UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

INFORMATION ENGINEERING DEPARTMENT

COMPUTER ENGINEERING - AI & ROBOTICS

Natural Language Processing 2023/2024

Francesco Crisci 2076739

Contents

Chapter 1

Introduction

1.1 General Introduction

Natural Language Processing is a field of artificial inteligneece that allows machines to read, derive meaning ffrom text, and produce documents. It works in the background of many services, from chatbots through virtual assistants to social media tracking. Such language technologies are already showing major penetration into the information and communication industry.
NLP distinguishes itself from other AI application domains, as for instance computer vision or speech recognition. Text data is fundamentally discrete. But new words can alway be created. Few words are fery frequent, and there is a long tail of rare words. Out-of-Vocabulary words are always being discovered.
Languages have the following characteristics:

- it is **ambiguougs:** units can have different meanings;
- it is **compositional:** meaning of a unit defined as a function of the meaning of its components.
- it is **recursive:** units cna be repeatedly combined.
- they unveils **hidden structure:** local changes in a sentence might have global effects.

Ambiguity

word can belong to several categories, like noun, verbs or modal. The word bank, for example, have different meanings: river bank or money bank. The morphological composition, e.g., the word un-do-able is ambiguous between not doable and can be undone.

Compositionality

At each level, meaning of a larger unit is provided by some function of the meaning of its immediate components and the way they are combined.

Recursion

The rules of the grammar can iterate to generate an infinite number of structures, each with its specific meaning. Recursion is considered the main difference between human and other animals' languages.

Hidden structure

Local changees can disrupt te interprettion of a sentence. This sugests the existence of hidden structure.

1.2 Search & Learning

Many natural language processing problems can be written mathematically in the form of optimization:

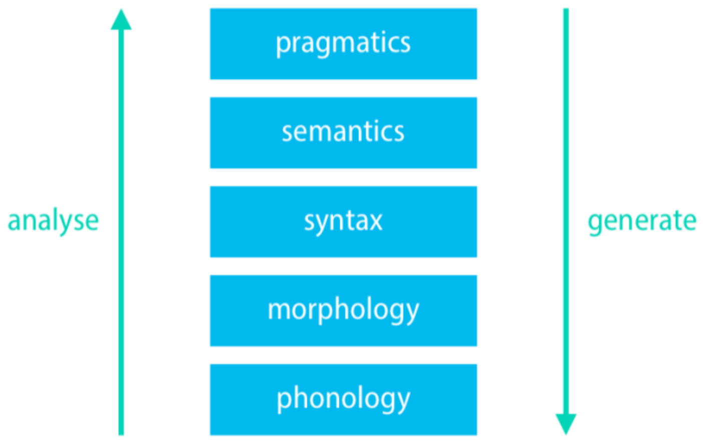
$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y}\in\mathcal{Y}(x)} \Psi(\mathbf{x}, \mathbf{y}; \boldsymbol{\Theta})$$

where x is the input, whicch is an element of a set $\mathcal{X}$; y is the output, which is an element of a set $\mathcal{Y}$(x). Ψ is a scoring function, also called the **model**, which maps from the set $\mathcal{X} \times \mathcal{Y}$ to the real numbers; $\boldsymbol{\Theta}$ is a vector of **parameters** for Ψ. $\hat{y}$ is the predicted output, which is chosen to maximize the scoring function.
The **Search** module is responsible for finding the candidate output $\hat{y}$ with the highest score relative to the input x. the **Learning** module is responsible for finding the model parameters Θ that maximizes the predictive

performance.
Structured prediction is an umbrella term for supervised machine learning techniques that involves predicting structured objects, rather than scalar discrete or real values.

1.3 Essentials of linguistics

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. Besides form and meaning, another important subject of study for linguistics is how languages is used in context. Phonology studies the rules that organize patterns of sunds in human languages.



It is different from phonetics, which is concerned with the production, transmission and perception of sounds, without prir knowledge of the language being spoken.
Morphology is the study of how words are composed by morphemes, which are the smallest meaningful units of language. The structure of a word consists of several morphemes: one root or stem and zero or more affixes. Inflectional morhpology means that there is no chang ein the grammatical category. Derivational morphology means that there is a change in the grammatical category.
A language can be of the following types:

- Isolating: a language in which each word form consists typically of a single morpheme;
- Analytic: no inflection to indicate grammatical relationship, may still contain derivational morphemes;
- Synthetic: uses inflection or agglutination to express relationships within a sentence.

The term morphologically rich language refers to a language in which substantial grammatical information is expressed at word level.
what we have seen in our examples so far is concatenative morhpology: morphemes are placed one after the other. Some languages, as for instance semitic languages, are based on template morhpology.
Syntax studies the rules and constraints that govern how words can be organized into sentences. Differently from formal language theory, NLP systems need to be robust to input that does not follow the rules of grammar.

Part of speech

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

- distributionally: Kim saw the {elephant, movie, mountain, error } before we did.
- functionally: verbs=predicates; nouns=arguments; adverbs=modify verbs, etc.

Open class tags: noun, verb, adjective, and adverb. New words in the language are usually added to these classes.
Closed class tags: determines, prepositions, conjunctions, etc. Closed word classes rarely receive new members.
There are several representations for syntactic structure. Most common are Phrase structure and Dependency tree.

