear transformation to every vector to transform then into a form that is acceptable to the next encoder or decoder layer, introducing complexity and expressive power to the model. The feed-forward layer makes up two-thirds of the parameters in a transformer model.
The FFNN in a transformer consists of two linear transformations with a non-linear activation function in between. The exact structure for each position i in the sequence is as follows:

- **First Linear Layer:** Transforms the input vector from the self-attention layer (dimension d_{model}) to a higher dimension
- **Activation Function:** A non-linear function like ReLU (Rectified Linear Unit) or GELU (Gaussian Error Linear Unit) is applied. This introduces non-linearity into the processing, enabling the network to learn more complex patterns.
- **Second Linear Layer:** Transforms the vector back to d_{model}.

Assuming the output from the self-attention layer for a particular token in the sequence is a vector x_i, here is how the FFNN processes this vector:

1. **Input:** x_i
2. **First Linear Transformation:**
$$z_i = \text{ReLU}(x_i W_1 + b_1)$$
where W_1 is the weight matrix and b_1 is the bias for the first layer.
3. **Activation:** The ReLU function is applied to z_i, adding non-linearity:
$$\text{ReLU}(z) = \max(0, z)$$
4. **Second Linear Transformation:** The activated vector z_i is then multiplied by another weight matrix W_2 and another bias b_2 is added to bring the dimension back to d_{model}:
$$y_i = z_i W_2 + b_2$$
where these matrices are learned as parameter.

At the end, the resulting output is passed through a residual connections plus normalization layer.
Encoder output is a high level, contextualized version of the input, serving as basis for decoding computation.
Before moving on the decoder part, it's important to say that the encoder takes as input a text and the adjustments are done parallelized for each token, as shown in Figure, where each T_i is a token.

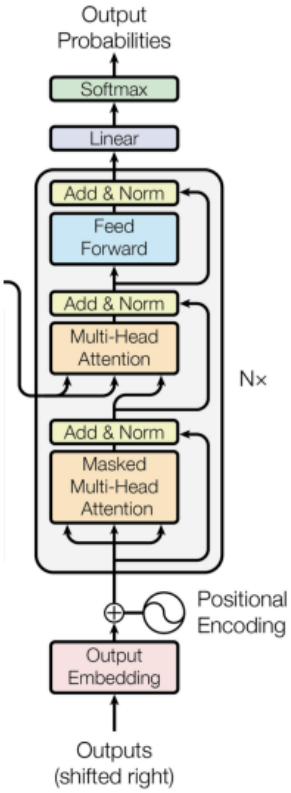


5.1.2 Decoder

In a transformer model, the decoder's primary role is to generate the output sequence (in machine translation, this would be the translated text) based on the input sequence provided to the encoder. The decoder receives two main types of inputs:

1. *Output from the Encoder*
2. *Previously Generated Tokens*: This is the sequence of tokens that the decoder has generated so far. During training, these tokens are typically the actual target tokens: since it's a supervised learning task, are provided, during training, the input sequence and the target output sequence, shifted to the right of one position. Supposing that the correct target is *Hello World Love*, we give as input <*SOS*> *Hello*, where <SOS> is the start of sequence token. If the predicted word is correct, in this case, World, we give it as input after Hello, and we move on.
   Instead, if there is an error on the predicted word, since we have the correct output, we adjust, very low, parameters in encoder and in decoder, in order to reduce the loss; it is given as input not the incorrect word, but the correct one, in our case, Love.
   This is known as **teacher forcing**, where the true output sequence from the training dataset is used as the next input, regardless of what the model predicts at each step. This ensures that the model is always "guided" by the correct sequence during training, which helps prevent the accumulation of errors.
   Instead, during inference (test the model on unseen words), the decoder starts with the <SOS> token and then uses the tokens it has predicted in the previous steps. For instance, if it has predicted "Hello", the next input will be <SOS> Hello, and it tries to predict the next word based on this input.

In Figure the representation of the decoder of a transformer.
The input of the decoder is then positional embedded, as the encoder, and it is



passed into stack of N identical blocks with three layers each.
The first block encountered is the **Masked Self-Attention Layer**: This layer ensures that the prediction for a token at position i can only depend on tokens at positions less than i. This is implemented using a mask that prevents positions from attending to future positions in the sequence. This maintains the autoregressive property of the model.
After that we take the output of the masked self-attention layer as query, and the encoder output as keys and values and we pass it through a **cross attention layer**, which allows the decoder to "attend" to different parts of the input sequence, focusing on the areas that are most relevant to generating the next token.
Then the output is given as input to a FFNN, with the same function of the encoder one: transforming the representation for the next token.
At the end, after an application of a linear function, the decoder apply softmax, returning a vector containing probability for each possible next word.
It's important to say that not always is chosen the most probably word, in order to give an answer that is varied.

NB: each layer is followed by Residual Connection around it and Layer Normalization.
Some applications use encoder only, or else decoder-only.

5.2 Contextualized Embeddings

Word embeddings, that we have seen before, learn a single vector for each **type** in the vocabulary V. That's why it's also called **static embeddings**.
Contextualized embeddings represent each token by a different vector, according to the context the token appears in.
It's also called **pre-trained language model** or dynamic embeddings and incorporating context into word embeddings has proven to be a watershed idea in NLP, opening the way to **transfer learning and fine tuning**.
How can we learn to represent words along with the context they occur in?
We train a NN combining ideas from word embeddings and language models in different ways:

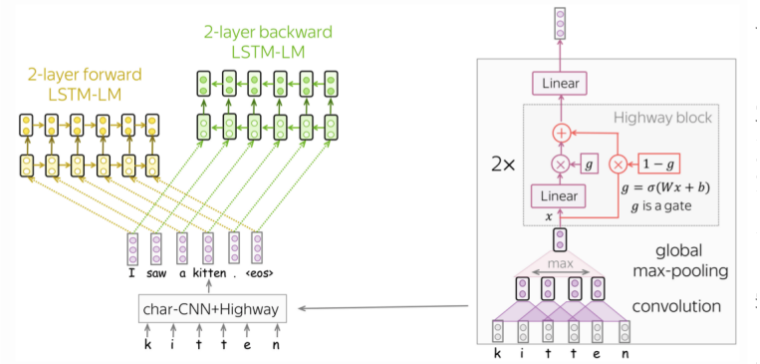
- predict a word from left context (as GPT-n family)
- predict a word from left and right context **independently** (as ELMo)
- predict a word from left and right jointly (as BERT)

Before starting, it's important to provide a preview on what will come next. When we deal with generative IA, we need to learn not only the meaning of the word, but also the context on where it is.
We need to **pre-train** our model: pre-training is the first stage where a is trained on a large, general dataset.
This step is crucial because it allows the model to learn a broad understanding of the language, including vocabulary, grammar, and contextual relationships between words.
This is done by **language models**, which are trained in order to be able to predict the next word, based on the context. This generalized training helps the model develop what's known as contextual embeddings, where each word's representation is informed by the words around it. ELMo and BERT are Language Models, since they have less than 1 billion parameters. Instead, GPT-n models are considered LLM.
The step after is to **fine-tune** the model: after pre-training, the model is trained further on a smaller, task-specific dataset. For example, if the task is sentiment analysis, the model might be fine-tuned on a dataset of movie reviews where each review is labeled with a sentiment. During fine-tuning, all or most of the model's parameters are adjusted. Sometimes, additional layers are added (like a classification layer), but the bulk of the model remains the same. The fine-tuning

process adjusts the pre-trained parameters to better suit the task, refining the embeddings and the model's predictive abilities.
The presentation follows an historical order, all presented models are language models.

5.2.1 ELMo

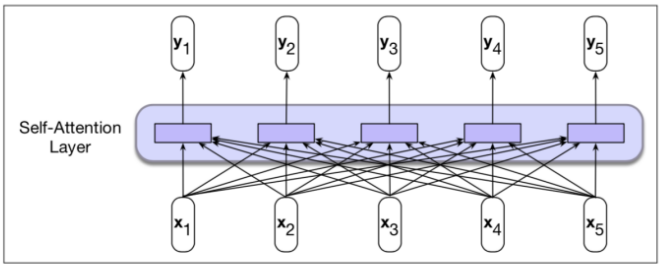
ELMo (embeddings from language model) looks at the entire sentence before assigning each word its embedding.
Tokens are processed by a character-level convolutional neural network (CNN) producing word-level embeddings.
This captures word morphology and resolves OOV words. These embeddings are processed by a left-to-right and a right-to-left, 2-layers LSTM, (Long Short-Term Memory). This is also called a bi-directional LSTM.
For each word, the output embeddings at each layer are combined, producing contextual embeddings, as shown in Figure



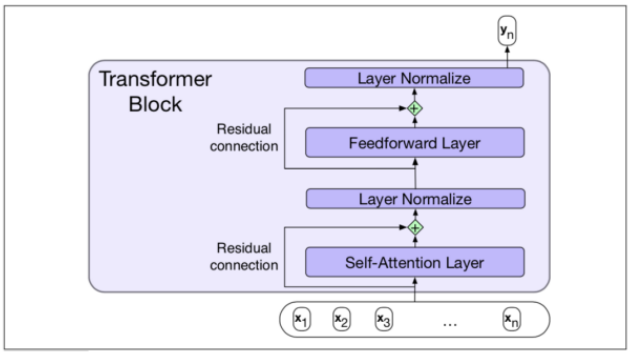
When **training**, add output layer with linear transformation to $|V|$ dimension and to softmax. At the end use cross-entropy as in NLM.
When input is encoded, model ignore output layer and retain the word embedding represented in the last hidden layer.
In this case, ELMo is not a LLM since it's not based on a transformer and it has not more than 1B parameters.

5.2.2 BERT

Bert is the acronym of bidirectional encoder representations from transformers. It produces word representations by jointly conditioning on both left and right context.
The model is based on the encoder component of the transformer neural network. Tokenizer uses WordPiece algorithm.
The bidirectional encoding in BERT is inherited from the self-attention mechanism of the transformer encoder, as shown in Figure



Input is segmented using subword tokenization and combined with positional embeddings. Then input is passed through a series of standard transformer blocks consisting of self-attention and feedforward layers, augmented with residual connections and layer normalization, as shown in Figure

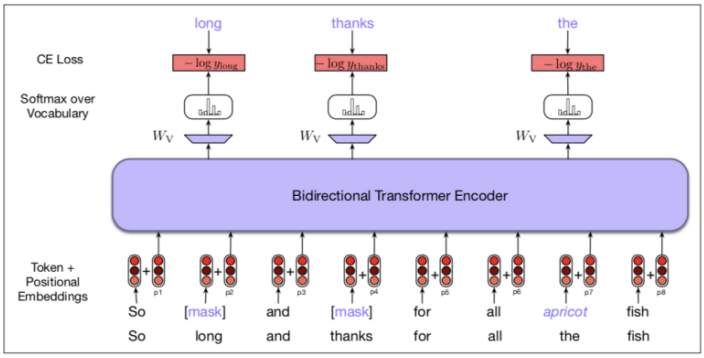


There are two ways to train the model: in the first one, the model learns to perform a fill-in-the-blank task, technically called the **cloze task**.
The approach is called **masked language modelling** (MLM), which understand relationship between words. We pick up sentences from web (billion of sentences), we remove randomly one word and we ask to transformer to set param-

eters in order to learn which word we have to put inside the hole.
We take randomly 15 % of tokens and:

- 80% are replaced by token $[mask]$
- 10% are replaced by another random token
- 10% left unchanged

The structure used is shown in Figure



We embedds token with positional embeddings + word embeddings. Then, we pass over a bidirectional transformer encoder and after the softmax over vocabulary, obtaining the correct word.
However, many important tasks involve relationship between pairs of sentences:

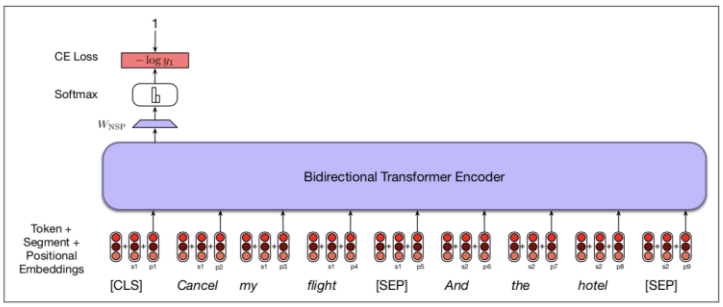
- paraphrase detection: detecting if two sentences have similar meanings, also called paraphrase identification
- semantic textual similarity: detecting the degree of semantic similarity between two sentences
- sentence entailment: detecting if the meanings of two sentences entail or contradict each other
- answer sentence selection: detecting if the answer to a question sentence is contained in the second sentence

This is not directly captured by language modeling.
In order to train a model that understand sentence relationships, BERT introduces a second learning objective called **next sentence prediction**.
In the NSP objective, BERT is trained to predict if one sentence follows another in a text pair. This helps BERT comprehend the logical connections between sentences, making it a master at understanding paragraphs and longer texts.

NSP is a binary decision task that can be trivially generated from any monolingual corpus. It produces sentence embeddings that can be used for tasks involving relationship between pairs of sentences.
The model is presented with pairs of sentence with:

- token $[CLS]$ before the first sentence
- token $[SEP]$ placed between the two sentences
- **segment embeddings** marking the first and second sentences are added to each word and positional embeddings

Each pair consists of an actual pair of adjacent sentences from the training corpus (50%) and a pair of unrelated sentences randomly selected (50%). We train our model using this, shown in Figure



The result of the MLM and NSP pre-training processes consists of the encoder, which is used to produce contextual embeddings for tokens in novel sentences. It is common to compute a representation for token w_t by averaging the output embeddings at t from each of the topmost layers of the model.
NSP embedding is also used as **sentence embedding** for tasks involving single sentence classification.
BERT is trained on both tasks, in order to obtain a better contextualized embeddings; however, it can not be classified as a LLM since it has been trained on less than 1B parameters.

5.3 Large Language Models

A language model is called **large language model** if it has the following features:

- neural network architecture based on transformer
- number of parameters larger than 1 billion
- pre-trained on vast amount of textual data
- can be adapted to a wide range of downstream tasks

Sometimes also called foundational models, LLMs are at the basis of modern applications of generative artificial intelligence.

5.3.1 GPT family

GPT is the acronym of generative pre-training transformer. It's a left to right language mod
GPT-2 is based on transformer's decoder-only, no cross-attention layer.
Layer normalization moved to the input of each sub-block. Tokenization is performed with BPE with vocabulary size from 32,000 to 64,000.
GPT-3 has the same basic architecture as GPT-2: transformer's decoder-only, including pre-normalization.
From GPT-2 the model adds several optimization techniques, including **attention sparsity** at half of the layers, computing only a subset of the elements of the attention matrix.
It has been trained on large textual collection and due to this reason, novel capabilities of LLMs emerge, called **in-context learning** (refers to the ability of a model to understand and perform tasks based solely on the examples or instructions provided within the immediate context of its input, without any prior explicit training on that specific task. This is different from the traditional approach of machine learning, where models are usually fine-tuned on a specific dataset tailored to a particular task.).
GPT-3 is capable of solving a problem on the basis of a few examples (**few-shot learning**) and is capable of solving new problem solely on the basis of natural language instructions (**zero-shot learning**).
GPT-4 is a **multimodal** large language model, capable of processing image and text inputs and producing text outputs. Two versions, with context windows of 8192 and 32768 tokens. It can interact with external interfaces.

There are other several different LLMs, as we can seen from slides.

5.3.2 Multilingual large language models

A MLLM is a computational model that is designed to understand, interpret, and generate text in multiple languages. These models leverage large-scale datasets composed of texts from various languages to learn representations that can capture the nuances, grammar, and semantics of each language included in the training process. The objective is to create a single model that can perform a given NLP task across different languages without the need for separate models for each language.
They are typically based on the encoder part of the transformer architecture. A MLLM is pre-trained using a large amounts of unlabeled data from multiple languages and it's very useful in **transfer learning** where low resources language can benefit from high resource languages.
It's trained by:

- During pre-training, MLLMs use two different sources of data, large monolingual corpora in individual languages and parallel corpora between some languages (parallel corpora consist of text pairs where each pair is a translation of the same content in two different languages).
- The objective function for training multilingual language models can be effectively designed using only monolingual data, by amalgamating monolingual text corpora from a variety of languages into a single, large dataset. This dataset is then utilized to train the model using a technique known as Masked Language Modeling (MLM).

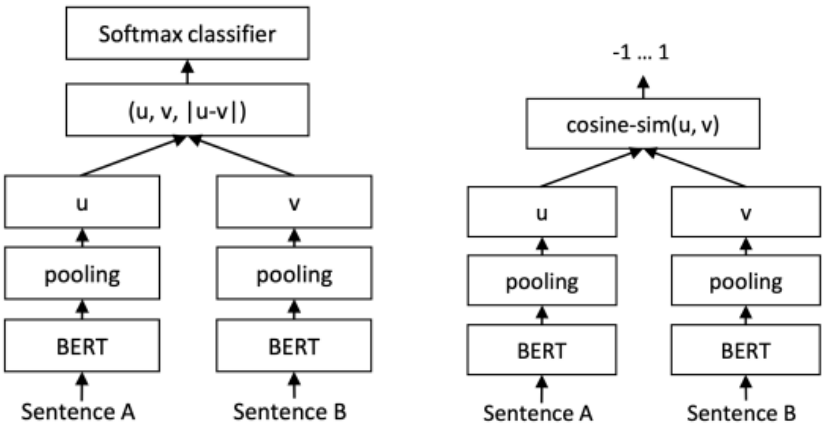
Doing this, the encoder learns word representations across languages that are **aligned (close vector)** without the need for any parallel corpora.
Objective functions based on parallel corpora **explicitly** force representations of corresponding words across languages to be close to each other in the multilingual encoder space.
Using MLM, to predict a masked word for a language A, the system can:

- rely on words for language A surrounding $[MASK]$
- rely on aligned word that has not been masked, that is a word representing a translation in a language B of the masked word

Since parallel corpora are generally much smaller than monolingual data, the parallel objectives are often used in conjunction with monolingual models.

5.3.3 SBERT

In some sentence-pair regression tasks, (is a specific kind of task where the goal is to predict a numerical value that represents the relationship between two sentences) training BERT on all sentence pairs is NP.
BERT and RoBERTa require that both sentences are fed into the network, which causes a massive computational overhead: finding the most similar pair in a collection of 10,000 sentences requires about 50 million inference computations (65 hours) with BERT. The construction of BERT makes it unsuitable for semantic similarity search as well as for unsupervised tasks like clustering.
Alternatively, **individual sentence embeddings** can be computed by averaging BERT embeddings or by using the output of CLS token. However, it results bad.
Sentence-BERT or **SBERT** is an efficient and effective method to compute sentence embeddings that can be compared using cosine-similarity.
It uses a siamese neural network, a special network that contains two identical subnetworks sharing their parameters and are widely used to compute **similarity scores** for the inputs.
The architecture is the one provided in Figure



In Figure there are two different architecture: the one on the left that is referred to training part and, the one on the right, referred to inference, which provide a similarity score between these two.
SBERT adds a pooling operation to the output of BERT / RoBERTa to derive a fixed sized sentence embedding n.
The sentence embeddings u and v are concatenated with the element-wise difference $|uv|$ and multiplied with the trainable weight W and Cross-entropy loss

is optimized to compute

$$o = softmax(W[u; v; |u - v|])$$

with $W \in \mathbb{R}^{3nxk}$ updated across both sub-networks.
At the end, the two embeddings are compared using a cosine similarity function, that will output a score for the two sentences.