Phrase structure is a tree-like representation with leaf nodes representing sentence words and internal nodes representing word grouping called **phrases.**
Dependency tree is a tree-like representation where the nodes represent words and punctuation in the sentence, and arc represent grammatical relations between a **head** and a **dependent.** Dependency trees use labels at arcs, representing grammatical relations. Arcs in a dependency tree are often called **dependencies**.
Semantics is the study of the meaning of linguistic expressions such as words, phrases and sentences. The focus is on what expressions conventionally/abstractly mean, rather than on what they miggtht mean in a particular context. The linguistic study of word meaning is called **lexical semantics.** The internal semantic structure of a word refers to the similarity with other words. The external semantic structure of a word refers to the allowability to combine with other words.
Lexical ambiguity arises because a word can have different meanings, called word senses.
Principle of compositionality: the meaning of a whole expression is a function of the meanings of its parts and of the way they are syntactically combined. Syntax provides the scaffolding for semantic composition: the meaning of a sentence isn't just the amalgamation of the meaning of its component words.
Meaning representation Many representations for semantic structure. Most common are logical, prediccate-argument or graph structure.
Pragmatics studies the way linguistic expressions with their semantic meanings are used for specific communicative goals. In contrast to semantics, pragmatics explicitly asks the question what an expression means in a given context. An important concept in pragmatics is the speech art, which describes an action performed through language.

Discourse analysis

Discourse analysis studies written and spoken language relation to its social context. **Discourse** refers to a piece of text with multiple sub-topics and ccoherence relations between them, such as explanation, elaboration andd contrast. There are many formalisms for representing discourse structure, as for example disccourse representation theory and rhetorical structure theory. **Dialogue** is a cooperative kind of discourse, where two or more participants are involved.

1.4 Text Language Processing

Cleaning up a text ddata set requires the use of specialized **regular expressions.** Most programmingg languages have facilities for compiling regular expressions into efficient finite automata and running these automata on input text. Regular expressions come in many variants. We describe here the so-called **extended** regular expressions. We have different categories and operators:

- **Search:** $/d/$ matches d in woodchuck.
- **Sets of characters:** $/[abc]/$ matches a, b or c; $/[\hat{}abc]/$ matches any character other than a, b and c; $/[a-z]/$ matches any character from a to z; $/./$ matches any character; also called **wildcard**.
- **Aliases:** $\backslash d$ stands for any digit; same as $[0-9]$; $\backslash D$ stands for any character other than a digit, same as $[\hat{}0-9]$; $\backslash w$ stands for any alphanumeric or underscore; $\backslash W$ is the converse of $\backslash w$; $\backslash s$ stands for any whitespace character; $\backslash S$ converse of $\backslash s$; $\backslash.$ stands for period; $\backslash n$ stands for newline.
- **Repetition:** $/\ d\{2,5\}/$ matches between 2 and 5 digits; $/\backslash d\{3,\}/$ matches 3 or more digits; $/\backslash d\{4\}\backslash w?/$ matches exactly 4 digits and one optional alphanumeric or underscore; $/[a-z]+/$ matches one or more lowercase letters; $/\backslash s+java\backslash s+/$ matches java with one or more whitespace characters before and after; $/[\hat{}\backslash+]*/$ matches zero or more characters other than + (Kleen star).
- **Anchors:** $/\hat{}/$ matches beginning of input string or line; $/\$/$ matches end of input string or line; $/\backslash b/$ matches word bondary; $/\hat{}The$ matches occurence of The at the beginning of a string. With $|$ you can have dijunctions and groups $/the|any$ matches the or any; $/grupp(y|ies)/$ matches gruppy and gruppies.
- **Substitute and Back-reference:** we can replace matches of a regular expression by a given pattern using the substitute operator $s/word1/word2$; we can use matches of a regular expression through the back-reference operator.

Words

We need to disstinguish between words and punctuation marks. Punctuation is critical to find sentence boundaries and for identifying some aspects off meaning. There are two ways of talking about words: **types** are the distinct words appearing in a document and **tokens** that are the individual occurences of words in a document. the set of all types in a corpus is the **Vocabulary** V. The vocabulary size $|V|$ is the number of types in the corpus. The size of the corpus N is the number of tookens, if we ignore punctuation marks. In very large corpora, the relation between N and $|V|$ can be expressed as

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

In very large corpora, the r-th most frequent type has frequency f(r) that scales acccording to

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

The **word-form** is the full inflecected or derived form of the word. Eacch word-form is associated with a single **lemma**, the citation form used in dictionaries. alternatively to $|V|$, another measure of the number of words in a corpus is the number of lemmas.

Corpora

Corpus: large collection of text, in computer-readable form. Several dimensions of varition for corpora should be taken into account: language, genre, etc.
Languag: it is important to tet NLP algorithms on more than one language. languages lacking large corpora are considered low-resource languages.
Genre: text documents might come from newswire, fiction, scientific articles, Wikipedia, etc.
Time: language changes over time. for some languages we have good corpora of texts from different historical periods.
Collection process: how big is the data and how was it sampled? How was the data pre-processed, and what metadata is available?
Annotation: what are the specifics of the used annotation? How was the data nnotated? How where the annotators trained?

Text Normalization

Text normalization is the process of transforming a text into some predefined standard form. This consists of seveeral tasks. There is no all-purpose normalization proccedure: text normalization depends on what type of text is being normalized and what type of NLP task needs to be carried out afterwards. Text normalization is also important for applications other than NLP, such as text mining and WEB search engines.

Language identification

Language identification is the task of detecting the source language for the input text. Several statistical techniques for this task: functional word frequency, N-gram language models, distance measure based on mutual information, etc.

Spell checker

Spell checkers correct grammatical mistakes in text. They use approximate string matching algorithms such as **Levenshtein distance** to find correct spellings.

Contractions

The following should be managed before further normalization: contracted forms, abbreviations and slangs. The most straightforward techniques is to create a dictionary of contractions and abbreviations with their corresponding expansions.

Punctuation

Punctuation marks in text need to be isolated and treated as if they were separatee words. This is critical for finding sentence boundaries and for identifying some aspects of meaning.