5.3.4 Miscellanea

Since LLMs are new developments, there are **emergent abilities** of LLMs that are been discovered, like **sharpness** (transitioning instantaneously from not present to present) or **unpredictability**(s the models become larger and more complex, new and unforeseen capabilities and behaviors are likely to emerge).
As we can see by Figure, by increasing the size of the model, we can obtain very different result on some tasks.
LLMs can be affected by **hallucination**, referring to phenomenon where a LLM generates text that is incorrect, nonsensical or not real. Since LLMs need to be creative, they must be able to invent scenarios that are not factual.
Mixture of Experts (MoE) is a machine learning technique where multiple expert networks (learners) are used to divide a problem space into homogeneous regions. Each expert is typically a specialized model trained to handle a specific subset of the data, while the subdivision of regions decides which experts to consult for a given input.
This technique has been recently used in LLMs. The result is a sparsely-activated model with a large number of parameters but a constant computational cost.

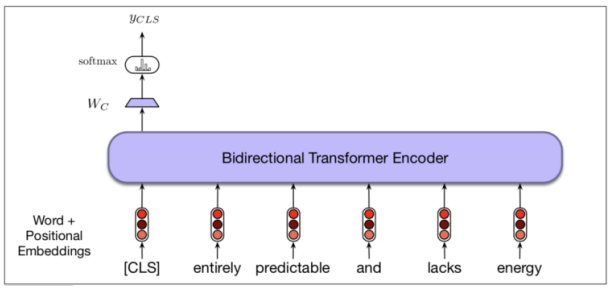
5.4 Fine-Tuning

To make practical use of LLMs, we need to interface these models with downstream applications (as sentiment analysis, speech recognition ecc).
We have seen that we train our models by using a pre-train technique, which confer embeddings for words based on them context and these models are trained only on the task of next word prediction (or similar).
Since there is a huge variety of task, we need to modify our parameters for the specific task we'll focus on.

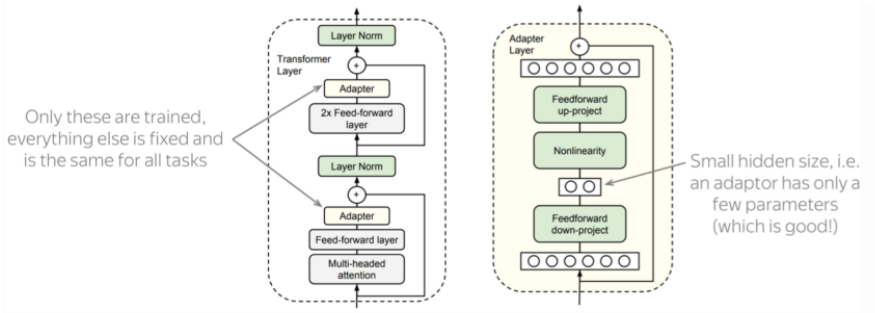
This process is called **adaptation** and it uses supervised learning for the task of interest.
Two most common form of adaptation are:

- **feature extraction** freeze the pre-trained parameters of the language model and train parameters of the model for the task at hand
- **fine tuning** make adjustement to the pretrained parameter

Retraining the whole model could be expensive: it's called **catastrophic forgetting**.
For example, in sentiment analysis, only top layer (the output embedding for CLS, z_{CLS}) need to be classified, so we add a layer as shown in Figure.



When we work with huge pre-trained models, fine tuning may still be inefficient. Alternatively, one could fix the pre-trained model and train only small simple components called **adapters** so the largest part of pre-trained model is shared between all downstream tasks.
Transformer with the **adapter module** consisting of a two layer feed forward network with a nonlinearity and a residual connection, as shown in Figure. As

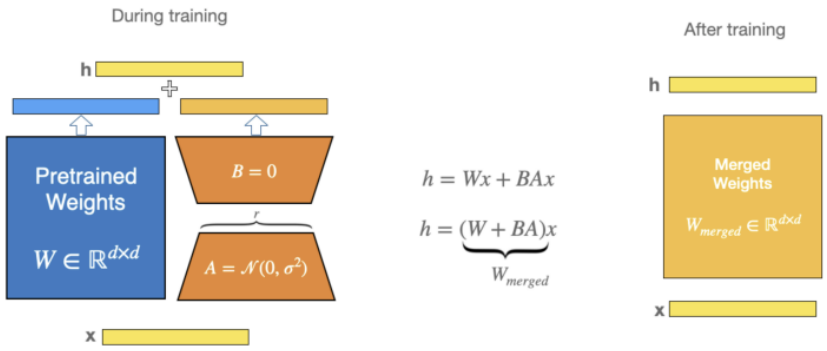


we can see from Figure, the yellow box represents the architecture of adapter: it has a small dimension of embeddings that means it has **only few parameters**.

Adapter tuning attains a similar performance to top-layer fine-tuning, as shown by Figure.
LoRA (Low Rank Adaptation) is a popular fine-tuning strategy, alternative to adapters. It reduces the number of trainable parameters and is based on **intrinsic dimensions**: there exists a low dimension reparameterization that is as effective for fine-tuning as the full parameter space.
LoRA is known for its high efficiency, and for avoiding catastrophic forgetting.
We can think the **weight update** as follows:

$$W \longleftarrow W + \Delta W, W \in \mathbb{R}^{dxd}$$

Instead, in LoRA, we update the weight by decomposing ΔW into two low-rank matrices $\Delta W = BA, B \in \mathbb{R}^{dxr}$, $A \in \mathbb{R}^{rxd}$ In Figure an image of this process So

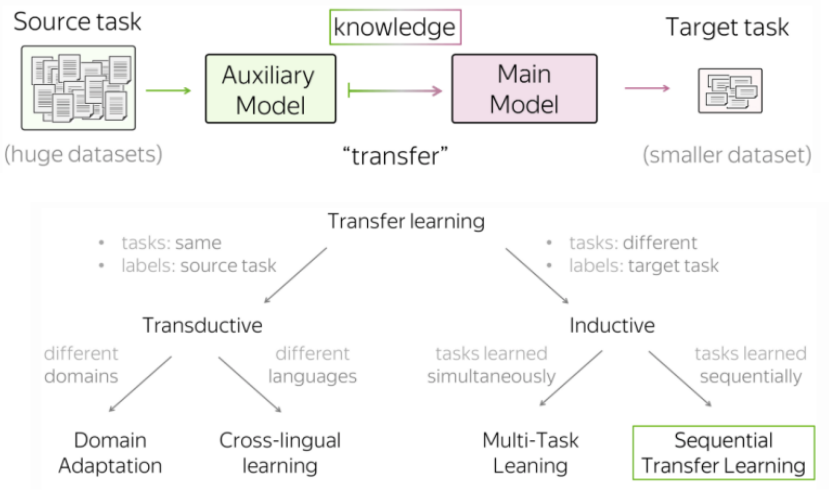


W it's frozen and does not receive gradient updates, while A and B contain trainable parameters.

5.4.1 Transfer learning

The pre-train/fine-tune paradigm is a special caase of a machine learning approach called **transfer learning**. Transfer learning is a technique in machine learning where a model developed for a certain task is reused as the starting point for a model on a second task. It's a popular approach in deep learning because it can significantly reduce the time and resources required to develop and train a model.
Transfer learning leverages the knowledge (such as weights and features) a model has learned from one task to perform another related task, instead of

starting the learning process from scratch. This method is especially useful when the first task has a large amount of available data, and the second task has relatively less data, so we have not enough data for the target task.
In Figure an idea of that, and, as we can see, there are several types of transfer learning.



Fine tuning is also known as a **sequential transfer learning**, and it's the most popular approach in NLP tasks.
Transfer learning is not the only alternative: we can handle a wide range of tasks with only a few examples, by formulating the original task as a word prediction problem, and not by updating the parameters in the underlying model. **Prompt learning**, particularly prominent in the field of NLP with the advent of LLMs like GPT-3, is a method of guiding a pre-trained model to perform specific tasks by framing the tasks as **"prompts"**.
A prompt is a piece of text inserted in the input examples, so that the original task can be formulated as a MLM problem.
This technique does not necessarily involve explicit retraining or fine-tuning of the model's weights. Instead, the task is formulated in such a way that the model understands it as part of its natural language processing capabilities. The model then generates responses based on the prompts, utilizing its pre-trained knowledge.
In Figure an example of this concept
Prompt learning is also referred to as in-context learning (capability of a model to understand and generate responses based on the context provided within the input itself, without any additional external training or fine-tuning). In the case

of prompt learning, we give with the prompt a few example to demonstrate a task and at the end our request: the model will be able to answer without using fine-tuning or adapters.
The already mentioned zero-shot/few-shot learning can also be viewed as special cases of prompt learning. In Figure an example
The advantage of prompt learning is that, given a suite of appropriate prompts, a single LM trained in an entirely unsupervised fashion can be used to solve several tasks. This method introduces the necessity for prompt engineering: finding the most appropriate prompt to allow a LM to solve the task at hand.
Prompt learning offers several advantages over traditional fine-tuning in natural language processing tasks, particularly when adapting pre-trained models to new downstream tasks. These advantages become especially apparent in certain challenging scenarios:

1. **Significant Gap Between Pre-training and Downstream Tasks:** The objectives during the pre-training stage often differ markedly from those in downstream tasks. While pre-training focuses on general language understanding or generation, downstream tasks may require specific, nuanced capabilities not directly covered during pre-training.

2. **Introduction of New Parameters:** For downstream tasks, it might be necessary to introduce new parameters to adapt the pre-trained model to the specifics of the task. This can complicate the training process, especially when the downstream task is substantially different from the tasks the model was originally trained on.

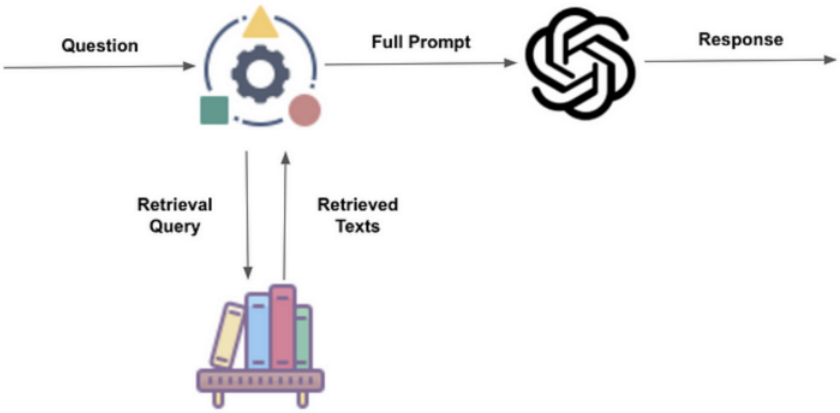
3. **Limited Availability of Training Examples:** When only a few training examples are available for a new downstream task, fine-tuning a large pre-trained model becomes challenging. The risk of overfitting increases, and the model may not generalize well to unseen data.

4. **Mitigating Model Hallucinations:** Prompting can be utilized to reduce the likelihood of model hallucinations. By providing detailed and constrained prompts, the model's outputs can be guided more accurately, enhancing the relevance and accuracy of the generated text.

5.4.2 RAG

Retrieval-augmented generation (RAG) is a two-step process combining external knowledge search with prompting.
Retrieval: In this first step, the system is given a user query or some form of

input question. An information retrieval neural model, which can be thought of as a sophisticated search engine, is used to search through a large database or corpus of documents.
The aim is to find documents or passages that are relevant to the given query.
This model is typically trained to understand the semantics of the query and to fetch content that has the highest relevance to the input, ensuring that the subsequent generation step has access to the most pertinent information. .
Generation: Once the relevant documents or passages are retrieved, they are used to construct a comprehensive prompt.
This prompt not only includes the original query but also integrates the retrieved information, effectively providing the LLM with a rich context to work from. The LLM then uses this enhanced prompt to generate a response.
The generation step is where the model synthesizes the information from the retrieved documents with its pre-existing knowledge (gained during its training phase) to produce a coherent, relevant, and often more informed response than it could have generated without access to the external documents.
In Figure an idea of that: Three important advantages of RAG approach:



- **Flexibility**: external knowledge source injects into LLM recent/specific information that wasn't available during pre-training.
- **Efficiency**: RAG works well also with small-size, more manageable LLMs.
- **Updatability**: external knowledge source can be updated, no need to retrain the LLM.

6

Part of Speech Tagging

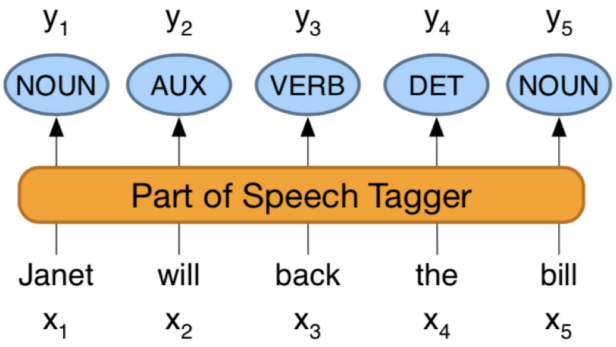
Part-of-speech (POS) tags are lexical categories such as noun, verb, adjective, adverb, pronoun, preposition, article, etc. Also known as word classes or morphological classes. Recall these classes are defined either distributionally or else functionally (see essentials of linguistics lecture).
We call a *tagset* the set of all POS tags used by some model.
Different languages, different grammatical theories, and different applications may require different tagsets. POS tags fall into two broad categories:

- **Closed tags**: preppositions, prnouns, articles. New words are rarely labeled in this class
- **Open class**: four major groups, nouns, verbs, adjectives and adverbs. New words appear almost always in this class.

The UD (Universal Dependencies) tagset contains the tagset shown in Figure:

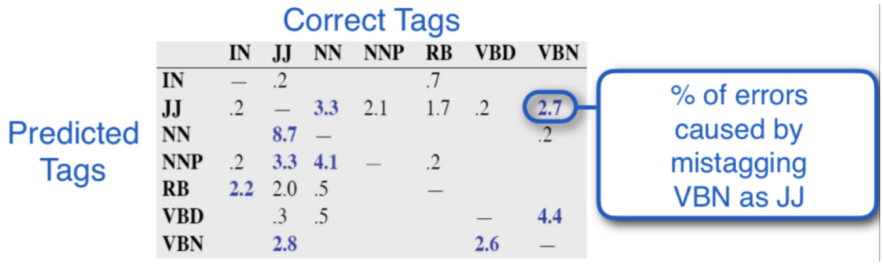
	Tag	Description	Example
Open Class	**ADJ**	Adjective: noun modifiers describing properties	*red, young, awesome*
	ADV	Adverb: verb modifiers of time, place, manner	*very, slowly, home, yesterday*
	NOUN	words for persons, places, things, etc.	*algorithm, cat, mango, beauty*
	VERB	words for actions and processes	*draw, provide, go*
	PROPN	Proper noun: name of a person, organization, place, etc..	*Regina, IBM, Colorado*
	INTJ	Interjection: exclamation, greeting, yes/no response, etc.	*oh, um, yes, hello*
Closed Class Words	**ADP**	Adposition (Preposition/Postposition): marks a noun's spacial, temporal, or other relation	*in, on, by under*
	AUX	Auxiliary: helping verb marking tense, aspect, mood, etc.,	*can, may, should, are*
	CCONJ	Coordinating Conjunction: joins two phrases/clauses	*and, or, but*
	DET	Determiner: marks noun phrase properties	*a, an, the, this*
	NUM	Numeral	*one, two, first, second*
	PART	Particle: a preposition-like form used together with a verb	*up, down, on, off, in, out, at, by*
	PRON	Pronoun: a shorthand for referring to an entity or event	*she, who, I, others*
	SCONJ	Subordinating Conjunction: joins a main clause with a subordinate clause such as a sentential complement	*that, which*
Other	**PUNCT**	Punctuation	; , ()
	SYM	Symbols like $ or emoji	$, %
	X	Other	asdf, qwfg

Words are **ambiguous** since they depend on the context in which they appear: same word could has different tags.
The task of **POS tagging** consists in the assignment of the **proper unique POS tag** to each word in an input sentence. POS tagging must be done in the context of an entire sentence, on the basis of the grammatical relationship of a word with its neighboring words.
As shown in Figure, the input is a sequence $x_1, x_2, ..., x_n$ of tokenized words and the output is a sequence of $y_1, y_2, ..., y_n$ of tags, assuming the possible tags (tagset) fixed.



POS tagging, from the moment that provides a sequence of tags as output, is a **structured prediction task** (could involve dependency tree, components of the output structure are not independent; the prediction of one part can depend on others) and not a classification task.
For example, if the first POS tag is a Noun, this causes change in other tags.
The number of structures in output can be **exponentially large** in length of input.
Evaluation: The accuracy of a POS tagger is the percentage of test set tags that match with human provided. We can use:

- **Human Ceiling:** The highest accuracy that human annotators can achieve on a given task. It represents the limit of performance based on human judgment and knowledge. It's around 97%
- **MFC Baseline (Most Frequent Class Baseline):** A simple model that assigns every instance in a dataset to the most frequently occurring class observed in the training set. It serves as a basic benchmark for evaluating the performance of more sophisticated models. It's around 92%
- **Qualitative evaluation**: generate a confusion matrix which represent a record of how often a word with a real tag y_i was mistagged as y_j, as shown in Figure



	IN	JJ	NN	NNP	RB	VBD	VBN
IN	—	.2			.7		
JJ	.2	—	3.3	2.1	1.7	.2	2.7
NN		8.7	—				.2
NNP	.2	3.3	4.1	—	.2		
RB	2.2	2.0	.5		—		
VBD		.3	.5			—	4.4
VBN		2.8				2.6	—

6.1 Hidden Markov model

HMM is a **generative model**: it models how a class could generate some input data. You might use the model to generate examples.
Contrast with discriminative models, discussed later, which only learn to distinguish classes, without learning much about them.

Let $w_{1:n} = w_1, w_2, \ldots, w_n$ be an input sequence of words, and let $\mathcal{Y}(w_{1:n})$ be the set of all possible tag sequences $t_{1:n} = t_1, t_2, \ldots, t_n$ for $w_{1:n}$.