Tokenization

Tokenization is the process of segmenting text into units called tokens. Tokenization teccniques can be grouped into three families: word, character and subword tokenization. Tokens are then organized into a vocabulary and, depending on the specific NLP application, may later be mapped into natural numbers.

Word tokenization

Word tokenization is a very common approach for European languages. For english, most of the text is already tokenized after previous steps, with the following important exceptions: special compound names and city names.

Character tokenization

Major east Asian languages write text without any spaces between words. For most Chinese NLP task, character tokenization works better than word tokenization: each character generally represents a single unit of meaning and word tokenization results in huge vocabulary, with large number of rare words.

Subword tokenization

Many NLP systems need to deal with **unknown words**, that is, words that are not in the vocabulary of the system. To deal with the problem of unknown words, modern tokenizers automatically induce sets of tokens that include tokens smaller than words, called **subwords**. Subword tokenization reduces vocabulary size, and has become the most common tokenization method for large language modelling and neural models in general. Subword tokenization is inspired by algorithms originally developed in information theory as a simple and fast form of data compression alternative to Lempel-Ziv-Welch.
Subword tokenization schemes consist off three diffferent algorithms:

- the **token learner** takes a raw training corpuss and induces a set of tokens, called **vocabulary**.
- the **token segmenter** takes a vvocabulary and a raw sentence, and segments the sentence into the tokens in hte vocabulary.
- the **token merger** takes a token sequence and reconstructs the original sentence.

Three algorithms are widely used for subword tokenization: byte-pair encoding, unigram and WordPiece tokenization.

BPE: learner

The BPE **token learner** is usually run inside words, not merging acoss word boundaries. To this end, use a special end-of-word marker. The algorithm iterates through 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 voocabulary reaches size k, a hyperparameter.

BPE: Encoder

Two versions of **BPE token segmenter**: apply merge rules in frequency order all over the data set, or, for each word, left-to-right, match longest token from vocabulary. Encoding is computationally expensive. Many systems use some form of caching: pre-tokenize all the words and save how a word should be tokenized in a dictionary; when an unknown word is seen: applu the encoder to tokenize the word and add the tokenization to the dictionary for future reference.

BPE: Decoder

BPE token merger: to decode, we have to concatenate all the tokens together to get the whole word and use the end-of-word marker to solve possible ambiguities.

WordPiece

WordPiece is a subword tokenization algorithm used by the large language model BERT. Like BPe, WordPiecce starts from the initial alphabet and learns merge rules. The main difference is the way pair A, B is selected to be merged:

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

The algorithm prioritizes the merging of pairs where the individual parts are less frequent in the vocabulary.
Text normalization also includes **sentence segmentation:** breaking up a text into individual sentences. This can be undone using cues like periods, question marks, or exclamation points.
Lower casing is very useful to standardize words and before stop word removal. Most programming languages have facilities for string lowercasing.
Stop word removal includes getting rid of common articles, pronouns, prepositions and coordinations. Stop word removal heavily depends on the task at hand, since it can wipe out relevant information.

Stemming refers to the process of slicing a word with the intention of removving affixes. Stemming is problematic in th elinguistic perspective, since it sometimes produces words that are not in the langage, or else words that have a different meaning.
Lemmatization has the objeccctive of reduccing a word to its base form, also called **lemma**, therefore grouping together different forms of the same word. Lemmatization and Stemming are mutually exclusive, and the former is much more resource-intensice than the latter.

Chapter 2

Word Embeddings

What is the meaning of a word? The linguistic study of word meaning is called lexical semantics. A model of word meaning should allow us to relate different words and draw inferences to address meaning-related tasks. A word-form can have multiple meanings; each meaning is called a word sense, or sometimes a synet. 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 are synonyms if they have a common word sense. Two words are similar if they have similar meanings. Two words are related if they refer to related concepts. Two word are antonyms if they define a binary opposition. one word is an hyponim of another if the firs has a more specific sense. notions of hypernym or hyperonym are define symmetrically. Words can have affective meanings implying positive or negative connotations/evaluation.

2.1 Distributional semantics

Distributional semantics develops methods for quantifying semantic similarities between words based on their distributional properties, meaning their neighboring words. The basic idea lays in the so-called distributional hypothesis: linguistic items with similar distribution 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.
Lexical semantic relationship, represented as word vector differences, aare generally preserved.

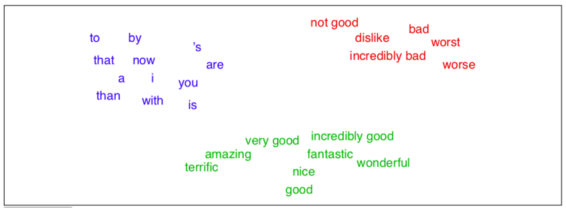


Figure 2.1: Example: Vectors from sentiment analysis application, projected into 2-dimentional space.

The obtained vectors are called word embeddings. The approach is called vector semantics and is the

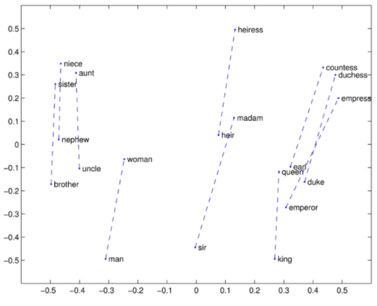


Figure 2.2: Example: Vectors projected into 2-dimentional space.

standard way to represent word meaning in NLP. All the learning algorithms we report are unsupervised. 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 optimization or approximation process.