The goal of POS tagging is to choose the most probable tag sequence $\hat{t}_{1:n} \in \mathcal{Y}(w_{1:n})$

$$\hat{t}_{1:n} = \arg \max_{t_{1:n} \in \mathcal{Y}(w_{1:n})} P(t_{1:n}|w_{1:n})$$

This is the typical formulation for a structured prediction problem. Note that $\mathcal{Y}(w_{1:n})$ is not the set of all strings of length n over the tagset, but all the possible sequences over the tagset..

Term $P(t_{1:n}|w_{1:n})$ is referred to as the **posterior** probability and can be difficult to model/compute, so we need to elaborate it in another way.

We break down the posterior probability using **Bayes' rule**:

$$\hat{t}_{1:n} = _{t_{1:n} \in \mathcal{Y}(w_{1:n})} P(t_{1:n}|w_{1:n})$$

$$= _{t_{1:n} \in \mathcal{Y}(w_{1:n})} \frac{P(w_{1:n}|t_{1:n}) \cdot P(t_{1:n})}{P(w_{1:n})}$$

$$= _{t_{1:n} \in \mathcal{Y}(w_{1:n})} P(w_{1:n}|t_{1:n}) \cdot P(t_{1:n})$$

where we have used the fact that $w_{1:n}$ is given, so $P(w_{1:n})$ is a constant in all

terms and so we could remove it.

The term $P(t_{1:n})$ is referred to as the **prior or marginal** probability. The term $P(w_{1:n}|t_{1:n})$ is the **likelihood** of the words given the tags.

HMM models $P(w_{1:n}|t_{1:n}) \cdot P(t_{1:n})$, which equals the **joint probability** $P(t_{1:n}, w_{1:n})$ and so we can write:

$$= t_{1:n} \in \mathcal{Y}(w_{1:n}) \; P(t_{1:n}, w_{1:n})$$

HMM POS taggers **make two simplifying assumptions**.

- The first is that the probability of a word depends only on its own POS tag and is independent of neighboring words and tags:

  $$P(w_{1:n}|t_{1:n}) \approx \prod_{i=1}^{n} P(w_i|t_i)$$

  The factor $P(w_i|t_i)$ is referred to as the **emission probability**.
  The emission probability answers the following questions: If we were going to generate t_i, how likely is it that the associated word would be w_i?

- The second assumption is the **Markov assumption** that the probability of a tag is dependent only on the previous tag, rather than the entire tag sequence

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

  The factor $P(t_i|t_{i-1})$ is referred to as the **transition probability**.

Putting everything together we have:

$$\hat{t}_{1:n} = \arg \max_{t_{1:n} \in \mathcal{Y}(w_{1:n})} P(w_{1:n}|t_{1:n}) \cdot P(t_{1:n})$$

$$\approx \arg \max_{t_{1:n} \in \mathcal{Y}(w_{1:n})} \left(\prod_{i=1}^{n} P(w_i|t_i) \cdot P(t_i|t_{i-1}) \right)$$

6.1.1 Probability estimation

Assume a **tagged corpus**, where each word has been tagged with its true label. We implement **supervised** learning using the relative frequency estimator.

For the transition probability we obtain:

$$P(t_i \mid t_{i-1}) = \frac{C(t_{i-1} t_i)}{C(t_{i-1})}$$

Similarly, for the emission probability we obtain:

$$P(w_i \mid t_i) = \frac{C(t_i, w_i)}{C(t_i)}$$

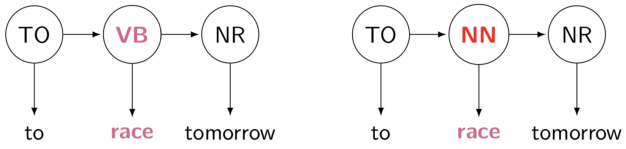
In Figure some examples related to these concepts

Example :

He/PPS is/VBZ expected/VBN to/TO race/VB tomorrow/NR

Why is VB more likely than NN for token race in sentence above?

There are at least two sequences of tags which we could consider:



The two emission probabilities turn out not to differ too much:

$$P(\text{race} \mid \text{VB}) = 0.00012$$
$$P(\text{race} \mid \text{NN}) = 0.00057$$

Nor is there much difference on whether NR follows NN or VB:

$$P(\text{NR} \mid \text{VB}) = 0.0027$$
$$P(\text{NR} \mid \text{NN}) = 0.0012$$

However, the big difference is in whether NN or VB follows TO:

$$P(\text{VB} \mid \text{TO}) = 0.83$$
$$P(\text{NN} \mid \text{TO}) = 0.00047$$

6.1.2 HMMs as automata

We can formally define an HMM as a special type of **probabilistic finite state automaton** which generates sentences instead of accepting.
States represent hidden information, POS tags which are not observed. The transition function is defined according to transition probabilities, between one state to another. At the end, each state **generates** a word (according to emission probabilities) so, the generated output, is a **word sequence**.
HMM definition:

- finite set of output symbols V (vocabulary, each state has the emission probability for each word of V)
- finite set of states Q, with initial state q_0 that is the start marker and final state q_f that is the end marker. Other states are possible tags

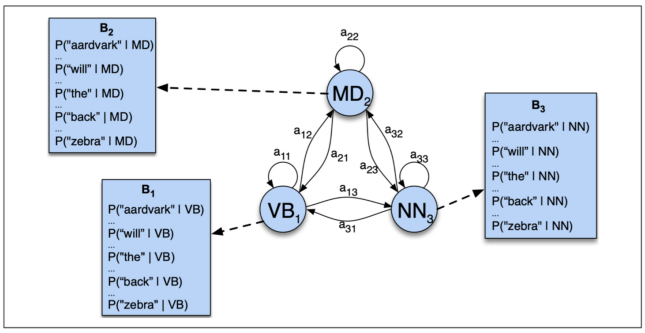
- transition probabilities $a_{q,q'}$ for each pair $q, q', q \in Q \setminus \{q_f\}, q' \in Q \setminus \{q_0\}$. So it's the probability of moving from a tag q to a tag q'
- emission probabilities $b_q(u)$ for each pair $q, u, q \in Q \setminus \{q_0, q_f\}, u \in V$. So it's the probability of having u being the tag q

Transition and emission probabilities are subject to:

- $\sum_{q'} a_{q,q'} = 1$ for all $q \in Q \setminus \{q_f\}$. So, given a tag q, if we sum all possible transition probabilities (going from q to other tags q'), the sum is 1
- $\sum_u b_q(u) = 1$ for all $q \in Q \setminus \{q_0, q_f\}$. The sum of all emission probabilities (probability of going from q to a word u) from one tag is 1.

Probabilities $a_{q_0,q}$ define the so-called **initial** probability distribution, sometimes also denoted as $\pi(q)$.

Probabilities a_{q,q_f} are the **stop** probabilities, not used in the textbook.



6.1.3 Viterbi algorithm: supervised learning

The first algorithm that we'll see is Viterbi algorithm, which uses annotated corpus in order to compute the POS tagging task to a sentence $w_{1:n}$ **Decoding problem for HMMs:** Given a sequence of observations $w_{1:n}$, find the most probable sequence of states/tags $\hat{t}_{1:n}$

$$\hat{t}_{1:n} = \underset{t_{1:n} \in \mathcal{Y}(w_{1:n})}{\operatorname{argmax}} P(t_{1:n} \mid w_{1:n})$$

$$= \underset{t_{1:n} \in \mathcal{Y}(w_{1:n})}{\operatorname{argmax}} \left(\prod_{i=1}^{n} P(w_i \mid t_i) \cdot P(t_i \mid t_{i-1}) \right)$$

Also called **inference problem**.

In order to compute $\hat{t}_{1:n}$ we can use the following **naive algorithm**

- enumerate all sequences (paths) of POS tags $t_{1:n}$ consistent with observed sentence $w_{1:n}$
- perform a max search over all the joint probabilities $P(t_{1:n}, w_{1:n})$

This algorithm requires **exponential time**, since there can be exponentially many $t_{1:n}$ sequences in $\mathcal{Y}(w_{1:n})$!

This is a very common scenario for structured prediction problems, since there are many branches..

The classical decoding algorithm for HMMs is the **Viterbi algorithm**, an instance of dynamic programming, so breaking in smaller parts our problem and then merge it.

The Viterbi algorithm computes the optimal sequence $\hat{t}_{1:n}$ and the associated joint probability $P(\hat{t}_{1:n}, w_{1:n})$ in **polynomial time**.

Let $w_{1:n} = w_1, w_2, \ldots, w_n$ be the input sequence.

In what follows

- q denotes a state/tag of the HMM
- i denotes an input position, $0 \leq i \leq n + 1$

Input positions 0 and $n + 1$ represent start and end markers, respectively.

We use a two-dimensional table $vt[q, i]$ denoting the probability of the **best path** to get to state q after scanning $w_{1:i}$. In other words, for each state q and each time step i, the Viterbi table $vt[q, i]$ holds the probability of the most likely path that the system could have taken through the state space to reach state q given the observations seen so far.

We use a two-dimensional table $bkpt[q, i]$ for retrieving the best path.

Initialisation step: for all q

- $vt[q, 1] = a_{q_0,q} \cdot b_q(w_1)$. So it is the probability that we move from the start marker to state q and the emitted word is w_1 (we know that the first word is that).
- $bkpt[q, 1] = q_0$

Recursive step: at each iteration i, for all q

- $vt[q, i] = \max_{q'} vt[q', i-1] \cdot a_{q',q} \cdot b_q(w_i)$. So I take the probability of the path that maximizes over all possible q' the path that brings to q, considering also the emission probability of word i from q.
- $bkpt[q, i] =_{q'} vt[q', i-1] \cdot a_{q',q} \cdot b_q(w_i)$. So here it's stored the state q' that brings to q maximizing the probabilities.

Termination step:

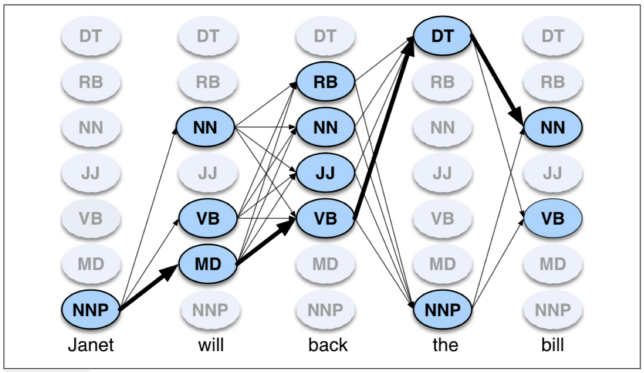
- $vt[q_f, n+1] = \max_{q'} vt[q', n] \cdot a_{q',q_f}$
- $bkpt[q_f, n+1] =_{q'} vt[q', n] \cdot a_{q',q_f}$

After execution of the algorithm we have

$$vt[q_f, n+1] = P(\hat{t}_{1:n}, w_{1:n})$$

where $\hat{t}_{1:n} = q_1, \ldots, q_n$ is the **most likely sequence** of tags for $w_{1:n}$.

The sequence of tags $\hat{t}_{1:n}$ can be **reconstructed** starting with $bkpt[q_f, n+1]$ and following the backpointers. So the first backpointer is the state q_n that has maximized the probability of arriving in q_f. Then we do the same fot q_n, until we arrive in q_0.



6.2 Forward-Backward algorithm: unsupervised learning

Until now we have talked about algorithm based on the fact that emission and transition probabilties are known, so, giving a new sequence of word, it outputs a tag sequence based on this.

It's different if we don't have a tagged corpus and we want to do the same task of POS tagging.

We need to train a model that estimates these probabilities and after could outputs tag sequences.

With the goal of developing an **unsupervised** algorithm (so we don't have

the correct tags) for the estimation of HMM, we now develop some auxiliary concepts and algorithm.

Consider the probability of the sequence $w_{1:n}$, defined as

$$P(w_{1:n}) = \sum_{t_{1:n} \in \mathcal{Y}(w_{1:n})} P(t_{1:n}, w_{1:n})$$

$$= \sum_{t_{1:n} \in \mathcal{Y}(w_{1:n})} \left(\prod_{i=1}^{n} P(w_i \mid t_i) \cdot P(t_i \mid t_{i-1}) \right)$$

This is also called the likelihood of the sequence $w_{1:n}$. So it's the sum over all possible sequences $t_{1:n}$ of getting $w_{1:n}$

6.2.1 Forward algorithm

The **forward algorithm** computes $P(w_{1:n})$ in **polynomial time**, exploiting dynamic programming.

The forward algorithm is very similar to Viterbi's algorithm, but *using summation instead of maximisation*.

No use of backpointers, since we do not need to retrieve an optimal sequence.

Let $w_{1:n} = w_1, w_2, \ldots, w_n$ be the input sequence.

We use a two-dimensional table $\alpha[q, i]$ denoting the sum of probabilities of **all paths** that reach state q after scanning $w_{1:i}$, so the probability that we are in q after scanning the first i words.

Formally, this is the joint probability of $w_{1:i}$ and state q at i, and can be written as

$$\alpha[q, i] = \sum_{t_{1:i} \in \mathcal{Y}(W_{1:i})} P(t_{1:i}, W_{1:i}) \quad \text{s.t. } t_i = q$$

Each of the above quantities is called **forward probability**.

Initialisation step: for all q

- $\alpha[q, 1] = a_{q_0,q} \cdot b_q(w_1)$

Recursive step: at each iteration i and for all q

- $\alpha[q, i] = \sum_{q'} \alpha[q', i-1] \cdot a_{q',q} \cdot b_q(w_i)$

Termination step:

- $\alpha[q_f, n+1] = \sum_{q'} \alpha[q', n] \cdot a_{q',q_f}$

After execution of the algorithm we have

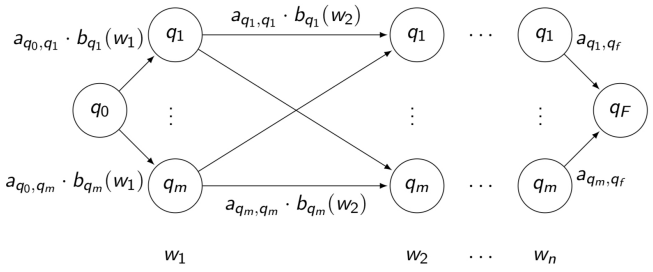
$$\alpha[q_f, n+1] = P(w_{1:n})$$

An intuitive interpretation of the forward algorithm is in terms of expansion of the HMM into a **trellis**, defined as follows.

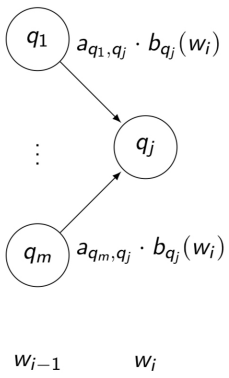
Given input $w_{1:n} = w_1, w_2, \ldots, w_n$

- introduce special nodes for q_0 and q_f
- for each token w_i and for each state $q \neq q_0, q_f$, introduce a node with label q
- for each pair of nodes q, q' associated with tokens w_{i-1}, w_i, respectively, introduce an arc with weight provided by the product $a_{q,q'} \cdot b_{q'}(w_i)$
- introduce special arcs for node q_f

Each path through the trellis corresponds to a sequence $t_{1:n}$ of states consistent with input $w_{1:n}$. In Figure we can see one trellis



In the forward algorithm we compute one value for each node, as shown in Figure



$$\begin{aligned}\alpha[q_j, i] \;=\; & \\ & \alpha[q_1, i-1] \cdot a_{q_1,q_j} \cdot b_{q_j}(w_i) + \\ & \cdots + \\ & \alpha[q_m, i-1] \cdot a_{q_m,q_j} \cdot b_{q_j}(w_i)\end{aligned}$$

6.2.2 Backward algorithm

The **backward algorithm** uses a two-dimensional table $\beta[q, i]$ denoting the sum of probabilities of **all paths** that start at state q, scan sequence $w_{(i+1):n}$, and reach state q_f, so the probability of being in q at iteration i and going to q_f at last iteration

Formally this is the **joint probability** of $w_{(i+1):n}$ and state q at i, and can be expressed as

$$\beta[q, i] = \sum_{t_{(i+1):n} \in \mathcal{Y}(w_{(i+1):n})} P(t_{(i+1):n}, w_{(i+1):n})$$

s.t. $t_i = q$

Each of the above quantities is called **backward probability**.

Initialisation step: for all q:

- $$\beta[q, n] = a_{q,q_f}$$

Recursive step: for all iterations from $i = n-1, \ldots, 1$ and for all q:

- $$\beta[q, i] = \sum_{q'} \beta[q', i+1] \cdot a_{q,q'} \cdot b_{q'}(w_{i+1})$$

Termination step:

- $$\beta[q_0, 0] = \sum_{q'} \beta[q', 1] \cdot a_{q_0,q'} \cdot b_{q'}(W_1)$$

After execution of the algorithm we have

$$\beta[q_0, 0] = \alpha[q_f, n+1] = P(w_{1:n})$$

We observe that the backward probability is the **dual** of the forward probability. More precisely, we have

$$\alpha[q, i] \cdot \beta[q, i] = \sum_{t_{1:n} \in \mathcal{Y}(w_{1:n})} P(t_{1:n}, w_{1:n})$$

s.t. $t_i = q$

In words, this is the probability of all paths in the HMM trellis for $w_{1:n}$ that go through state q at position i. That's why $\alpha[q, i]$ is the probability of all possible

paths reaching the tag q after have scanned the first i words and $\beta[q, i]$ is the probability of starting from q and end in q_f after scanning from i to n words.

A relation similar to the above plays an important role in our unsupervised learning algorithm.

6.2.3 Baum-Welch algorithm

Suppose we are given an unannotated corpus and a tagset. Now we cannot count transitions and emissions directly, because we don't know which path through the HMM is the right one.
Can we train HMM with tags as internal states? This is possible using the **forward-backward algorithm**.

To keep discussion simple, assume we have a single unannotated sentence $w_{1:n}$ to train the model.

Let vector θ be an assignment for all the parameters $a_{q,q'}$ and $b_q(w_i)$ of the HMM.

We write $P_\theta(t_{1:n}, w_{1:n})$ to denote the joint distribution for $t_{1:n}$ and $w_{1:n}$, $t_{1:n} \in \mathcal{Y}(w_{1:n})$, based on θ.

All of the expectations defined below are computed under distribution $P_\theta(t_{1:n}|w_{1:n})$, over all paths in $\mathcal{Y}(w_{1:n})$.

$c(q, q')$ is the expectation of the transition (q, q'), the expected number of transitions from state q to q

$c(q, u)$ is the expectation of the emission of symbol $u \in V$ at state q, the expected number of times state q emits symbol u.

$c(q)$ is the expectation of each state q.

The forward-backward algorithm is an iterative algorithm for the unsupervised learning of parameter vector θ.

The algorithm starts with some initial assignment θ, and updates θ by iterating the following steps

- E-step (expectation) computes feature expectations $c(q, q')$, $c(q, u)$ and $c(q)$, on the basis of P_θ
- M-step (maximization) estimates a new assignment $\hat{\theta}$, in a way that maximizes the log-likelihood of the training data

The algorithm stops when the parameters do not change much anymore.