There is a review of vectors and vector's opertions which i'm gonna omit. You are at the fifth year of engineering. If you need a review, i'm sorry, but you dumb :)

2.2 Term-context matrix

Let $\{w_1, ..., w_W\}$ be a set of **term** words and let $\{c_1, ..., c_C\}$ be a set of **context** words. For each word w_i in the corpus, we consider context words appearing inside a window of size L, at the left or at the right of w_i. A **term-context matrix** F is a matrix $W \times C$, where each element $f_{i,j}$ is the number of times context word c_j appears in the context of term word w_i in the corpus.

2.3 Probabilities & information theory

Pointwise mutual information is a measure of how often two events x and y occur, compared with what we would expect if they were independent:

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

It is simmetric, it can take positive or negative values. If the randome varaibles are independent then it is zero. The PMI between 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 tw words co-occur than we expected by chance. We can rewrite the PMI as:

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

PMI may then be understoodd as quantifying how much our confidence in the outcome w_i increases after we observe c_j.

2.4 Probability estimation

How do we estimate the Probabilities $P(w_i, c_j), P(w_i), P(c_j)$? We can estimate:

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

Note that the denominator is the total number of possible target/context pairs.

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

Similarly we can estimate:

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

It is difficult to estimate negative values of PMI. We can use **Positive PMI**:

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

2.5 Practical issues

PMI has the problem of being **biased** towards infrequent events: very rare words ttend to have very high PMI values. One way to reuce this bias is to slightly change the computation for $P(w_c)$ in the PPMI:

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

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

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, and 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 dense word embeddigns from th ePPMI term-context matrix. This is done through a matrix decomposition technique known as **truncated singular value decomposition**.

2.5.1 Truncated singular value decomposition

For a matrix $A \in \mathbb{R}^{n \times m}$, the **Frobenius norm** is

$$||A||_F = \sum_{i=1}^{n} \sum_{j=1}^{m} a_{i,j}^2$$

where $a_{i,j}$ are the elements of the matrix. Let $|V|$ be the vocabulary size, and let $K << |V|$ be the size of the desired embedding. Let $P \in \mathbb{R}^{|V| \times |V|}$ ne the PPMI term-context matrix. Let $U, V \in \mathbb{R}^{|V| \times K}$ be learnable parameters. The basic idea is to approximate P by means of a matrix $\tilde{P}(U, V) \in \mathbb{R}^{|V| \times |V|}$ such that

$$\min_{U,V} ||P - \tilde{P}(U, V)||_F$$

Truncated singular value decomposition solves the following constrained optimization problem:

$$\min_{U \in \mathbb{R}^{|V| \times k}, S \in \mathbb{R}^{K \times K}, V \in \mathbb{R}^{|V| \times K}} ||P - USV^T||_F$$

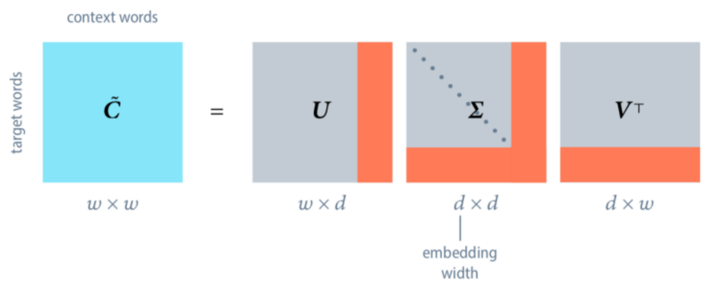
subject to conditions $U^T U = \mathbb{I}, V^T V = \mathbb{I}$ and S is diagonal with its elements in decreasing order. U are the target embeddings, SV^T are the context embeddings.
Let $U_k, V_k \in |\mathbb{V}| \times \mathbb{K}$ be the k-th columns of matrices U and V, respectively. In truncated SVD, P is approximated by the sum of K rank-1 matrices:

$$P \approx \sum_{k=1}^{K} s_k U_k V_k^T$$

Equivalently, each element $p_{i,j}$ of P is approximated through the dot product between the embedding of the i-th target word and the embedding of the j-th context word, weighted by the singular values

$$p_{i,j} \approx \sum_{k=1}^{K} s_k u_{i,k} v_{j,k}$$

We can alternatively view the previous approximation approacch as a factorization problem for matrix P, $P = USV^T$, where $U, S, V \in |\mathbb{V}| \times |\mathbb{V}|, U^T U = \mathbb{I}, V^T V = \mathbb{I}$ and S is a diagonal with **singular values** $s_1 > s_2 > \ldots > s_V$ in its diagonal. We then choose $K << |V|$ on the basis of mass distribution at the singular values, and truncate matrices U,S,V accordingly.



2.6 Neural word embeddings

Word embeddings can also be computed using neural networks. Generally speaking, neural word embeddings have the following properties: small number of dimension, vectors are dense, the dimensions don't have a clear interpretation, components can be negative. Neural word embeddings work better than previous embeddings in every NLP task.
Word2vec is a software package including two different algorithms for learning word embeddings: skip-gram with negative sampling (SGNS) and continuous bag-of-words (CBOW). The idea is that instead of counting how oftne a context word appears near a target word, we train a classifier on the following binary prediction task: *Is a giiven context word likely to appear near a given target word?* We don't really care about this prediction task: intead, we use the learned parameters as the word embeddings.

2.6.1 Skip-Gram

More specifically, the basic steps of the **skip-gram** algorithm are as follows:
For each target word $w_t \in V$:

I treat w_t andd any neighboring context wor w_c as positive examples;

II randomly sample other words $w_n \in V$, called **noise words**, to produce negative examples for w_t

Using **logistic regression** tto train a classfier to distinguish positive and negative examples. Use the learned weights as the embeddings.