We show how to compute expectations $c(q, q')$, $c(q, u)$ and $c(q)$.

The sum of probabilities of all paths $t_{1:n} \in \mathcal{Y}(w_{1:n})$ that go through transition (q, q') at input position i is

$$\alpha[q, i] \cdot a_{q,q'} \cdot b_q(w_{i+1}) \cdot \beta[q', i+1]$$

That is the forward probability of get at step i the tag q, multiplied by the probability of moving from q to q' at iteration $i+1$, multiplied by the emission probability of word w_{i+1} from q', multiplied by the backward probability of all paths starting from q' at iteration $i+1$ and reaching q_f.

Dividing by the sum of probabilities of all paths, we obtain the probability of (q, q') at position i given $w_{1:n}$

$$c(q, q', i) = \frac{\alpha[q, i] \cdot a_{q,q'} \cdot b_q(w_{i+1}) \cdot \beta[q', i+1]}{\sum_{t_{1:n} \in \mathcal{Y}(w_{1:n})} P(t_{1:n}, w_{1:n})} = \frac{P(w_{1:n})}{}$$

Summing up for all positions i in $w_{1:n}$ we get

$$c(q, q') = \sum_i c(q, q', i)$$

The sum of probabilities of all paths $t_{1:n} \in \mathcal{Y}(w_{1:n})$ that go through transition q at input position i while emitting w_i is

$$\alpha[q, i] \cdot \beta[q, i]$$

Dividing by the sum of probabilities of all paths, we obtain the probability of emitting w_i at state q, given $w_{1:n}$

$$c(q, w_i, i) = \frac{\alpha[q, i] \cdot \beta[q, i]}{P(W_{1:n})}$$

Summing up for all positions i with emission $u \in V$ we get

$$c(q, u) = \sum_{i:w_i=u} c(q, u, i)$$

We have that

- an occurrence of a state must be followed by a transition
- an occurrence of a state must be associated with an emission

Then it is not difficult to see that

$$c(q) = \sum_{q'} c(q, q') = \sum_{u} c(q, u)$$

We estimate a new parameter assignment $\hat{\theta}$ based on expectations $c(q, q')$, $c(q, u)$ and $c(q)$.

$$\hat{a}_{q,q'} = \frac{c(q, q')}{c(q)}$$

$$\hat{b}_q(u) = \frac{c(q, u)}{c(q)}$$

Very similar to the relative frequency estimator, but we now use feature expectations in place of counts.

We now have a *refined* probability distribution $\hat{P}_{\hat{\theta}}(t_{1:n}, w_{1:n})$.

The forward-backward algorithm generally converges to some *local optimum*, with a (relative) maximum for the likelihood of training data.

Starting with different initial guesses for parameter vector θ may lead to different solutions (different local optima).

No effective algorithm is known to compute *global optimum*, maximising likelihood of unannotated training material.

The forward-backward algorithm is an instance of a more general class of algorithms called *EM* (expectation-maximisation). In general, it is hard for generative models like HMMs to add rich features directly in a clean way.
We had some limitations: **Independence assumptions** for the model features are quite strong: when multiplying model features we assume independence, if this does not hold sequence probabilities do not sum up to one.
Inconsistency between local training and global testing: probabilities of each label are trained locally, but output is the highest probability sequence, which is searched globally.

6.3 Conditional random fields

Conditional random fields (CRF) are discriminative sequence models based on log-linear models. Discriminative classifiers learn what features from the input are most useful to discriminate between the different possible classes.
Generative models are interested in modeling the joint probability distribution

of the input data and output labels. This is represented by $P(X, Y)$ where X represents the input data and Y represents the output labels. By understanding how data is generated ($P(X|Y)$ and $P(Y)$), these models can generate new data instances and provide a comprehensive picture of the data landscape.
Discriminative models focus directly on the decision boundary between different classes. They model the conditional probability $P(Y|X)$ where the goal is to determine the most probable label Y given the input X. These models are optimized for prediction, making them more effective in classification tasks where the primary goal is accurate labeling rather than understanding data generation.

The model can only distinguish the classes, perhaps without learning much about them: it is unable to generate observations/examples.

We describe the *linear chain CRF*, the version of CRF that is most commonly used for language processing, and the one whose conditioning closely matches HMM.

Let $x_{1:n} = x_1, x_2, \ldots, x_n$ be the input word sequence, and let $\mathcal{Y}(x_{1:n})$ be the set of all possible tag sequences $y_{1:n} = y_1, y_2, \ldots$ for $x_{1:n}$.

CRFs solve the problem:

$$\hat{y}_{1:n} = \underset{y_{1:n} \in \mathcal{Y}(x_{1:n})}{\arg\max}\ P(y_{1:n}|x_{1:n})$$

In contrast with HMMs, we directly model the posterior probability $P(y_{1:n}|x_{1:n})$.

In contrast to HMM, the CRF does not compute a probability for each tag at each time step.

Instead, at each time step the CRF computes log-linear functions over a set of relevant local features, and these features are aggregated and normalised to produce a global probability and in this way we do not need the independence assumption of HMM.

Let us assume we have K global feature functions $F_k(x_{1:n}, y_{1:n})$, and weights w_k for each feature.

We define

$$P(y_{1:n}|x_{1:n}) = \frac{\exp\left(\sum_{k=1}^{K} w_k F_k(x_{1:n}, y_{1:n})\right)}{\sum_{y'_{1:n} \in \mathcal{Y}(x_{1:n})} \exp\left(\sum_{k=1}^{K} w_k F_k(x_{1:n}, y'_{1:n})\right)}$$

where the numerator is the weighted sum of **global feature** for the single

$x_{1:n}, y_{1:n}$ while the denominator is the weighted sum of global feature but computed on all possible output label sequence.

The denominator is the so-called **partition function** $Z(x_{1:n})$, a normalization term that only depends on the input sequence $x_{1:n}$.

$$Z(x_{1:n}) = \sum_{y'_{1:n} \in \mathcal{Y}(x_{1:n})} \exp\left(\sum_{k=1}^{K} w_k F_k(x_{1:n}, y'_{1:n})\right)$$

We compute the global features by decomposing into a sum of **local features**, for each position i in $y_{1:n}$

$$F_k(x_{1:n}, y_{1:n}) = \sum_{i=1}^{n} f_k(y_{i-1}, y_i, x_{1:n}, i)$$

Practical assumption: Each local feature depends on the current and previous output tokens, y_i and y_{i-1} respectively.

The specific constraint above characterizes linear chain CRF. This limitation makes it possible to use the Viterbi algorithm.

Local features: For a **predicate** x, we write $I(x)$ to denote 1 if it is true, 0 otherwise. In Figure some examples of features in linear-chain CRF.

What features to use is a decision of the system designer but in order to avoid handwriting, specific features are automatically instantiated from **feature templates**.

Using information from y_{i-1}, y_i and $x_{1:n}$ some templates could be:

$$< y_i, x_i >, < y_i, y_{i-1} >, < y_i, x_{i-1}, x_{i-2}$$

and an example is shown in Figure

Example : Using dataset

Janet/NNP will/MD back/VB the/DT bill/NN

when x_i is the word **back** the following features would be generated (indices generated at random)

- $f_{3743} : \mathbb{I}\{x_i = \text{back}, y_i = \text{VB}\}$
- $f_{156} : \mathbb{I}\{y_i = \text{VB}, y_{i-1} = \text{MD}\}$
- $f_{99732} : \mathbb{I}\{y_i = \text{VB}, x_{i-1} = \text{will}, x_{i+2} = \text{bill}\}$

For better understand how are used feature templates, look at next chapter,

under section about feature extraction.

6.3.1 Inference

The inference problem for linear-chain CRF is expressed as:

$$\hat{y}_{1:n} = \arg \max_{y_{1:n} \in \mathcal{Y}(x_{1:n})} P(y_{1:n}|x_{1:n})$$

$$= \arg \max_{y_{1:n} \in \mathcal{Y}(x_{1:n})} \frac{1}{Z(x_{1:n})} \cdot \exp\left(\sum_{k=1}^{K} w_k F_k(x_{1:n}, y_{1:n})\right)$$

$$= \arg \max_{y_{1:n} \in \mathcal{Y}(x_{1:n})} \sum_{k=1}^{K} w_k F_k(x_{1:n}, y_{1:n})$$

$$= \arg \max_{y_{1:n} \in \mathcal{Y}(x_{1:n})} \sum_{i=1}^{n} \sum_{k=1}^{K} w_k f_k(y_{i-1}, y_i, x_{1:n}, i)$$

We can still use the Viterbi algorithm because the linear-chain CRF depends at each time-step on only one previous output token y_{i-1}.

Recall that for HMMs the recursive step states, for each position i and state q:

$$v_t[q, i] = \max_{q'} \left\{v_t[q', i-1] \cdot a_{q',q} \cdot b_q(w_i)\right\}$$

For CRF we need to replace the prior and the likelihood probabilities with the CRF features (where t, t' are tags):

$$v_t[t, i] = \max_{t'} \left\{v_t[t', i-1] + \sum_{k=1}^{K} w_k f_k(t', t, x_{1:n}, i)\right\}$$

6.3.2 Training

The parameters w_i in CRF can be learned in a supervised way as in the method of logistic regression.

We minimize the negative log-likelihood as our objective function. It is possible to show that this is equivalent to minimizing the expectation of the cross-entropy of all the conditional probability distributions.

As in the case of multinomial logistic regression, $L1$ or $L2$ regularization is important.

To optimize the objective function, we use stochastic gradient descent.

Let $D = \{(y_{1:n}^{(h)}, x_{1:n}^{(h)}) | 1 \leq h \leq N\}$ be a training set, where each $x_{1:n}^{(h)}$ is a sentence and each $y_{1:n}^{(h)}$ is the associated sequence label.

Let also $\mathbf{w}$ be the parameter vector.

The objective function is:

$$\mathcal{L}(D, \mathbf{w}) = \frac{\lambda}{2}\|\mathbf{w}\|^2 - \sum_{h=1}^{N} \log P(y_{1:n}^{(h)} | x_{1:n}^{(h)})$$

$$= \frac{\lambda}{2}\|\mathbf{w}\|^2 - \sum_{h=1}^{N} \log \frac{1}{Z(x_{1:n}^{(h)})} \exp\left(\sum_{k=1}^{K} w_k F_k(y_{1:n}^{(h)}, x_{1:n}^{(h)})\right)$$

$$= \frac{\lambda}{2}\|\mathbf{w}\|^2 - \sum_{h=1}^{N} \left(\sum_{k=1}^{K} w_k F_k(y_{1:n}^{(h)}, x_{1:n}^{(h)}) + \log Z(x_{1:n}^{(h)})\right)$$

In order to compute the gradient of $\mathcal{L}(D, \mathbf{w})$ we have to be able to efficiently compute feature expectations:

$$\sum_{y_{1:n} \in \mathcal{Y}(x_{1:n})} P(y_{1:n} | x_{1:n}) \cdot F_k(x_{1:n}, y_{1:n})$$

In practice, feature expectations are computed "under the hood" by modern software libraries, such as PyTorch, using automatic differentiation of the objective function.

Alternatively, feature expectations can be efficiently computed using the forward-backward algorithm, which we have already seen for unsupervised learning with HMMs.

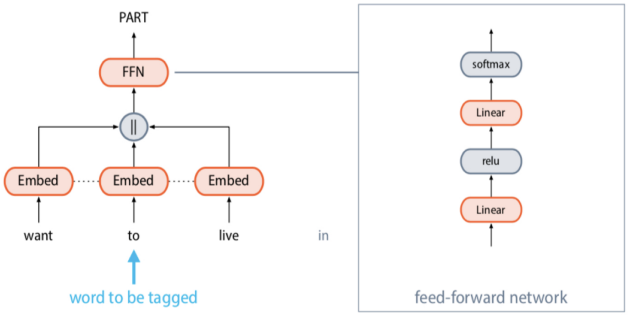
6.4 Neural POS tagger

In neural network approaches to POS tagging, we construct distributed feature representations (dense vectors) for each tagging decision, based on the word and its context.

There are two main approaches:

Local search: Neural networks can perform POS tagging as a per-token multi-

classification decision.
For doing POS tagging we can use **a fixed window neural network** which architecture is given in Figure



There is the initial embedding, which could be provided or trained. Then, embeddings are concatenated and given as input of a FFN which firstly rearrange in a dimension d the features, then a non-linear function is applied and then, by using a linear function, it's rearranged in a way such that for each cell, there is a tag. At the end it's applied a softmax function which returns output probabilities.
The fixed-window model is very efficient, since it limits the context from which information can be extracted. However,it has limitations in capturing the full linguistic structure and dependencies that might be important for understanding the full meaning of a sentence.
An alternative could be the **recurrent neural model**: Let $x_1, x_2, \ldots, x_n$ be the input word sequence and $y_1, y_2, \ldots, y_n$ be the associated output POS tags.

We assume word embeddings $e_1, e_2, \ldots, e_n$ for $x_1, x_2, \ldots, x_n$.

Recurrent neural networks can be generally described as implementing the following recursive relation, for $t = 1, \ldots, n$:

$$h_t = f(g(e_t, h_{t-1}))$$
$$= f(W^h h_{t-1} + W^e e_t + b)$$

with f some nonlinear component. W^h, W^e, b are learnable parameters. Gated RNNs such as LSTM networks implement more complex recurrence relations. We score each POS tag y by means of a **linear scalar function** of hidden state vector h_t, and then retrieve the highest score tag for x_t:

$$\psi(y, h_t) = \beta_y \cdot h_t$$

where β_y is a learnable parameter

$$\hat{y}_t = \arg\max_y \psi(y, h_t)$$

The score $\psi(y, h_t)$ can also be converted into a probability using the softmax operation:

$$P(y|x_1 : t) = \frac{\exp \psi(y, h_t)}{\sum_{y'} \exp \psi(y', h_t)}$$

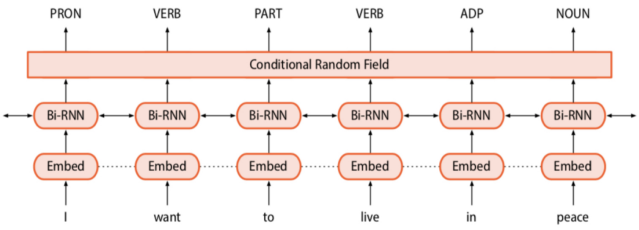
The last possibility is to use **recurrent bidirectional model**: hidden state vector h_t encodes left context up to position t but it ignores subsequent tokens, which might be relevant to y_t as well.

Bidirectional RNN are used to address this problem:

$$\overrightarrow{h_t} = g(e_t, \overrightarrow{h_{t-1}})$$
$$\overleftarrow{h_t} = g'(e_t, \overleftarrow{h_{t+1}})$$
$$h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}]$$

The scoring function $\psi(y, h_t)$ is then applied to the concatenation of the two vectors.

The model, presented in Figure, is still based on local search, each tagging decision is made independently and no global search is performed.



Global search: Neural sequence labelling can be combined with global search by augmenting local scores as:

$$\psi(y_t, y_{t-1}, h_t) = \beta_{y_t} \cdot h_t + \eta_{y_{t-1}, y_t}$$

where η_{y_{t-1},y_t} is a scalar, learnable parameter for the POS tag transition (y_{t-1}, y_t).

As in CRF, Viterbi algorithm is used for inference (joint tagging) and the forward-backward algorithm is used for training.
Global search neural model can be thought of as a combination of recurrent bidirectional model and CRF: the feature vector is extracted by the neural model and provided to the CRF algorithm, as shown in Figure

Morphological rich languages (MRL) have much more information than English coded into word morphology, like case (nominative, accusative, genitive) or gender (masculine, feminine). This information is really important for tasks like parsing and coreference resolution.

Tagsets for MRL are therefore sequences of morphological tags rather than a single primitive tag. Tagsets for these languages can be 4 to 10 times larger than English.

With such large tagsets, specialized POS taggers need to be developed where each word needs to be morphologically analyzed.

6.4.1 Sequence labelling

POS tagging is an instance of a more general problem called **sequence labelling**, assigning to an input word sequence $x_1, x_2, \ldots, x_n$ an output sequence $y_1, y_2, \ldots, y_n$ over an arbitrary set of categories.

Again, this is a structured prediction problem, and categories must be assigned contextually.

We are going to briefly overview other NLP tasks that can be cast as sequence labelling problems:

- **Named entity recognition (NER)** seeks to locate multi-word expressions referring to entities such as person names, organizations, locations, time expressions, quantities, monetary values, etc, as shown in Figure
  In Named Entity Recognition (NER), the task involves not only the *tagging* of individual words as entities but also identifying the *span* of named entities, which is the sequence of words that make up a single entity. For example, in the sentence "Elon Musk founded SpaceX in California," the span of the named entity corresponding to the person includes both "Elon" and "Musk," marking the beginning and the end of the entity respectively. Correctly identifying the span is critical as it affects the understanding of the sentence structure and the relationships between entities. The standard approach for span-recognition is **BIO tagging**. In BIO tagging:
  - tokens that begin a span are marked with label B

- tokens that occur inside a span in a position other than the leftmost one are marked with I
- tokens outside of any span of interest are marked with O

In Figure an example for a sentence

Words	IO Label	BIO Label	BIOES Label
Jane	I-PER	B-PER	B-PER
Villanueva	I-PER	I-PER	E-PER
of	O	O	O
United	I-ORG	B-ORG	B-ORG
Airlines	I-ORG	I-ORG	I-ORG
Holding	I-ORG	I-ORG	E-ORG
discussed	O	O	O
the	O	O	O
Chicago	I-LOC	B-LOC	S-LOC
route	O	O	O
.	O	O	O

Named entity recognizers are evaluated by:.

Precision is the percentage of named entities found by the learning system that are correct.

Recall is the percentage of named entities present in the corpus that are found by the system.

F1-score is the harmonic mean of the two. Precision, recall and F1-score are computed for each individual BIO label. This accounts for

- assignment of wrong entity types
- wrong entity boundaries

7 Dependency Parsing

7.1 Syntax and Phrase-Structure

In linguistics, **syntax** is the set of rules, principles, and processes that govern the structure of sentences in a given language, including word order.
We need **syntactic analysis** or syntactic parsing in order to be able to perform most NLP tasks.
Syntactic analysis is a fundamental process that involves the analysis of sentences in a given text to determine their grammatical structure. It plays a crucial role in understanding how the components of a sentence relate to one another and is vital for several reasons in the context of NLP: by understanding the structure of a sentence, NLP systems can better infer the meaning of the text.
For example, distinguishing between "The man saw the telescope" and "The man with the telescope saw the stars" requires recognizing syntactic cues that indicate whether "the telescope" is an object being seen or an instrument used for seeing.
In syntactic analysis a **constituent**, also called **phrase**, is a group of words that function as a single unit within a hierarchical structure. For example, in the sentence "The quick brown fox jumps over the lazy dog," "the quick brown fox" can be considered a constituent because it acts together as the subject of the sentence.