2.6.2 Logistic Regression

For any pair of words $w_t, u \in V$, we need to define the probability that u is/is not a context word for target word w_t

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

For each word $w \in V$, we construct two complementary embeddings: a target embedding $e_t(w) \in \mathbb{R}^d$ and a context embedding $e_c(w) \in \mathbb{R}^d$. Integer d is the size of the embedding. Operator $e_t(w)$ is always assigned to target words, operator $e_c(w)$ is always assigned to context or noise words. We use the **dot product**:

$$e_t(w) \cdot e_c(w) = |e_t(w)||e_c(w)| \cos\theta$$

where θ is the angle between $e_t(w)$ and $e_c(w)$. We also use the **logistic sigmoid function** σ, which maps real values to probabilities. We can now define:

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

We then have vectors $e_t(w), e_c(w)$ with small angle: positive cosine similarity, $P(+|w, u)$ close to one, and vectors $e_t(w), e_c(w)$ with large angle: negative cosine similarity, $P(+|w, u)$ close to zero.

2.7 Training

For simplicity, let us consider a dataset with only one target/context pair (w,u), along with k noise words $v_1, ..., v_k$(negative examples). Skip-gram makes the simplifying assumption that all context words are independent. Then the log-likelihood of the data is:

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

Which we can rewrite as:

$$LL_w = \log \sigma(e_t(w) \cdot e_c(u)) + \sum_{i=1}^{k} \log(1 - \sigma(e_t(w) \cdot e_c(v_k)))$$

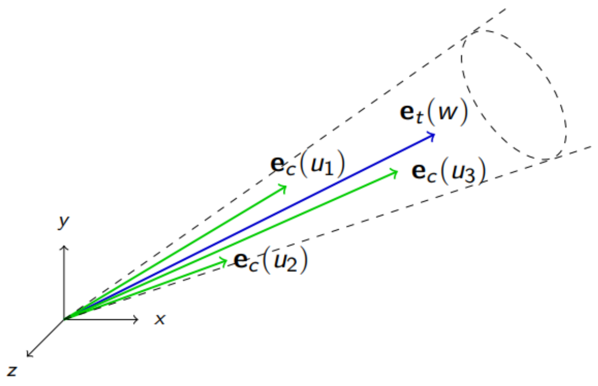
We can alternatively minimize the inverse of LL_w, which in ccaase of the logistic regression is the **cross-entropy** loss function:

$$L_{CE} = -\log \sigma(e_t(w) \cdot e_c(u)) - \sum_{i=1}^{k} \log(1 - \sigma(e_t(w) \cdot e_c(v_k)))$$

The model parameters are the two $\mathbb{R}^{|V| \times d}$ matrices with the target and context embeddings $e_t(w), e_c(w), w \in V$, which are randomly initialized. By making an update to minimize L_{CE} we force an increase in similarity between $e_t(w)$ and $e_c(w)$, and we force a decrease in similarity between $e_t(w)$ and $e_c(v_k)$, for all of the noise words v_k. We train the model with **stochastic gradient descent**, as usual for logistic regression. in the end, we retain the target embeddings $e_t(w)$ and ignore the context embeddings $e_c(w)$.

2.7.1 Geometric interpretation

Target vector $e_t(w)$ is surrounded by its context vectors $e_c(u_j)$ which form a thin hypercone:
Let w be a target word with context words $u_j's$. Let also w' be a target word similar to w. Then we have that $e_c(u_j)$ form a thin cone surrounding vector $e_t(w)$; vector $e_t(w')$ must be placed within that cone, because w' and w share their context word; $e_t(w)$ and $e_t(w')$ are force to be at a small angle. As a result, similar words cluster together. Within a similar reasoning, we can argue that different words repel each other.



2.7.2 Practical isseus

Clean up text: convert any punctuation into token $< PERIOD >$. Remove all words w such that $f(w) \leq D$: greatly redduces issues due to noise into data and improves the quality of the vector representation:
Indexing: convert words in V to non-negative integers and back again, this makes your code simpler.
Words that shouw up very often don't provide much context and are often regardes as noise. The process of discarding occurences of these qords by mean of some probability distribution is called subsampling. We discard occurence w_i with probability given by

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

where t is a threshold parameter depending on dataset size.
The rato k between the positive and the negative examples is a hyperparameter and must be adjusted on the development set. The noise words are chosen according to the relative frequency estimator and a α corrector:

$$P_\alpha(v) = \frac{C(v)^\alpha}{\sum_{w \in V}(C(w))^\alpha}$$

Once hyperparameter L is fixed, context window size can be chosen randomly in the range $[1, ..., L]$, with uniform distribution. this results in linear decay for context words, depending on distances from target word: words at distance 1 from target will be chosen with P=1, words at distance L from target will be chosen with $P0\frac{1}{L}$.
Should we estimate one embedding per word form or else estimate distinct embeddings for each word sense? Intuitively, if word representations are to capture the meaning of individual words, then words with multiple meanings should have multiple embeddings.

2.8 Miscellanea

2.8.1 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.

2.8.2 GloVe

GloVe (Global Vectors), 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.

2.8.3 Semantic properties

Both sparse and dense vectors are sensible to the context window size used to collect counts, controlled by hyperparameter L. The choice of L depends on the goals of the representation:

- small values of L provides semantically similar words with the same parts of speech.
- large values of L provides words that are tpically related but not similar.

another ssemantic property of embeddings is their ability to capture relational meanings. The parallelogram model can be used to solve analogy problems.

2.9 Evaluation

Extrinsic evaluation: use the model to be evaluated in some end-to-end application and measure performance.
Intrinsic evaluation: look at performance of model in isolation, w.r.t a given evaluation measure.
The most common evaluation metric for embedding models is extrinsic evaluation on end-to-end tasks: machine translation, sentiment analysis, etc. Three types of intrinsic evaluation:

- word **similarity**: compute a numerical score for the semantic similarity between two words;
- word **relatedness**: compute the degree of how much one word has to do with another words;
- word **analogy**: comparison of two things to show their similarities.

2.10 Cross-lingual word embedding

Cross.lingual word embeddings encode words from two or more languages in a shared high dimensional space. Vectors representing words with similar meaning are closely located, regardless of language.
Cross-lingual word embeddings are very useful in comparing the meaning of word across languages; are key to applications such as machine translation, multilingual lexicon induction, andd cross-lingual information retrieval; enable model transfer between resourse-rrich and low-resource languages, by rpoviding a common representation space.
The class of **objective functions** minimized by most cross-lingual word embedding methods can be formulated as:

$$J = \mathcal{L}^1 + ... + \mathcal{L}^\ell + \Omega$$

where $\mathcal{L}^i$ is the monolingual loss of the i-th language and Ω is a regularization term. Joint optimization of multiple non-convex losses is difficult: most approaches optimize one loss at a time, while keeping certain variables ffixed; this step-wise approach is approximate and does not guaranteee to reach even a local optimum.
The choice of multilingual parallel data sources is more important for the model performance than the actual underlying architecture. Method differ along the following two dimensions: type of text **alignment**: at the level of words, sentences, or documents, which introduce stronger or weaker supervision; Type of text **comparability**: extract translations, weak translations, or similar meaning.
Variation in ambiguity: ambiguity for word w in one language does not transfer to the translation of w in anoter language.

Chapter 3

Language Models

3.1 Language Modeling

Language modeling 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 wwant to know the probability of the next word w_{t+1}:

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

When the string is understood from the context, we write $w_{1:t}$.
Rather than as **predictive** models, language models can also be viewed as **generative** models that asign probability to a piece of text:

$$P(w_1 ... w_t)$$

These two view are **equivalent**, as the probability of a sequence can be expressed as a product of conditional probabilities:

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

Conversely, a conditional probability can be expressed as a ratio of two sequence probabilities:

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

Probability Estimation

Assume the text *its water is so transparent that*. We want to know the probability that the next word is *the*. Let C(its water is so transparent that) be the number of times the string is seen in a large corpus. One way to estimate the above probability is to set

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

In general, given a large corpus, we can set:

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

The quantities $C(w_{1:t})$ are called **frequencies**, the ratio $\frac{C(w_{1:t})}{C(w_{1:t-1})}$ is called **relative frequency**. The estimator above is therefore called **relative frequency estimator**. In practive, consider an aggressive upper bound of N=20 words in our sequence, and an English vocabulary V of size $|V| = 10^5$. Then the number of possible sequencces is $|V|^{20} = 10^{100}$. This estimator is extremely data-hungry, and suffers from **high varaince**: depending on what data happens to be in the corpus, we could get very different probability estimations.

3.2 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. The 2-gram model, for example, makes the approximation

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

The general equation for the N-gram model is

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

The relative frequency estimator for the N-gram model is Then

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

For N=2 we have:

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

For N=1 we have $P(w_t) = \frac{C(w_t)}{n}$ where n is the length of the training set. The model requires estimating and storing the probability of only $|V|^n$ events, which is exponential in N, not in the lenght of the sentence.

3.2.1 Learning

N is a hyperparameter. When setting its value, we face the **bias-variance tradeoff**. When No is to small, it fails to recover long-distance word relations. When N is too large, we get data sparsity.
The relative frequency estimator can be mathematiccally derived by maximizing the **likelihood** of the dataset. This can be done by solving a **constrained optimization problem**, using Lagrange multipliers. Therefore, the relative frequency estimator is also called the **maximum likelihood estimator** (MLE).

3.2.2 Practical issues

To compute 3-gram probabilities for words at the start of a sentence, we use two markers. Similarly for higher order N-grams. Multiplying many small probabilities results in **underflow**. it is mush safer and more efficient to use **negative log probabilities**, -log(p), and add them.

3.2.3 Evaluation

Extrinsic evaluation: use the model in some application and measure performance on that application.
Intrinsic evaluation: look at performance of model in isolation, with respect to a given evaluation measure.

Perplexity

Intrinsic evaluation of language models is based on the inverse probability of the test set, normalized by the number of words. For a test set $W = w_1w_2...w_n$ 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 multiplicative inverse probability $1/P(w_j|w_{1:j-1})$ can be seen as a measure of how surprising the next word is. The degree of the root averages overall words of the test set, providing **average surprise per word**. For large enough test data obtained in a uniform way, perplexity is mor or less constant, i.e. independent of n. The lower the perplexity, the better the model.

Sparse data

If there isn't enough data in the training set, counts will be zero for some grammatical sequences. Then some of the N-gram probabilities will be **zero** or **undefined**.

3.3 Smoothing

Smoothing techniques (also called **discounting**) deal with words that are in our vocabularry V but were never seen before in the given context. Smoothing prevents LM from assigning **zero probability** to these events. The idea is to shave off a bit of probability mass from more frequent events and givve it to the events we have never seen in the training set.

3.3.1 Laplace Smoothing

Laplace Smoothing does not perform well enough, but provides a useful baseline. The idea is to pretend that everything occurs once more than the actual count. In order to apply smoothing to our N-gram model, let us rewrite the relative frequency estimator in a more convenient form:

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

In the summation u is a single word ranging over the entire vocabulary V andd the sentence end marker.

Laplace for 1-grams

Let n be the number of tokens, that iss, the lenght of the training set, and call that $|V|$ is the number of word types. The adjusted estimate of the probability of word $w_t \in V$ is then:

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

The extra $|V|$ comes from pretending there are $|V|$ more observations, one for each word type.
Alternatively, we can think of P_L as applying an adjusted count C^* to the n actual observations

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

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

Under this view, the smoothing algorithm amouts to discounting (lowering) count for high frequency words and redistributing

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

We can consider the relative discount $d(w_t)$, defined as the ratio of the discounted counts to the original counts:

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

By solving $d(w_t) < 1$, we find that discounting happens for high frequency types u succh that $C(u) > \frac{n}{|V|}$.