In Figure, an example of constituent in a sentence

Example : The sentence "He saw the house on the hill" has the following constituents (among others)

- He
- the house
- on the hill

while the following sequences are **not** constituents

- He saw the
- house on
- on the

There are several types of constituents, each characterised by the context in which they can occur their internal structure.

- **Sentence**, abbreviated S, is a constituents representing a complete proposition or clause.

  Example :
  - This is a sentence
  - Alice mentioned that Bob has been to Venice

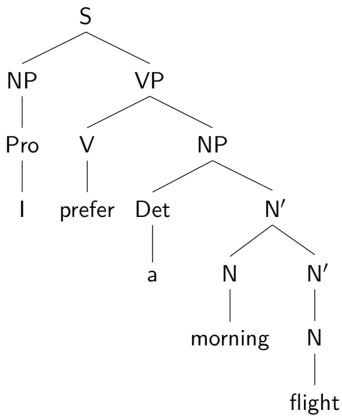
- **Noun phrase**, abbreviated NP is typically built around a common noun, proper noun, or personal pronoun and can occur just before a verb as subject, or after a verb as object

  Example :
  - I prefer the morning flight
  - My colleagues prefer the flight in the morning

- **Verb phrase**, abbreviated VP, typically starts with at least one verb and it is optionally followed by some argument, such as NP or S
- **Prepositional phrase**, abbreviated PP typically starts with preposition followed by a NP and may be omitted from a sentence
- **Adjective phrase**, abbreviated AP, typically built around some adjective and it is possibly preceded by adverbs

Phrase structures are tree-like representations used to describe a given language's syntax on the basis of the linear order of words in the sentence, the groupings of words into constituents (phrases) and the hierarchical relation between constituents.
In Figure an example of a phrase structure tree.

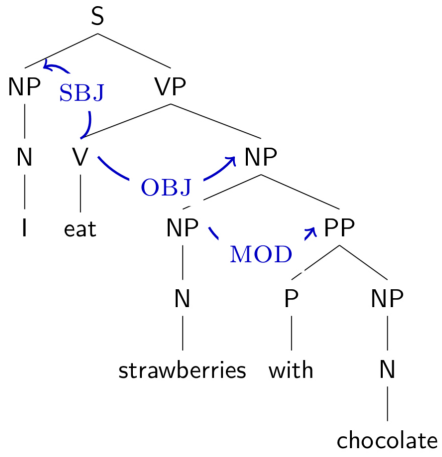
Example : Phrase structure tree for sentence "I prefer a morning flight"



The head is the word in the constituent that is grammatically the most important. The head identifies the constituent type: N is the head of an NP, V is the head of a VP, and so forth.
The head selects the arguments and modifiers appearing in the phrase:

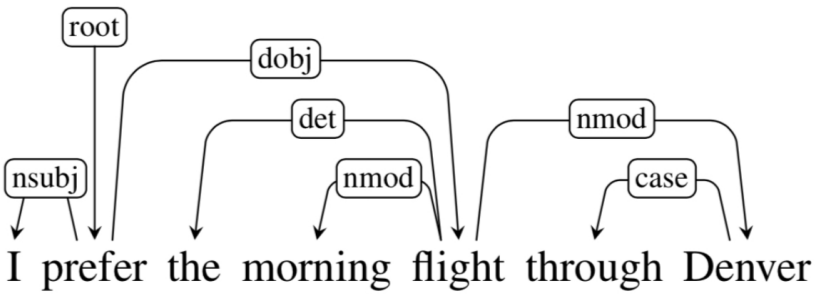
- **Arguments**: are inherent to the meaning of the phrase; they appear in fixed number depending on the head's semantics
- **Modifiers**: are optional phrases which merely supplement the head with additional information; they can appear in any number

Grammatical relations such as subject, direct object, and modifier can be retrieved from phrase structure, as we can see from Figure



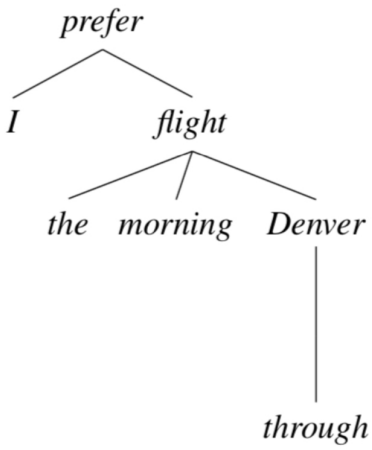
7.2 Dependency Grammar

There are different approach in order to do syntactic parsing; one approach could be phrase structure parsing, also known as constituency parsing.
This method focuses on breaking down a sentence into its constituent parts, which are often phrases like noun phrases (NP), verb phrases (VP), etc. The result is a tree (constituency parse tree) where each node represents a constituent, and the tree as a whole depicts how smaller constituents combine to form larger constituents up to the complete sentence.
Dependency parsing is another approach to syntactic parsing that instead focuses on the dependencies between words in a sentence. Each sentence is represented as a directed graph rather than a tree of constituents. In dependency grammar, we will use tree but not already seen phrase structure but dependency tree.
We will see also connection between words that could seem relation between constituents; however, these will be simply relation between words.
Dependency Tree is a very good balance between linguistic expressivity, annotation cost, and processing efficiency.
In a dependency tree, *constituents and phrase-structure rules do not play a direct role*.
A dependency tree describes syntactic structure solely in terms of the words (or lemmas) in a sentence and using an associated set of directed binary grammatical relations (such as subject, object, etc.) that hold among the words.
Example of a dependency tree: In Figure, an example of a dependency tree



Grammatical relations are depicted above the sentence with directed, labeled arcs from heads to dependents.
The root relation departs from a dummy node, called root, not displayed.

Instead, in Figure, a tree-shaped representation of previous analysis. Nodes and arcs are the same, but word ordering is dropped.

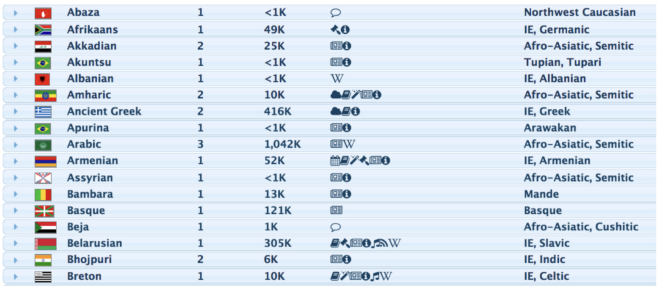


Dependency trees abstract away from word order information. This is a major advantage with relatively free word order languages, that are a problem for phrase-structure grammars, leading to a large number of rules.
Head-dependent relations provide an approximation to semantic relationships (introduced in next lecture). Phrase structures provide similar information, but it often has to be extracted with further processing.
Grammatical functions are the basis for dependency grammar. They can be subdivided in **argument** and **modifier**. Grammatical functions provide the linguistic basis for the head-dependency relations in dependency structures. The notion of head was already introduced in the context of phrase structure.
Grammatical functions can be broken into two types:

- **Syntactic roles with respect to a predicate (often a verb)**: In this case, the dependent is called an argument. It must to be present
  - Example: subject, direct object, indirect object
- **Functions that describe ways in which words can modify their heads**: In this case, the dependent is called a modifier. It is optional
  - Example: noun modifier, determiner, adverb, case

Retaking Figure above, considering **dependency relations** like $dobj$ or $case$, we can break them into argument and modifier. For example, in the $dobj$, prefer is an argument. Instead, in $case$, Denver is a modifier.

Not always what is attached to an argument is a modifier.
The Universal Dependencies (UD) project provides an inventory of dependency relations that are linguistically motivated and cross-linguistically applicable.



Abaza	1	<1K	Northwest Caucasian
Afrikaans	1	49K	IE, Germanic
Akkadian	2	25K	Afro-Asiatic, Semitic
Akuntsu	1	<1K	Tupian, Tupari
Albanian	1	<1K	IE, Albanian
Amharic	2	10K	Afro-Asiatic, Semitic
Ancient Greek	2	416K	IE, Greek
Apurina	1	<1K	Arawakan
Arabic	3	1,042K	Afro-Asiatic, Semitic
Armenian	1	52K	IE, Armenian
Assyrian	1	<1K	Afro-Asiatic, Semitic
Bambara	1	13K	Mande
Basque	1	121K	Basque
Beja	1	1K	Afro-Asiatic, Cushitic
Belarusian	1	305K	IE, Slavic
Bhojpuri	2	6K	IE, Indic
Breton	1	10K	IE, Celtic

Further constraints on dependency structures are specific to the underlying grammatical formalism.

Nodes correspond to words in the input sentence. However, they might also correspond to punctuation, or morphological units (stems and affixes).

An arc from a head to a dependent is said to be **projective** if there is a path from the head to every word that lies between the head and the dependent in the sentence. The arc from *flight* to its modifier *was* is non-projective since there is no path from *flight* to the intervening words *this* and *morning*.
More generally, if there are crossing arcs, the tree is **non-projective**

A dependency tree is said to be projective if all the arcs are projective. Otherwise, the tree is non-projective. The issue of projectivity affects the typology of the parsing algorithm, as we will see later.

Informally, languages are classified as mostly projective or non-projective according to the proportion of sentences that have the mentioned property. English is an example of a projective language; Czech is an example of a non-projective language.

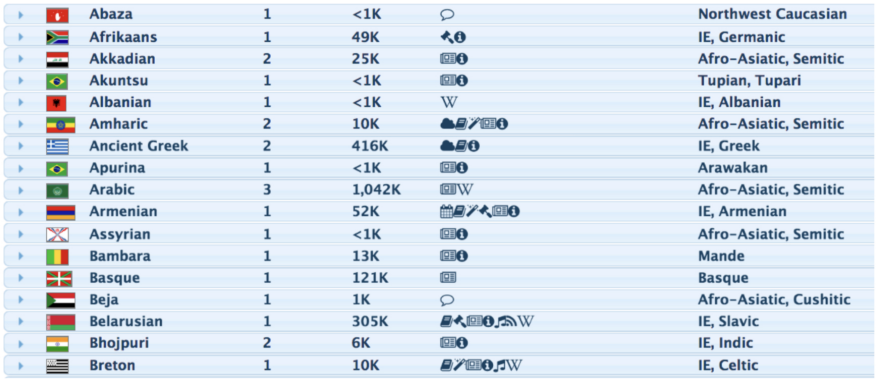
7.3 Training a parser

Dependency treebanks are dataset for training and evaluating dependency parsers, algorithm that do dependency parsing in order to catch syntax between words.
As with constituent-based methods, treebanks play a critical role in the development and evaluation of dependency parsers.

The major English dependency treebanks have been automatically produced from existing phrase structure resources, such as the Penn Treebank. Translation algorithm to be presented in next slides.

The already mentioned Universal Dependency project represents the largest effort in producing dependency treebanks.

Now, the UD project covers over 100 languages with over 200 dependency treebanks, as shown in Figure



Abaza	1	<1K	Northwest Caucasian
Afrikaans	1	49K	IE, Germanic
Akkadian	2	25K	Afro-Asiatic, Semitic
Akuntsu	1	<1K	Tupian, Tupari
Albanian	1	<1K	IE, Albanian
Amharic	2	10K	Afro-Asiatic, Semitic
Ancient Greek	2	416K	IE, Greek
Apurina	1	<1K	Arawakan
Arabic	3	1,042K	Afro-Asiatic, Semitic
Armenian	1	52K	IE, Armenian
Assyrian	1	<1K	Afro-Asiatic, Semitic
Bambara	1	13K	Mande
Basque	1	121K	Basque
Beja	1	1K	Afro-Asiatic, Cushitic
Belarusian	1	305K	IE, Slavic
Bhojpuri	2	6K	IE, Indic
Breton	1	10K	IE, Celtic

7.3.1 Transition-Based Dependency Parsing

Transition-based parsing is a popular technique at the time of writing. It is also used for semantic parsing, as we will see in the next lecture.

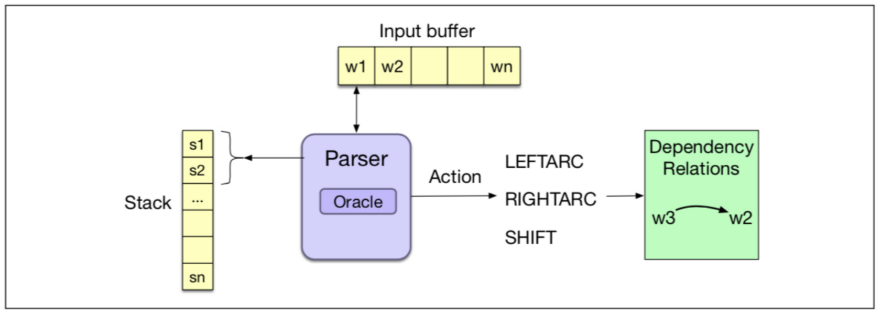
Inspired by *push-down automata*, it uses two main data structures:

1. **buffer**, initialized with the input string
2. **stack**, used as working memory

In contrast with push-down automata, in a transition-based parser, there is no use of internal states, and the stack alphabet is the same as the input alphabet (vocabulary V).

A **configuration** (state) of the parser consists of a stack (two elements), an input buffer (from which we remove, at each iteration, the already processed word), and a set of dependencies constructed so far (partial tree), as shown in Figure.

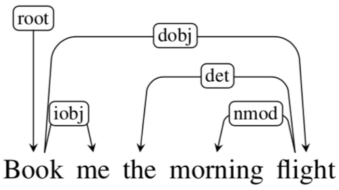
We look the two topmost words in the stack and the first word in the buffer and then we give this information to an oracle, which looks to these words and it will say what to do, with three possibilities: **LeftArc, Shift, RightArc**.



The parser is **nondeterministic**, and an oracle is used to drive the search. We will use supervised machine learning methods to produce oracles.
We consider the **arc-standard model** for **projective** parsing, using three transition operators:

1. $shift$: remove the first buffer element and push it into the stack.
2. $leftArc$: pop the second top-most stack element and attach it as a dependent to the top-most element.
3. $rightArc$: pop the top-most stack element and attach it as a dependent to the second top-most element.

The parser is purely **bottom-up**: after an element is attached, it is no longer available for further processing. In Figure an example for a sentence



Step	Stack	Word List	Action	Relation Added
0	[root]	[book, me, the, morning, flight]	SHIFT	
1	[root, book]	[me, the, morning, flight]	SHIFT	
2	[root, book, me]	[the, morning, flight]	RIGHTARC	(book → me)
3	[root, book]	[the, morning, flight]	SHIFT	
4	[root, book, the]	[morning, flight]	SHIFT	
5	[root, book, the, morning]	[flight]	SHIFT	
6	[root, book, the, morning, flight]	[]	LEFTARC	(morning ← flight)
7	[root, book, the, flight]	[]	LEFTARC	(the ← flight)
8	[root, book, flight]	[]	RIGHTARC	(book → flight)
9	[root, book]	[]	RIGHTARC	(root → book)
10	[root]	[]	Done	

The parser uses a **greedy strategy**: the oracle provides a single choice at each step, and alternative options are dropped.

The parser runs in linear time in the input string; each word must first be shifted, and later the word is attached (reduction).

The oracle tells which is the action that we have to perform in order to reach the best tree.
For the same tree, there could be a different sequence of instruction since we can do first the left-arc and then the right-arc but also the viceversa in some situations. We will have to rest on some **canonical strategy**, which does firstly the left dependencies and then right dependencies, in nearness order.
If the oracle takes the wrong choice, there is not the possibility of doing back-propagation (change our choice).

To produce **labeled dependencies** (labeled arrow), we need to expand the set of transition operators, using for instance leftArc(nsubj), rightArc(dobj), etc.

How the oracle is done? We need to create a training set in order to learn our model.
It's not enough to have labeled sentences, we need to construct it.

In transition-based parsing, the parsing task is broken down into a sequence of transitions, chosen by an oracle. The oracle takes as input a **parser configuration** and returns a **transition operator**.

Supervised machine learning is used to train classifiers that play the role of the oracle.

We need to produce **training instances** for these classifiers that are pairs of the form (configuration, transition).

We do this by deriving canonical sequences of transitions of the parser for each reference dependency tree.

Given a configuration c and a reference tree t, training instance is produced as follows, using gold sentences:

- choose $leftArc$ if on c it produces a dependency in t

- otherwise, choose $rightArc$ if

- on c it produces a dependency in t - all of the dependents of the word at the top of the stack in c have already been assigned (the top-most word has not other dependencies, its process is complete)

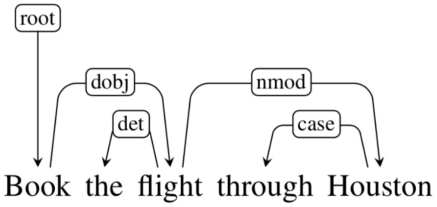
- otherwise, choose $shift$.

We assign precedence to left attachment, thus solving spurious ambiguity

$rightArc$ choice is restricted to ensure that a word is not popped before all its dependents have been attached

The above oracle is defined for the arc-standard parser. In Figure an example

of operations extracted by a gold sentence.



Step	Stack	Word List	Predicted Action
0	[root]	[book, the, flight, through, houston]	SHIFT
1	[root, book]	[the, flight, through, houston]	SHIFT
2	[root, book, the]	[flight, through, houston]	SHIFT
3	[root, book, the, flight]	[through, houston]	LEFTARC
4	[root, book, flight]	[through, houston]	SHIFT
5	[root, book, flight, through]	[houston]	SHIFT
6	[root, book, flight, through, houston]	[]	LEFTARC
7	[root, book, flight, houston]	[]	RIGHTARC
8	[root, book, flight]	[]	RIGHTARC
9	[root, book]	[]	RIGHTARC
10	[root]	[]	Done

We have a treebank (a set of sentences with its gold tree of dependencies). We take a sentence and we look into the gold tree. When we train the oracle we give to it configurations (stack + buffer)

However, due to the fact that the dominant approaches to training transition-based dependency parsers have been perceptron, multinomial logistic regression and SVM, we need to **extract numerical features** from training instances just obtained (c, op).
This process is done by defining some **feature function** f which provides a vector $f(c, op)$ of feature values.
Feature function, as for CRF, are specified by **feature templates**, defined as pairs in the form *location.properties*. In Figure possible locations and properties:

Possible locations are

- s_i: the stack element at position i (1 is the top-most position)
- b_i: the buffer element at position i (1 is the left-most position)
- the set of dependency relations r

Useful properties are

- the word form w
- the lemma l
- the POS tag t

In order to avoid data sparseness we focus on top levels of the stack and on words near the front of the buffer.
However, due to the fact that $LEFTARC$ and $RIGHTARC$ operate on the top two elements of the stack, feature templates that **combine properties from these positions** are also very useful, called **second order feature templates**.
In Figure a set of feature templates used in most of the work for transition-based parsing.

Source	Feature templates		
One word	$s_1.w$	$s_1.t$	$s_1.wt$
	$s_2.w$	$s_2.t$	$s_2.wt$
	$b_1.w$	$b_1.w$	$b_0.wt$
Two word	$s_1.w \circ s_2.w$	$s_1.t \circ s_2.t$	$s_1.t \circ b_1.w$
	$s_1.t \circ s_2.wt$	$s_1.w \circ s_2.w \circ s_2.t$	$s_1.w \circ s_1.t \circ s_2.t$
	$s_1.w \circ s_1.t \circ s_2.t$	$s_1.w \circ s_1.t$	

Remembering that a configuration is a sort of photo of our parser in a moment (so how are composed the stack and the buffer), we need to convert each configuration in numerical values.
We have different types of templates and, for each template, different results (for example, if we have a template $s_1.t \cdot s_2.t$, we have all possible combinations of two tags in the tagset). For each of these combinations (template + output required by the template) we have a feature f_i.
So, supposing that we choose 50 template, there would be 50x(possible combinations for each template) feature.
So, for each configuration (c, op) there would be a vector of this dimension where there is a 1 if the predicate for the feature f_i at position i is true (so the output required by the template corresponds to the output of the configuration) and 0 otherwise.
So, for each instance (c, op) there would be a vector of 0 and 1 very sparse, due to the high number of cells very selective.
Once these vectors are generated, their dimensionality is reduced: are removed from all vectors those features which are happened less than Δ times.
In Figure there is an example

Example : From the feature template $s1.t \circ s2.t$ we can set

$$f_{3107}(c, op) = \mathbb{I}(s_1.t \circ s_2.t = \text{NNS} \cdot \text{VBD}, op = \text{SHIFT})$$

where positions s_1 and s_2 are relative to c.

Recall that $\mathbb{I}(\mathcal{P}) = 1$ when predicate $\mathcal{P}$ holds true and $\mathbb{I}(\mathcal{P}) = 0$ otherwise.

Vectors $f(c, op)$ are very sparse and are usually implemented as dictionaries.

We create features only for those template instantiations that are observed at least $\Delta > 0$ times in the training set.

7.3.2 Alternative models for dependency parsing

A frequently used alternative to the arc-standard is the **arc-eager** parser for projective parsing, based on the following transition operators:

- $shift$: remove the first buffer element and push it into the stack.
- $leftArc$: pop the second top-most stack element and attach it as a dependent of the first buffer element.
- $rightArc$: attach first buffer element as a dependent of the top-most stack element, and shift first buffer element into the stack
- reduce: pop the stack

Another possibility is the **Attardi parser** that is able to produce **non-projective** dependency trees. It uses five transition operators:

- $shift, leftArc, rightArc$: as in arc-standard
- $leftArc2$: pop third top-most stack element and attach it as a dependent to the top-most stack element
- $rightArc2$: pop top-most stack element and attach it as a dependent to the third top-most stack element

Until now, we have defined an oracle for the arc-standard parser that provides a canonical transition operator and assumes that, in the parsing history, there are not error. These are called **static oracles**.
Instead, oracles that provides a sets of transition and not only a single one, are called **nondeterministic oracle**: we have seen that due to spurious ambiguity, there could be more than one correct transition operator.
However, oracle provides the best choice, without considering that maybe, branching another path, there might be better performance. In order to avoid

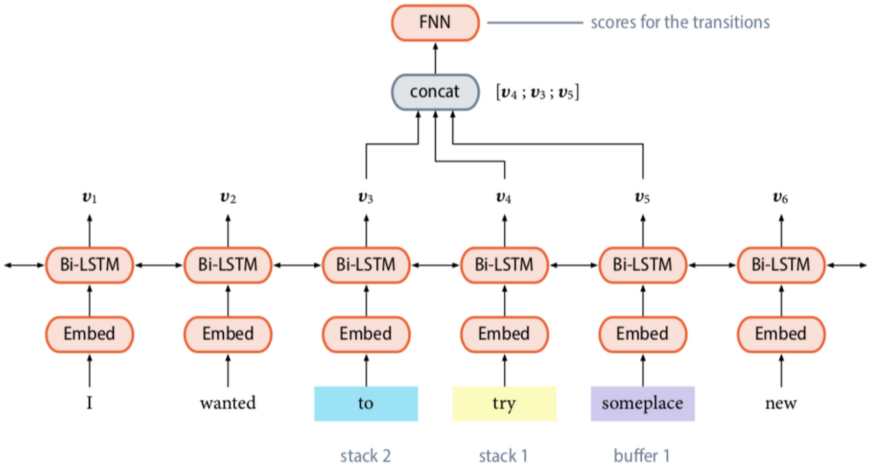
this problem, we could use **beam search**, based on an agenda of configurations having size b.
At each step we apply all possible transition operators to each configuration of the agenda and we score the resulting configurations, then we take the best b configurations and we re-apply this process until all configurations bring to a final result. Then we take the best.
We define the score of a configuration c_i on the basis of the associated **parsing history** $op_1, ..., op_i$:

$$ConfigScore(c_0) = 0.0$$

$$ConfigScore(c_i) = ConfigScore(c_{i-1}) + ConfigScore(c_{i-1}, op_1)$$

7.4 Neural Dependency Parsing

A crucial step in parser design is choosing the right feature function for the underlying statistical model. Previous machine learning approaches require complex, hand-crafted features.
Neural networks allow a much simpler approach in terms of feature engineering. Bidirectional long short term memory networks (BiLSTM) excel at representing words together with their contexts, capturing each element and an "unbounded" window around it. Crucially, the BiLSTM **is trained with the rest of the parser**, in order to learn a good feature representation of the problem.
In Figure the architecture: An LSTM maps a sequence of input vectors



$$\mathbf{x}_{1:t} = \mathbf{x}_1, \ldots, \mathbf{x}_t$$

to a sequence of output vectors

$$\mathbf{h}_{1:t} = \mathbf{h}_1, \ldots, \mathbf{h}_t$$

where $\mathbf{x}_i \in \mathbb{R}^{d_{in}}$ and $\mathbf{h}_i \in \mathbb{R}^{d_{out}}$. We schematically write this by means of the recurrent relation

$$\text{LSTM}(\mathbf{x}_{1:t}) = \text{LSTM}(\mathbf{h}_{t-1}, \mathbf{x}_t) = \mathbf{h}_t$$

Vector $\mathbf{h}_t$ is conditioned on all the input vectors $\mathbf{x}_{1:t}$.

Let $w = w_1 w_2 \ldots w_n$ be the input sentence, and let $t_1 t_2 \ldots t_n$ be the corresponding POS tags.

Assume POS tags are provided by an external module. It's so important to include POS tags: we'll see that, in order to obtain semantic relations, we need syntactic relations and, in order to obtain them, we need POS tags.

We define vectors

$$\mathbf{x}_i = \mathbf{e}(w_i) \oplus \mathbf{e}(t_i),$$

where $\mathbf{e}()$ is an embedding and $\oplus$ denotes vector concatenation.

A BiLSTM is composed by two LSTMs LSTM_L and LSTM_R reading the sentence in opposite order

$$\text{BiLSTM}(\mathbf{x}_{1:n}, i) = \text{LSTM}_L(\mathbf{x}_{1:i}) \oplus \text{LSTM}_R(\mathbf{x}_{n:i}) = \mathbf{v}_i$$

During training

- the BiLSTM encodings v_i are fed into further network layers
- the back-propagation algorithm is used to optimize the parameters of the BiLSTM

The above training procedure causes the BiLSTM function to extract from the input the relevant information for the task at hand.

Let $w = w_1 w_2 \ldots w_n$ be the input sentence, and let $t_1 t_2 \ldots t_n$ be the corresponding POS tags. Assume POS tags are provided by an external module. We define vectors

$$x_i = e(w_i) \oplus e(t_i),$$

where $e()$ is an embedding and $\oplus$ denotes vector concatenation. A BiLSTM is composed by two LSTM $LSTM_L$ and $LSTM_R$ reading the sentence in opposite order

$$\text{BiLSTM}(x_{1:n}, i) = LSTM_L(x_{1:i}) \oplus LSTM_R(x_{n:i}) = v_i$$

Our feature function $\phi(c)$ is the concatenation of the BiLSTM vectors for

- the top 3 items on the stack
- the first item in the buffer

$$\phi(c) = v_{s3} \oplus v_{s2} \oplus v_{s1} \oplus v_{b1}$$

$$v_i = \text{BiLSTM}(x_{1:n}, i)$$

Note that this feature function does not take into account the already built dependencies.

Transition scoring is then defined as a multi-layer perceptron (MLP), op is a transition of our parser

$$\text{SCORE}(\phi(c), op) = MLP(\phi(c))[op]$$

So we obtain, for each possible action, a probability of happening.
Use a margin-based objective, aiming to maximize the margin between

- the highest scoring correct action
- the highest scoring incorrect action

Let A be the set of possible transitions and G be the set of correct (gold) transitions at the current step. In the case of the arc-standard oracle in previous slides, we always have $|G| = 1$, since it's deterministic.

The hinge loss at each parsing configuration c is defined as

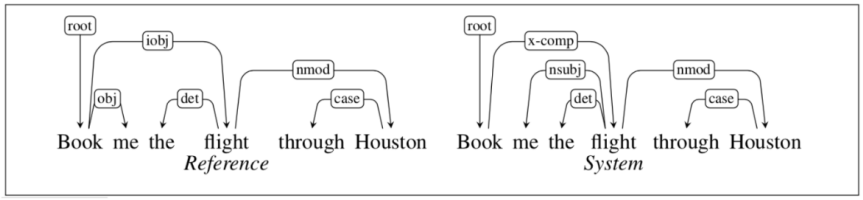
$$\max\left(0, 1 - \max_{op \in G} \text{SCORE}(\phi(c), op) + \max_{op' \in A \setminus G} \text{SCORE}(\phi(c), op')\right)$$

7.5 Evaluation

The evaluation of dependency parsing measures how well a parser works on a test set.
A crude metric is **exact match** or EM, defined as the number of sentences that are parsed correctly. However it's so selective and under EM most sentences in the test set will be marked as wrong.
Most common method for evaluating dependency parsers is based on the accuracy of the dependency relations. Given the system output and the reference parse:

- **unlabeled attachment score** (UAS) is the percentage of words in the input that are attached to the correct head, ignoring the label **labeled attachment score** (LAS) is the percentage of words in the input that are attached to the correct head with the correct dependency relation

In Figure an example of evaluation giving a reference sentence and an output sentence



8

Semantics parsing

Semantics is the study of the meaning of linguistic expression at the level of words, phrases and sentences.
Semantics focuses on what expressions conventionally mean, rather than on what they might mean in a particular context, generally presented as the distinction between **referential meaning** and **associative meaning**.
A **word sense** is a discrete representation of one aspect of the meaning of a word.
In Figure an example: **WordNet** is a large database that represents word senses

Example : For the word **bank** we highlight two word senses through in context examples

- bank[1]: ... a bank can hold the investments in a custodial account ...
- bank[2]: ... as agriculture burgeons on the east bank, the river ...

and semantic relations between word senses.
In WordNet, every word has a *set of near synonyms* clalled **synset**.
For example, in WordNet the word *bass* is associated to the synset (*bass, bass voice, basso*).
Semantic relations between word senses include:

- **synonymy**
- **antonymy** (opposite)
- **hyponymy** (enlarging the class for the correlation, for example dog/animal)
- **hypernymy** (the inverse of hyponymy)

The task of selecting the correct sense for a word given context is called **word sense disambiguation** (WSD) and there are several algorithms in order to do that.
It's important to talk about **semantic roles** or **thematic roles**, such that the role that arguments of a predicate (verb) take in the associated event, as we can see in the example of Figure.

Example :
Sasha broke the window

- Sasha is the **agent**, expressing volitional causation
- window is the **theme** or **patient**, affected in some way by the action

Roles are invariant w.r.t. syntactic construction: changing the order of words doesn't change the semantic roles of words.
In Figure a set of most used semantic roles.

Thematic Role	Example
AGENT	*The waiter* spilled the soup.
EXPERIENCER	*John* has a headache.
FORCE	*The wind* blows debris from the mall into our yards.
THEME	Only after Benjamin Franklin broke *the ice*...
RESULT	The city built a *regulation-size baseball diamond*...
CONTENT	Mona asked *"You met Mary Ann at a supermarket?"*
INSTRUMENT	He poached catfish, stunning them *with a shocking device*...
BENEFICIARY	Whenever Ann Callahan makes hotel reservations *for her boss*...
SOURCE	I flew in *from Boston*.
GOAL	I drove *to Portland*.

Defining a standard set of semantic roles may be very difficult, sometimes further specialization is needed.
For example, in Figure we can see that the label *instrument* is not enough since in the sentence, there are intermediary instruments, or passive instruments, and enabling instruments, or active instruments.

Example :
At least two kinds of **instrument** roles: intermediary instruments and enabling instruments

- the cook opened the jar with the new gadget
- the new gadget opened the jar
- Shelly ate the sliced banana with a fork
- *The fork ate the sliced banana

Each sense of a verb is associated with a set of semantic roles, called the **thematic grid**. Thematic grids are at the basis of NLP tasks for semantic role labelling that construct **shallow meaning** representations.

Of very high importance is also **Proposition Bank**, which is a resource containing sentences with a little bit of syntactic analyses for grouping sentences, and are completely labeled with semantic roles. Here roles are given only numbers rather than names, that are specific for an individual verb sense like $ARG0$, $ARG1$ and so on.
So roles of arguments of a predicate are assigned according to:

- *ARG0* represents several cases of agent
- *ARG1* represents several cases of patient
- *ARG2* is often benefactive, instrument, attribute etc.

PropBank also has a number of non-numbered arguments called $ArgMs$, ($ArgM-TMP, ArgM-LOC$, etc.) which represent modification or adjunct meanings. The semantics of the other roles are less consistent.
In Figure the PropBank entry for one of the senses of the verb **agree**.

agree.01
ARG0: Agreer
ARG1: Proposition
ARG2: Other entity agreeing

Example :

- [ARG0 The group] agreed [ARG1 it wouldn't make an offer]
- [ARGM-TMP Usually] [ARG0 John] agrees [ARG2 with Mary]

FrameNet is another role-labelling resource and here semantic roles are specific to a **frame**, a structure that can be shared across several words.
A frame is a conceptual structure that describes a specific type of event, relation, or entity along with its participants and properties. Each frame includes a variety of roles, which represent the participants and other conceptual roles in the scenarios described by the frame. An example is shown in Figure below.

Example : All of the words highlighted below share the same frame

- [ITEM Oil] **rose** [ATTRIBUTE in price] [DIFFERENCE by 2%]
- [ITEM Microsoft shares] **fell** [FINALVALUE to 7 5/8]
- A steady **increase** [INITIALVALUE from 9.5] [FINALVALUE to 14.3] [ITEM in dividends]
- [ITEM It] has **increased** [FINALSTATE to having them 1 day a month]

In FrameNet, lexical units are words or phrases that evoke a specific frame. For

example, the verb "sell" evokes a Commercial Transaction frame, involving roles such as Seller, Buyer, Goods, and Money.
This allows to make inferences across different verbs, and also between verbs and nouns.
In Figure an example: here the frame was **change-position-on-a-scale** that is used by words such as increase, fall, rise, grow.

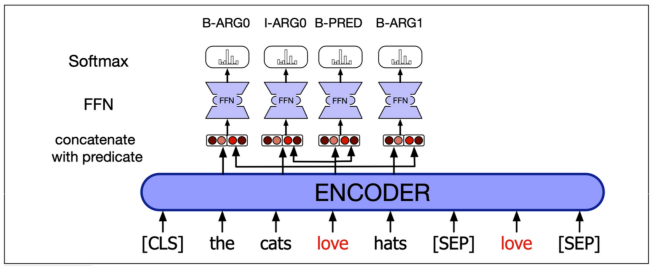
Core Roles	
ATTRIBUTE	The ATTRIBUTE is a scalar property that the ITEM possesses.
DIFFERENCE	The distance by which an ITEM changes its position on the scale.
FINAL_STATE	A description that presents the ITEM's state after the change in the ATTRIBUTE's value as an independent predication.
FINAL_VALUE	The position on the scale where the ITEM ends up.
INITIAL_STATE	A description that presents the ITEM's state before the change in the ATTRIBUTE's value as an independent predication.
INITIAL_VALUE	The initial position on the scale from which the ITEM moves away.
ITEM	The entity that has a position on the scale.
VALUE_RANGE	A portion of the scale, typically identified by its end points, along which the values of the ATTRIBUTE fluctuate.
Some Non-Core Roles	
DURATION	The length of time over which the change takes place.
SPEED	The rate of change of the VALUE.
GROUP	The GROUP in which an ITEM changes the value of an ATTRIBUTE in a specified way.

8.1 SEMANTIC ROLE LABELLING

Semantic role labelling (SRL) is the task of automatically finding the semantic role of each argument of a predicate in an input sentence.
Most popular approaches use supervised learning (PropBank or FrameNet):

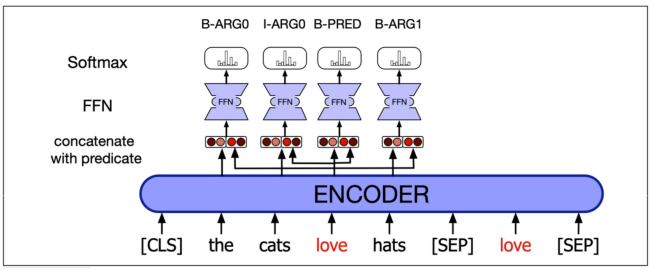
- guided by phrase-structure trees
- guided by dependency trees
- exploit sequence labelling

In Figure an example



A simple approach to SRL is to view this task as a sequence labeling problem, using BIO tags (seen in POS tags chapter).

So we assume an input sequence of words $w_{1:n}$ and the set $\mathcal{Y}(w_{1:n})$ of all tag sequences compatible with $w_{1:n}$ and we are interested in computing the tag sequence $\hat{y_{1:n}}$ that has maximum probability in the set of possible tag sequences.
We can leverage on a neural network algorithm which architecture is shown in Figure. Each input word is mapped to an embedding by using a biLSTM encoder



and then each token is concatenated with the predicate/target embedding.
These augmented embedding are given as input of a FFNN with a final softmax layer which maps results to a distribution over possible SRL labels.
The standard evaluation for SRL is to require that each argument label must be assigned to the exactly correct word sequence. As usual for sequence labeling, precision, recall, and F-measure are computed.
It's also important to talk about **selectional restriction** that is a semantic constraint that a verb imposes on the semantic characteristics of its arguments. For example, the verb "eat" typically requires an animate subject (like "people" or "animals") and an edible object ("food", not "ideas").
This helps also to solve the PP attachment problem (in the sentence "I saw the man with the telescope," the PP "with the telescope" could either modify "the man" (suggesting the man has a telescope) or the verb "saw" (indicating that "I" used a telescope to see the man).

8.2 Meaning

In recent years, the field of natural language processing (NLP) has seen a shift towards more integrated and holistic approaches to understanding text.
Traditionally, tasks like word sense disambiguation (WSD), named-entity recognition (NER), semantic role labeling (SRL), and coreference resolution were often tackled independently. Each task addresses different aspects of semantic analysis.