Laplacce for 2-grams

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-grm $w_{t-1}w_t$ 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-gramm sequences that occur zero or few times in the training set. However, $P_L(w_t|w_{t-1})$ will be much lower ffor 2-graam sequences that occur often. So Laplace smoothing is too crude in practice.

Add-k smoothing

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

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

Jeffreys-Perks law corresponds to the case k=0.5, which works well in practice and benefits from some theoretical justification.

Smoothing and Perplexity

When smooting a language model, we are redistributing proobabiility mass to outcomes we have never observed. This leaves a smaller fraction of the probaibility 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.

3.4 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. These techniques prevent LM from creating **undefined probabilities** for these events.

3.4.1 Backoff

Backoff combines fine grained models with coarse graained models. The idea is simple: if you have a trigram use those; if not use bigrams; if you don't have neither, use unigrams.
Katz backoff is a popular but rather complex algorithm for backoff.

3.4.2 Stupid backoff

With very large text collections a rough approximation of Katz backoff is often sufficient, called **stupid backoff**. For some small λ:

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

3.4.3 Linear interpolation

In **simple linear interpolation**, we can combine different order N-grams by linearly interpolating all the models. Simple linear interpolation for N=3:

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

for some choices of possitive $\lambda_1, \lambda_2, \lambda_3$ such that $\sum_j \lambda_j = 1$.
What are good choices for the $\lambda_j' s$? Algorithms exist that attempt to optimise likelihood of training data. We might have different values for the $\lambda_j' s$, depending on sequences $w_{t-2}w_{t-1}$, subject to

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

3.4.4 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 fewe than d times in the training set, d some small number. Proceed to train LM as before, treating $< UNK >$ as a regular word. At the test time, replace all unknown words by $< UNK >$ and run the model.

3.4.5 Limitations

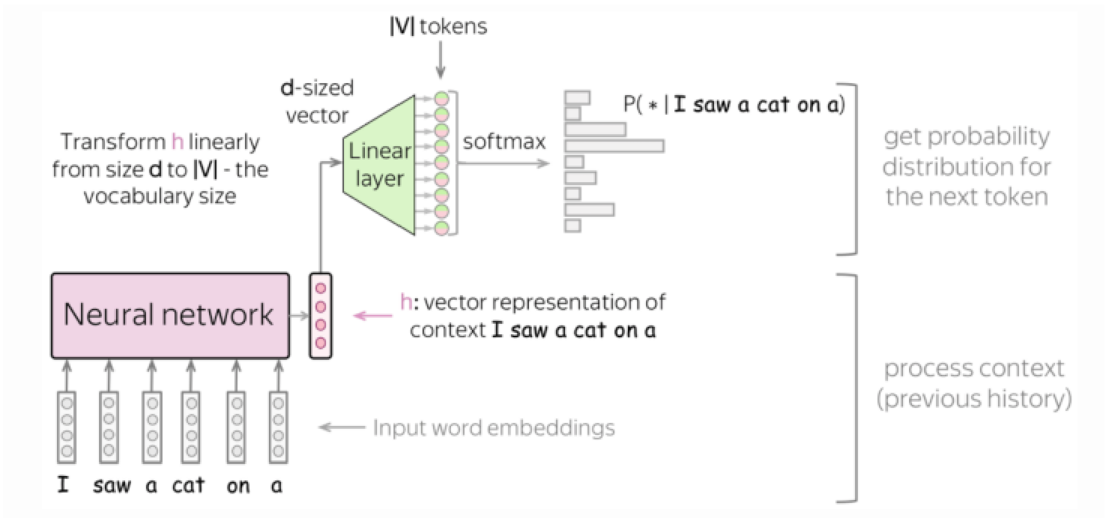
N-gram language models have several **limitations**. Scaling to larger N-gram sizes is problematic, both for computational reasons and because of increased sparsity. Smoothing techniques are intricate and require careful engineering to retain a well-defined probabilistic interpretation. Without additional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

3.5 Neural language models

N-gram language models have been largely supplanted by **neural language models** (NLM). Main advantages of NLM:

- can incorporate arbitrarily distant contextual information, while remaining computationally and statistically tractable
- can generalize better over contexts of similar words, and are more accurate at word-prediction

On the other hand, as compared with N-gram language models, NLM are much more complex, are slower, need more time to train, and are less interpretable.
The idea is to have a vector representation for the previous contet and to generate a probability distribution for the next token. Most natural choice for NN architecture is recurrent neural network but feedforward neural network and convolutional neural network have also been exploited.


|V| tokens
d-sized vector
Transform h linearly from size d to |V| - the vocabulary size
Linear layer
softmax
P(* | I saw a cat on a)
get probability distribution for the next token
Neural network
h: vector representation of context I saw a cat on a
process context (previous history)
Input word embeddings
I saw a cat on a