Recently, there has been a major effort in NLP to design systems and frameworks that can process and integrate all of these semantic tasks simultaneously. This integration aims to provide a deeper and more unified understanding of text by considering how these various aspects of semantics interact with each other.
The meaning of linguistic expressions can be captured by means of formal structures called **meaning representations**.
Meaning representations use symbols that correspond to:

1. **Objects or Variables:** These are the entities or elements discussed or referred to within a sentence.
2. **Properties of Objects:** These are the attributes or descriptors that define or modify the objects.
3. **Relations Among Objects:** These describe the interactions or relationships between the objects.

We want sentences with the same semantics to be assigned the same meaning representation, as shown in Figure.

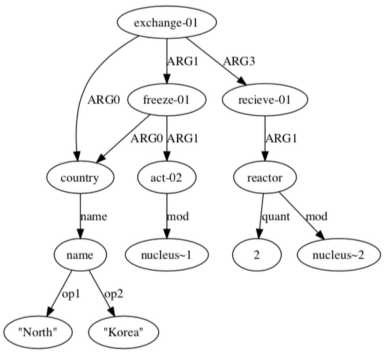
Example :

- he described her as a genius
- his description of her: genius
- she was a genius, according to his description

Several semantic formalisms for meaning representations have been proposed in the literature:

- abstract meaning representation (AMR): these are rooted, labeled, directed graphs where:
  - nodes denote concepts, events, properties, and states
  - arcs denote semantic relations and semantic roles When an entity plays multiple roles in a sentence, we employ graph re-entrancy (nodes with multiple parents). AMR are thus dependency structures, but differently from the syntactic case, they show re-entrancies, as shown in Figure.
- universal conceptual cognitive annotation
- universal decompositional semantics
- minimal recursion semantics (MRS)
- discourse representation theory (DRT)

'North Korea froze its nuclear actions in exchange for two nuclear reactors.'



Also **First-order logic** (FOL) has been traditionally used for meaning representation.

The process of mapping sentences to meaning representations is called **semantic parsing or semantic analysis**.

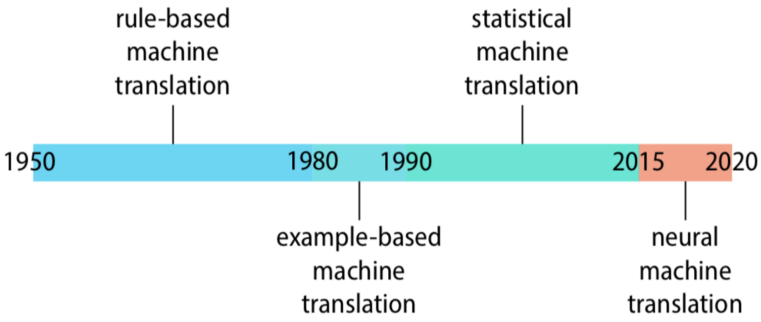
AMR parsing is receiving increasing attention. Due to the already mentioned similarity between AMR and dependency structures, transition-based parsers have been adapted to AMR parsing: we consider a model with the following transition operators:

- $SHIFT$: decide if and what to push on the stack after consuming a token from the buffer
- $LEFTARC(l)$: creates an edge with label l between the top-most node and the second top-most node in the stack, and pops the latter
- $RIGHTARC(l)$: is the symmetric operation, but does not pop any node from the stack
- $REDUCE$: pops the top-most node from the stack and recovers reentrant edges with its sibling nodes

9

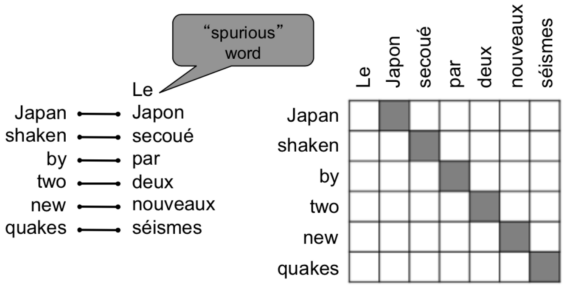
Machine Translation

Machine translation (MT) investigates models and algorithms to translate from one language to another. In Figure a timeline of machine translation:



Translation is a difficult task due to these facts:

- Languages differ in the basic word order of verb, subject and object in simple declarative clauses
- In translating individual words one needs to detect word sense, since different word senses map to different words in target language. Moreover, the target language could have more constraints than the source language (for example Italian has grammatical gender, English not)
- There may be a lexical gap between a pair of languages, where no word in the target language can express the exact meaning of a word in the source language. In these case one needs to resort to **paraphrasing**.
  Word alignment is the correspondence between words in the source and target sentences.
  Spurious words have no counterpart in the target language, as shown in Figure



The alignment can be **many to one** (aboriginal people ↪ autoctoni, two words is aligned to a single word) or **one to many** ('implemented' ↪ 'mis en application') and can also be **many to many**.

9.1 Statistical machine translation

As we can see from the pipeline shown in Figure above, one of tha latest method of doing machine translation is referred to **statistical machine translation**.
Machine translation can be formulated as a **structured prediction** task that given a source sentence x, the model needs to find the most probable target sentence $\hat{y}$:

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

However, as always said, posterior probabilities are not known a priori so we apply the Bayes' rule to decompose them into two components, obtaining the **statistical machine translation** (SMT):

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

where $P(x|y)$ is a **translation model** and $P(y)$ is a language model. Here could born a question: where is the advantage of using Bayes'rule since we move from probabilities of a sentence conditioned by a sentence to the same thing?
You get an advantage during training due to the fact that language model $P(y)$ is charged of the word ordering and the search of the maximum argument is simplier since sentences with an improbably order will have very low probability. It's well explained below:
Why two models rather than one?

- $P(y|x)$ needs to account for **both** the word translation and the ordering of the output y
- $P(x|y)$ assigns large probability to strings that have the necessary words (roughly at the right places)
- $P(y)$ assigns large probability to well-formed strings y, regardless of the connection to x.

$P(x \mid y)$ and $P(y)$ collaborate to produce a large probability for well-formed translation pairs.
So, using these models, we need to estimate:

- the language model
- the translation model

using an efficient **search** for the string y that maximizes the product for given X.
One possible way of moving is using the **alignment relation**: if a is a variable which ranges over all possible alignment relations for x and y, we can write:

$$P(x|y) = \sum_a P(x, a|y)$$

What we want to say is that the translation of y from x can came from different alignments, so we can split the probability as the sum of probabilities for each alignment of x, breaking the model in small pieces, easier to use.
There are several ways to solve this problem, with increasing order of complexity.

9.2 Neural Machine Translation

Neural Machine Translation (NMT) models the translation task through a single artificial neural network.
Let x be the source text and $y = y_1 \cdots y_m$ be the target text.
In contrast to SMT, in NMT we **directly model** $P(y|x)$ using an approach similar to the one adopted for language modeling:

$$P(y \mid x) = P(y_1 \mid x) \cdot P(y_2 \mid y_1, x) \cdot P(y_3 \mid y_1, y_2, x) \cdots P(y_m \mid y_1, \ldots, y_{m-1}, x) =$$

$$= \prod_{t=1}^{m} P(y_t \mid y_1, \ldots, y_{t-1}, x)$$

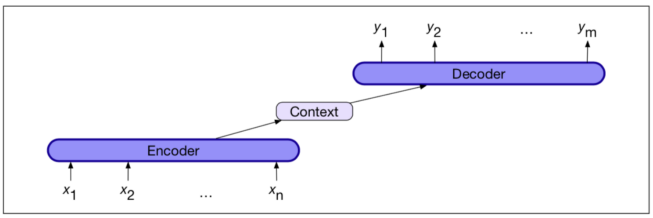
And so now the question is, how we compute the probability $P(y_t|y_1, ..., y_{t-1}, x)$? The architecture proposed is called **Encoder-decoder networks**. Encoder-decoder networks, also called sequence-to-sequence (seq2seq) networks, are models capable of generating contextually appropriate, arbitrary length sequences.
Encoder-decoder networks have been applied to a wide range of NLP applications

- machine translation
- summarization
- question answering
- dialogue

The encoder-decoder model consists of two components.
The encoder is a neural network that produces a representation of the source sentence. The decoder is an **autoregressive language model** that generates the target sentence, conditioned on the output of the encoder.
Autoregressive means that it takes its own output as new input.
The encoder-decoder model can be implemented in two ways, through recurrent networks or else through the transformer. In general, use of attention techniques improves model performance.

9.2.1 Encoder-decoder networks using RNN

Encoder-decoder models based on RNN use the following main idea, shown in Figure.

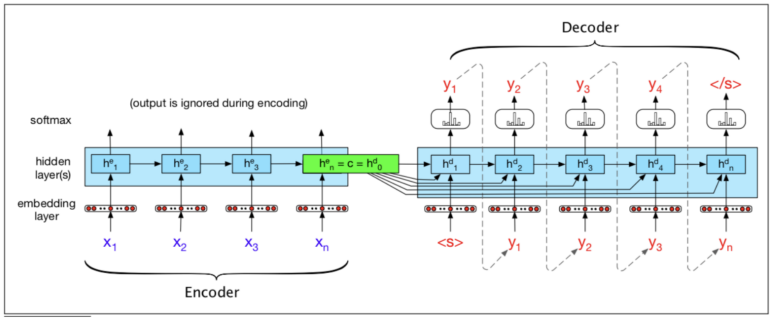


The output of the encoder is called context and drives the translation, together with the decoder output.
Inference: We present a **greedy** inference algorithm. Later we also discuss a beam search inference algorithm. $P(y \mid x)$ can be computed as follows:

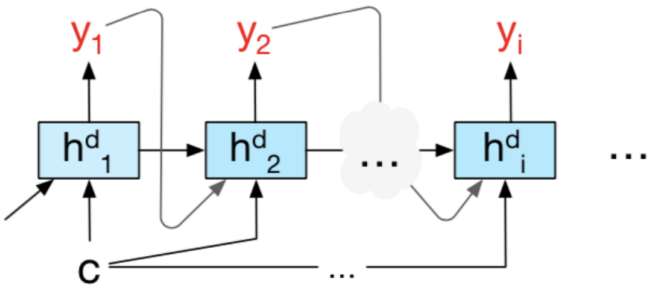
- run an RNN **encoder** through $x = x_1 \ldots x_n$ performing forward inference, generating hidden states $h_t^e, t \in [1..n]$ where e stands for encoder and t for the token

- run an RNN **decoder** performing autoregressive generation; to generate $y_t, t \in [1..m]$ next token, use
  - encoder hidden state h_n^e so the hidden state of the final token which embedds all information about context
  - decoder hidden state h_{t-1}^d of the previous iteration
  - embedding of word y_{t-1} of the last token generated

We can see the overview of inference process in Figure As we can see from



Figure, in the decoder we start by giving as input an empty sentence, then the first word y_1 is predicted and is then added to the next input. The decoder stop when the EOS marker is found.
The final hidden state of the encoder h_n^e is a representation of the entire input sequence x, also called context C. It's given as input of the decoder at each step of decoder, with also the previous decoder hidden state and the output generated by the previous state, as we can see from Figure.



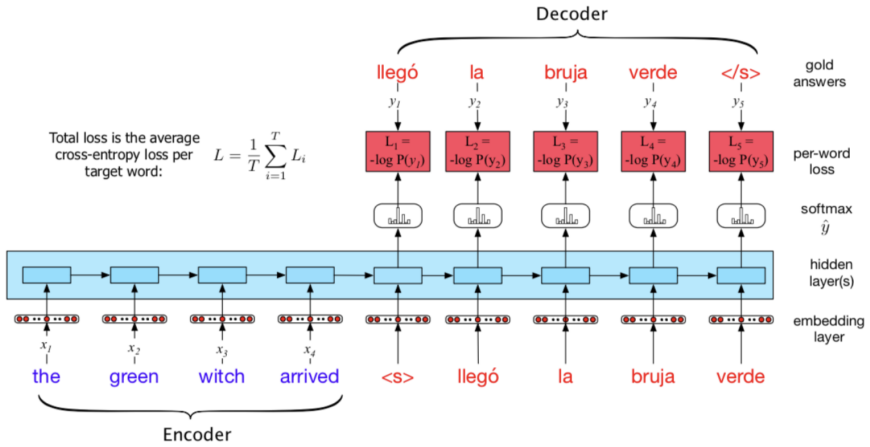
Assume:

- $b(u)$ is an embedding of word u from the vocabulary V
- g affine transformation combined with non-linear component;
- f affine transformation

The model equations are:

$$
\begin{aligned}
h_t^e &= \text{RNN}(h_{t-1}^e, b(x_t)) \\
c &= h_0^d = h_n^e \\
h_t^d &= g(c, h_{t-1}^d, b(\hat{y}_{t-1})) \\
z_t &= f(h_t^d) \\
s_t &= \text{softmax}(z_t) = P(\cdot \mid \hat{y}_1, \ldots, \hat{y}_{t-1}, x) \\
\hat{y}_t &= \arg\max_{u \in V} P(u \mid \hat{y}_1, \ldots, \hat{y}_{t-1}, x)
\end{aligned}
$$

This is done at each step in order to generate the next token y_t.
We use multi-layer architectures for encoder and a widely used approach is by using biLSTMs, where the hidden states from the top layers from the forward and backward passes are concatenated.
We have said that it's a greedy algorithm due to the fact that at each iteration we increase our input.
Training: The encoder-decoder is optimized as a single system during training, so that a small change is propagated across the entire system.
Training proceeds as with as any RNN-based language model as already seen, so, given the source text and the gold translation, we compute the average loss w.r.t. our predictions on next word in the translation.
In Figure the schematic representation for computing the average loss during training.



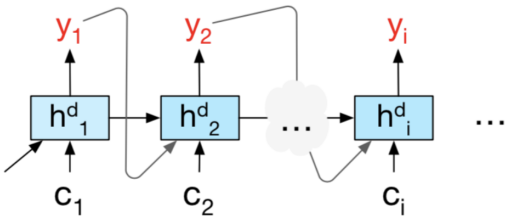
Remembering what is the cross entropy, that is a sort of distance between two distribution.

We compute cross-entropy at each step, using the gold distribution and the founded distribution (vector of probabilities of all possible outputs for the next token y_t). Due to the fact that in the gold distribution all words probabilities are set to zero, exception done for the correct one, is selected only one word and we'll get the $-logP(y_t)$ where is the log of the probability of the predicted model for the correct word.
What we do is to take the average of the cross entropy computed at each step and we minimize it, y changing parameters of decoder and encoder, in order to train our model.
During training, the decoder uses gold translation words as the input for the next step prediction. This is called **teacher forcing**.
During inference, the decoder uses its own estimated output as the input for the next step prediction. Thus the decoder will tend to deviate more and more from the gold target sentence as it keeps generating more tokens.

9.2.2 Attention

The context vector $\vec{c}$ must represent the whole source sentence in one fixed-length vector. This is called the **bottleneck problem**.
If the sentence is very long, the context vector may miss some information
In addition, which information from $\vec{c}$ do we use at each step in the translation?
The **attention mechanism** allows the decoder to get information from all the hidden states of the encoder using a **dynamic context**.
The idea is to compute context $\vec{c}_i$ at each **decoding step** i, so for each token i is used a specific context vector. This context is computed as a **weighted sum** of all the encoder hidden states $\vec{h}^e_j$ where **weights** are computed as a function of the decoding step i.
So the weights are used to focus on particular hidden states of the encoder (so precise tokens) that are relevant for the token being predicted at step i, given to them higher weights.
So attention replaces the static context to dynamic context computed at each step of decoding using encoder hidden states; so now the computation of the hidden state of the decoder at step i depends on things illustrated in Figure.
At each step i of decoding, are computed **relevant scores** $score(\vec{h^d_{i-1}}, \vec{h}^e_j)$ for each encoder state $\vec{h}^e_j$.
The simplest such score, called **dot-product attention**. implements relevance as

$$\mathbf{h}_i^d = g(\mathbf{c}_i, \mathbf{h}_{i-1}^d, y_{i-1})$$



similarity through dot-product:

$$score(\vec{h_{i-1}^d}, \vec{h_j^e}) = \vec{h_{i-1}^d} \cdot \vec{h_j^e}$$

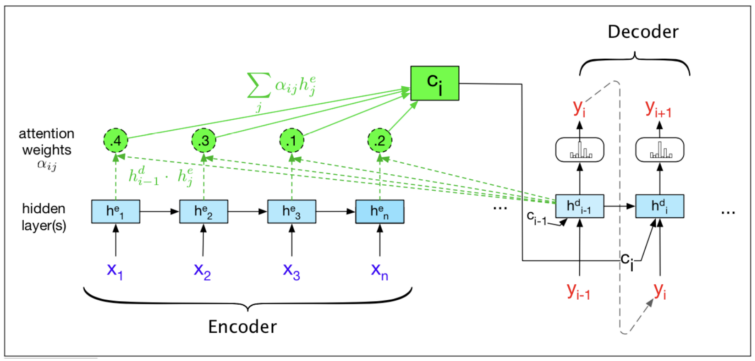
Then the scores **are normalized in order to create weights** $a_{ij}, j \in [1..n]$ using the formula illustred in Figure.

$$\begin{aligned} \alpha_{ij} &= \text{softmax}(\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e)) \\ &= \frac{\exp \text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e)}{\sum_k \exp \text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_k^e)} \end{aligned}$$

The quantity a_{ij} is proportional to the relevance of encoder hidden state $\vec{h_j^e}$ for predicting token i.
So finally we compute the fixed-length context vector that is **dynamically updated**:

$$\vec{c_i} = \sum_j a_{ij} \vec{h_j^e}$$

In Figure the architecture for this method



More sophisticated scoring functions for attention have been proposed, as the

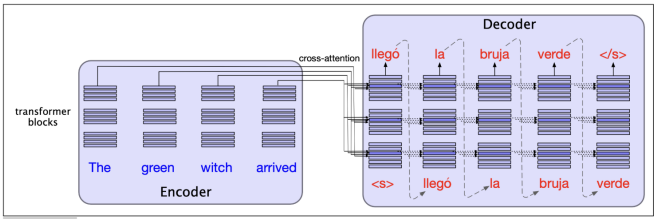
bilinear model:

$$score(\vec{h_{i-1}^d}, \vec{h_j^e}) = (\vec{h_{i-1}^d})^T W_s \vec{h_j^e}$$

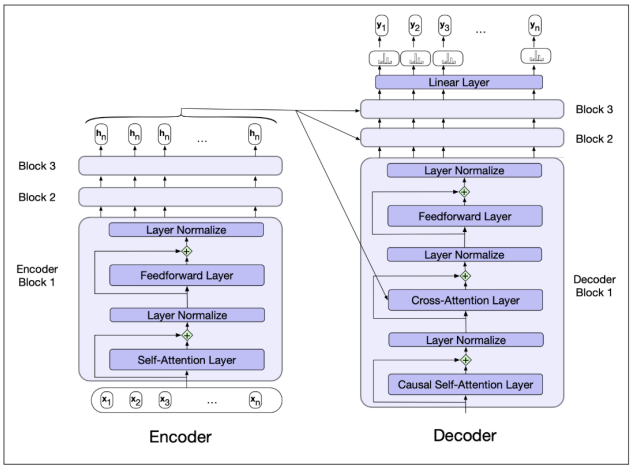
Learnable parameters W_s weight all feature combinations from decoder and encoder hidden states.
This score allows the encoder and decoder to use different dimensions for their hidden states.

9.3 Encoder-decoder with Transformer

Encoder-decoder models based on Transformers use the following main idea: the self-attention in the encoder is allowed to look ahead at the entire source language text instead of only the last hidden state, as we can see in Figure.



Now the standard architecture for neural MT is the **transformer**.
Remembering how it is composed a transformer, the architecture for MT is as shown in Figure where each block of the **decoder** has an **extra layer implementing cross attention**.



The cross-attention in the decoder is a variant of the self-attention in the encoder where:

- the queries are computed from the previous layer of the decoder, as in the encoder
- the keys and values are computed from the output of the encoder

Assume that

- $\mathbf{H}^{enc}$ is the output from the topmost encoder layer
- $\mathbf{H}^{dec[i-1]}$ is the output from the previous decoder layer **CHIEDI SE E L'ULTIMO OUTPUT**
- $\mathbf{W}^{Q}$ is the cross-attention layer's query weights
- $\mathbf{W}^{K}$, $\mathbf{W}^{V}$ are the cross-attention key and value weights

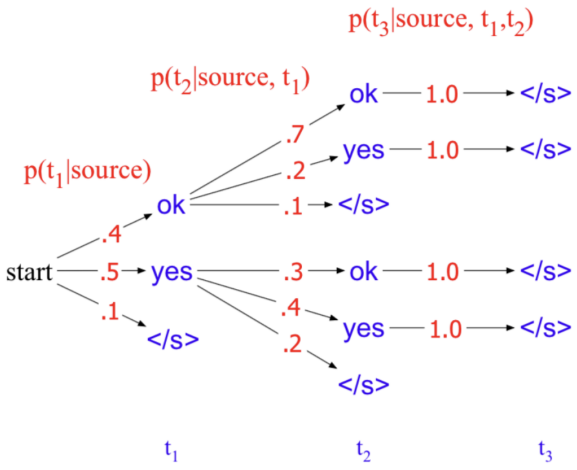
The model equations are

$$\mathbf{Q} = \mathbf{W}^{Q}\mathbf{H}^{dec[i-1]}$$
$$\mathbf{K} = \mathbf{W}^{K}\mathbf{H}^{enc}$$
$$\mathbf{V} = \mathbf{W}^{V}\mathbf{H}^{enc}$$
$$\text{CrossAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^{T}}{\sqrt{d_k}}\right)\mathbf{V}$$

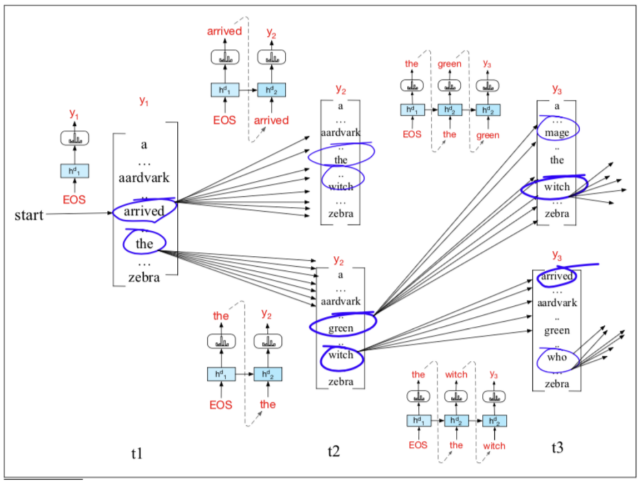
In summary, the self-attention layer in the encoder is allowed to look ahead at the entire source language text (no masking), the self-attention layer in the decoder is causal, based only on the already generated tokens and the cross-attention layer in the decoder attends to each of the source language words.
Also here is used teacher forcing: at each time step in decoding we force the system to use the gold target token rather than the decoder output.

9.3.1 Beam search

We have seen a **greedy algorithm** for inference, that is an example of local search, where it's chosen the token as the single most probable at each step of decoding. However, greedy algorithms makes choices that are locally optimal but, at the end, this may not find the highest probability path.
So we define a **search tree** for the decoder, an example of global search, where the nodes are states, representing the already generated part of translation and the branches are the actions of generating the next token.



An example is shown in Figure 1
Unfortunately, **dynamic programming** is not applicable to the searche tree because of long-distance dependencies and an **exhaustive search** is infeasible.
So what we could do is **beam search** where we keep K possible hypotheses at each step.

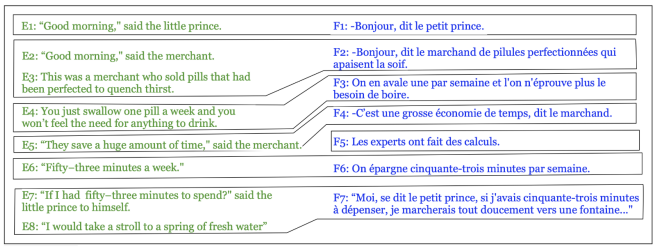


As we can see from the exampe with beam width $K = 2$ in Figure, we start by selecting the K best hypothesis. Then, for each selection we expand it using the selected token as new token for the input and we generated the vector of all possible output tokens. Then we take the K best and we repeat it until a complete hypothesis is found and $K = 0$.
It's important to notice that greedy algorithm is an istance of beam search with $K = 1$.
However, we can encounter a problem because models generally assign lower

probabilities to longer strings, the algorithm favours completed hypotheses which are shorter.
It's useful to apply some form of length normalization to hypotheses based on number of words.

9.3.2 Training on parallel corpora and evaluation

As we said before, the training part is done using supervised learning techniques; Machine translation models are trained on **parallel corpora**, which are texts in two or more languages with **aligned sentences**, as we can see from Figure.



Given two documents that are translations of each other, we can automatically align sentences using:

1. a score function measuring how likely a source span and a target span are translations
2. an alignment algorithm that finds a good alignment between the documents using the scores

Score function usually exploits multi-lingual word embeddings and cosine similarity and alignment algorithm exploits dynamic programming and are simple extension of the minimum edit distance algorithm.
The most popular automatic metric for MT systems is **BLEU**, for BiLingual Evaluation Understudy.
The N-gram precision for a candidate translation is the percentage of N-grams in the target sentence that also occur in the reference translation.
BLEU combines N-gram precisions for $N \in [1..4]$ using the geometric mean. A brevity penalty for too-short translations is also added.
BLUE score ranges in $0-100\%$ and over 50% is a very good score; instead, under 15% means bad translation quality, high level of post-editing will be required.
BLEU has been criticised for not correlating well with human judgement.

An alternative metric is **METEOR**, for Metric for Evaluation of Translation with Explicit ORdering, designed to fix some of the problems found in BLEU, producing better correlation with human judgement at the sentence or segment level.
The metric is based on the harmonic mean of unigram precision and recall, with recall weighted higher than precision.

10

Question Answering

Question answering (QA) systems focus on factoid questions, that is, questions that can be answered with simple facts.
An example could be:

- 'Where is the Louvre Museum located?'
- 'What is the average age of the onset of autism?'

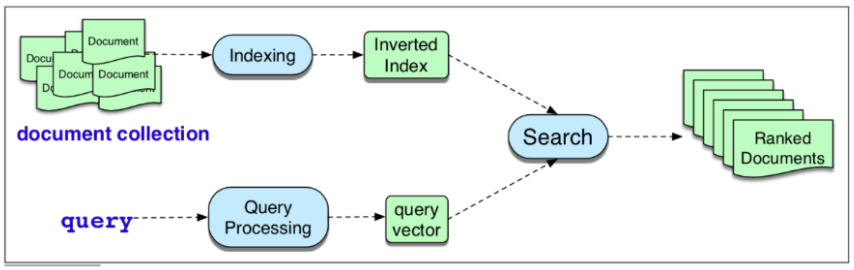
Instead, **non-factoid questions** requires articulated answer. These questions often require deeper understanding, analysis, or opinion and cannot be answered with a simple fact or a short phrase.
An example could be 'What are the main causes of climate change?'.
There are two main approaches used to build systems that can answer factoid questions:

1. **Text-based QA**: These systems use efficient algorithms (information retrieval, after question understanding) to find relevant documents from text collection and use reading comprehension algorithms on relevant documents to select span of text containing the answer.

2. **Knowledge-based QA**: These systems produce a semantic representation (vector which embedds context) of the query and match semantic representation against fact database (fact database is a structured repository of information, each fact in the database is also represented in a way that captures its semantics, often using similar embedding techniques).

10.1 Text-based QA

We can subdivide these systems in two main steps.
The first step in text-based QA uses information retrieval (IR) systems to map input queries to a set of documents from some collection, ordered by **relevance**, as shown in Figure.



The image illustrates the process of document retrieval in an information retrieval system.
It begins with a document collection, where a large set of documents is stored and prepared for indexing.
During the indexing phase, these documents are processed to create an inverted index, a data structure that maps terms to their occurrences within the documents.

When a user inputs a search query, the system processes the query by tokenizing and normalizing it, transforming it into a query vector that represents the search terms.
This query vector is then used to search through the inverted index. The search component calculates the similarity between the query vector and the vectors of documents in the index, retrieving those that match the search terms.

Finally, the retrieved documents are ranked based on their relevance to the query. The ranking is determined by the similarity scores, with higher scores indicating greater relevance.
The result is a list of ranked documents presented to the user, efficiently providing the most relevant information from the document collection.
The second step in text-based QA is called span based **machine reading** (related to human reading comprehension).
The input is a factoid question concatenated with a passage that could contain the answer and the output is the answer fragment, or else $NULL$ if there is no

answer in the passage.
An example is shown in Figure

> **Example** :
> question: "How tall is Mt. Everest?"
> passage: "Mount Everest, reaching 29,029 feet at its summit, is located in Nepal and Tibet ..."
> output fragment: "29,029 feet"

When we talk about a passage, we talk about a fraction of a document of small size. In a document there are more passages and, if the document is very small, the passage is the entire document.
Let $q = q_1, \ldots, q_N$ be a **query** and let $p = p_1, \ldots, p_M$ be a **passage**, where q_i and p_j are tokens.

Lengths are imbalanced: $N \approx 15$, $M \approx 100$.

A **span** is any fragment $p_i, \ldots, p_j$ of p, with i the start position and $j \geq i$ the end position.

The goal is to compute the probability $P_{\text{span}}(p_i, \ldots, p_j \mid q, p)$ that span $p_i, \ldots, p_j$ is the answer to q.

We present two simple **neural** approaches to span based machine reading, computing $P_{\text{span}}(p_i, \ldots, p_j \mid q, p)$ in two different ways:

- on the basis of BERT-like, pre-trained contextual embeddings
- on the basis of RNN and attention-like mechanisms

These approaches are not state-of-the-art (SoTA) but, when hyperparameters are properly tuned, they provide very good performance.

10.1.1 Using contextual embeddings

To compute the probability $P_{\text{span}}(p_i, \ldots, p_j \mid q, p)$, we make the **simplifying** assumption that:

$$P_{\text{span}}(p_i, \ldots, p_j \mid q, p) \approx P_{\text{start}}(i \mid q, p) \cdot P_{\text{end}}(j \mid q, p)$$

So we assume independence for start and end of span, and we compute the probability as the product of the probability that the start of the span is token

i and the end of the span is token j. The independence assumption is a simplification: in real word, these probabilities are conditioned on both query and passage.

Firstly is used pre-trained encoder BERT in order to encode question and passage, separated by a $[SEP]$ token. So, for each word there is the embedding vector $\vec{e}(\cdot)$ which contain information on the context.

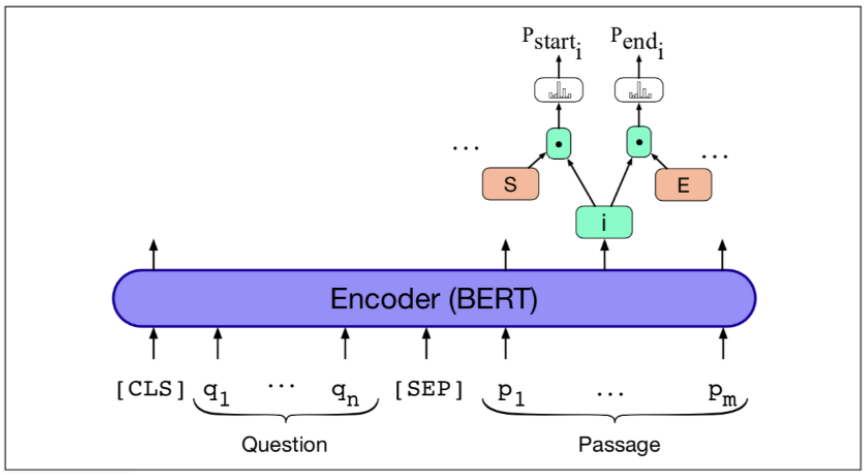
Let $e(\vec{p}_i)$ be the BERT embedding of token p_i within passage p.

During fine-tuning, **start vector S** is learned to estimate start probabilities for each position i, using dot product and softmax:

$$P_{\text{start}}(i \mid q, p) = \frac{\exp(\mathbf{S} \cdot e(\vec{p}_i))}{\sum_j \exp(\mathbf{S} \cdot e(\vec{p}_j))}$$

So, during training is learned the parameter **S** which is used, in the formula above, in order to obtain starting probabilities for each position of i.

Similarly, we learn end vector **E** to estimate $P_{\text{end}}(j \mid q, p)$.
In Figure the architecture.



At **inference** time, the score of a candidate span from position i to j is:

$$\mathbf{S} \cdot e(\vec{p}_i) + \mathbf{E} \cdot e(\vec{p}_j)$$

The model prediction is the highest scoring span with $j \geq i$.

For each **training** instance, we switch to probabilities and we compute the negative sum of the log-likelihoods of the correct start position i^* and the correct

end position j^*:

$$L = -\log P_{\text{start}}(i^* \mid q, p) - \log P_{\text{end}}(j^* \mid q, p)$$

So it's a supervised learning task where we force correct positions and we change **S** and **E** until the loss is minimized.

Averaging for all instances provides the fine-tuning **loss**.

Many datasets contain **negative examples**, that is, (q, p) pairs in which the answer to q is not in the passage p.
Negative examples are conventionally treated as having start and end indices pointing to the $[CLS]$ special token.
However, for many datasets the annotated documents/passages have length M larger than the maximum 512 input tokens BERT allows. In such cases slide over the entire document a window of size $512 - N$, the question length and in order to do not miss any response, we use a **stride** of $\Delta = 128 tokens$ so passages are overlapped of 128 tokens.

This produces several instances in the training set for the given document, most of which are negative examples. If number of negative examples is disproportionately large, then you need to balance the training set by downsampling.

10.1.2 Stanford attentive reader

This method leverage on RNN combined with an attention-like mechanism. Assume a query $q = q_1, ..., q_N$ and a passage $p = p_1, ..., p_M$ where q_t and p'_t are tokens. Let $e(\vec{q}_t)$ and $e(\vec{p}'_t)$ be **static, non contextual** embeddings associated for tokens q_t and p'_t.
We use **bidirectional LSTM** to encode individual tokens from query and passage, encoded independently.
First we start with **monodirectional embeddings** as shown in Figure So in vector $h_t^{\vec{(q)}}$ is the hidden information for token t of the query, computed using the next and the previous hidden layer and the embedding of q_t.
So we concatenate these monodirectional embeddings for **passage tokens**, obtaining: Then we concatenate also **boundary query token embedding** in order to encode entire query: Then we derive an attention distribution (weights that tell us how much interesting for us is position i in the passage) by computing a vector of **bilinear products** and applying a softmax: where W is a learnable

matrix, learned during training.
So, for each token i in the passage we have a weight that provide us information about how much it's important that token.
Then we use attention to combine passage tokens, and compute an output vector $\vec{o}$, function also of the query q: Each candidate answer c, fragment of passage, is encoded as a vector $\vec{x_c}$, using some function of the vectors $\vec{h_i^{(p)}}$ of the corresponding passage (for example. combination of the boundary token hidden layer of the answer).
Finally, we score each candidate c computing an inner product (similarity) between it and $\vec{o}$ (remember that is function of the query) and we take the maximum.
This architecture can be trained end-to-end from a loss based on the log-likelihood of the correct answers.
Performance of the two previous models can be improved using any of the following ideas:

- Encode q by a weighted learnable combination of all query states, not just the boundary states
- Use more than one layer when implementing BiLSTM
- Use similarity score between entire q and p
- Concatenate vector representation of each token in passage with additional **features** like POS, NER tags, term frequency etc.

10.1.3 Retrieval-augmented generation

Recall that using LLM and prompting, we can recast the task of question answering as word prediction, as we can see in the example.
LLMs have an enormous amount of knowledge encoded in their parameters. However, LLMs:

- may lead to hallucination
- may not be up-to-date with their knowledge
- do not provide textual evidence to support their answer

RAG is the method that response is generated from LLM, conditioned by some retrivied passages, as shown in Figure.
More formally, we reduce the Q/A problem to the problem of computing the following probability...

10.1.4 Miscellanea

Several datasets for machine reading, containing tuples of the form (question, passage, answer).
Machine reading systems often evaluated using two metrics:

- **Exact match**: percentage of predicted answers that match the gold answer exactly
- For each question, treat prediction and gold as a bag of tokens:
  1. precision as the ratio of the number of shared words to the total number of words in the prediction
  2. recall as the ratio of the number of shared words to the total number of words in the ground truth
  3. $F1$ score as the harmonic mean of precision and recall

  and return average $F1$ over all questions

Answer sentence selection (AS2) is a task related to machine reading: given a question and a document, choose the sentence in the document that contains the answer fragment: using the document as context helps identifying the sentence with the correct answer.

10.2 Knowledge-based QA

Text-based QA uses textual information over the web; instead, **Knowledge-based QA** answers a natural language question by mapping it to a query over some structured knowledge repository.
There are two main approaches:

- **Graph-based QA**: models the knowledge base as a graph, with entities as nodes and relations as edges. For example the Google knowledge graph (now expired).
- **QA by semantic parsing**: maps queries to logical formulas, and queries a fact database, as we can see in the example below

Both approaches to knowledge-based QA require algorithms for entity linking.
Entity linking is the task of associating a mention in text with the representation of some real-world entity in a structured knowledge base.
The most common knowledge base for factoid question answering is Wikipedia,

in which case the task is called **wikification**.
Entity linking is done in (roughly) two stages: **mention detection** and **mention disambiguation**, using two approaches:

- classical approaches based on dictionaries and network structure
- modern approaches based on neural networks