3.5.1 Feedforward NLM: inference

Like the N-gram language model, the feedforward 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. For $w \in V$, let $ind(w) \in [1..|V|]$ be the index associated with w.
We represent each input word w_t as a one-hot vector x_t of size $|V|$, defined as follows: element $x_t[ind(w_t)]$ is set to one, and all the other elements of x_t are set to zero.
At the first layer we convert one-hot vectors for the words in the N-1 window into word embeddings of size d and we concatenate the N-1 embeddings. The first hidden layer equation is(assuming N=4):

$$e_t = [Ex_{t-3}; Ex_{t-2}; Ex_{t-1}]$$

where E is a $d \times |V|$ is a learnable matrix with the word embeddings, x_{t-i} is $|V| \times 1$ are 1-hot representation of word w_{t-i} and e_t is a $3d \times 1$ is the concatenation of the embeddings of the N-1 previous words.
The model equations for the remaining layers:

$$h_t = g(We_t + b), \qquad z_t = Uh_t, \qquad \hat{y}_t = \text{softmax}(z_t)$$

where W is a $d_h \times 3d$ learnable matrix, with d_h the size of the second hidden representation, b is a $d_h \times 1$ learnable vector, h_t has dimension $d_h \times 1$, obtained through some activation function g, U is a $|V| \times d_h$ learnable matriz, and $z_t, \hat{y}_t$ are $|V| \times 1$ scores and distribution.
The vector z_t can be thought of as a set of scores over V, also called logits: rwa predictions that a classification model generates. Passing these scores through the softmax function normalizes into a probability distribution. The element of $\hat{y}_t$ with index $ind(w_t)$ is the probability that the next word is w_t:

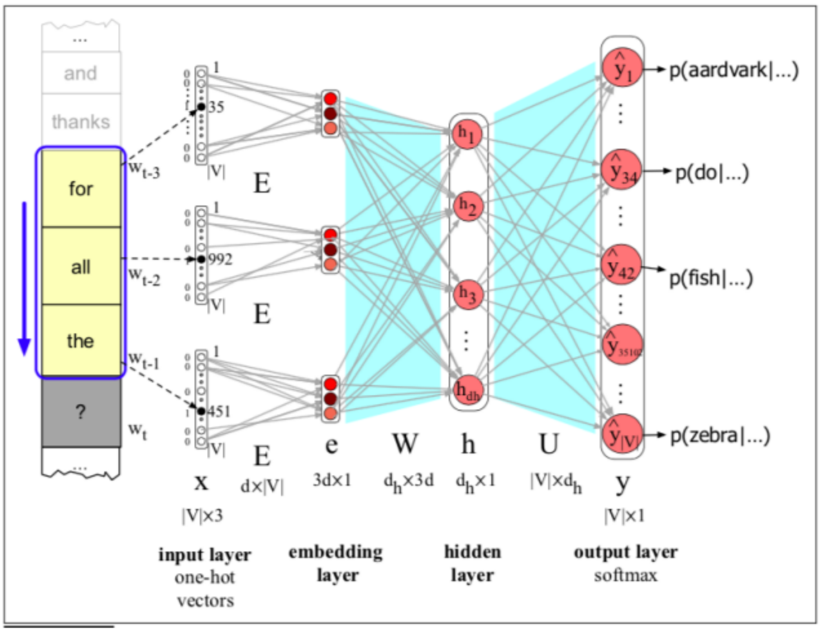
$$\hat{y}_t[ind(w_t)] = P(w_t|w_{t-3:t-1})$$

The parameters of the model are $\theta = E, W, U, b$. Let w_t be the word at position t in the trainign data. Then the true distribution y_t for the words at position t is a 1-hot vector of size $|V|$ with $y_t[ind(w_t)] = 1$ and $y_t[k] = 0$ everywhere else. We use the cross-entropy loss for training the model:

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

Recall the equation for the estimated distribution:

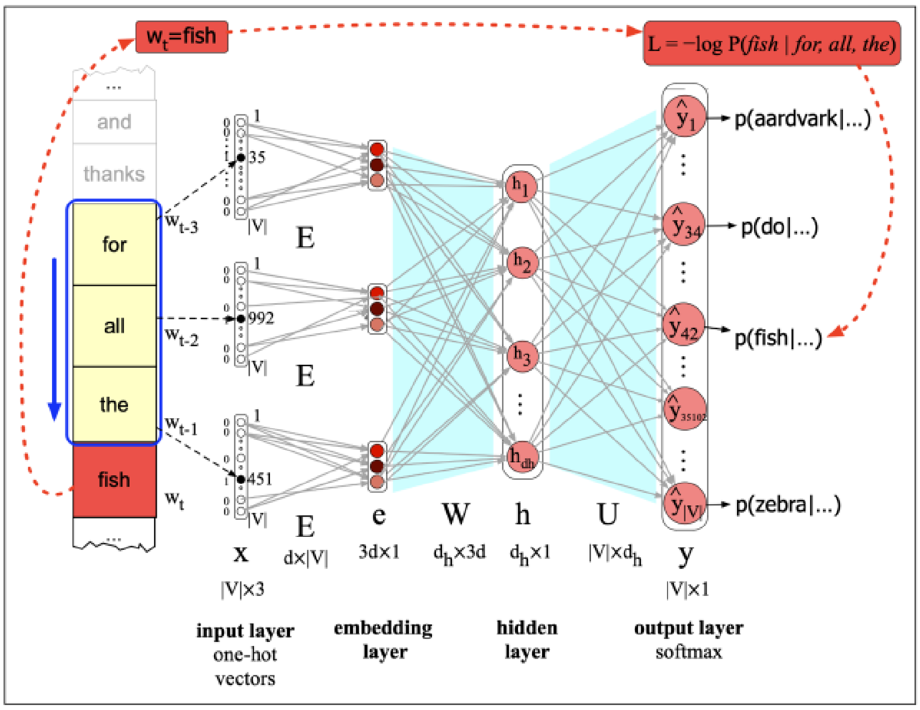
$$\hat{y}_t[ind(w_t)] = P(w_t|w_{t-N+1:t-1})$$



Replacing in the cross-entropy loss equation, we obtain:

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

Observe that the cros-entropy loss equals the negative log likelihood of the training data.



3.5.2 Recurrent NLM

RNN language models process the input one word at a time, predicting the next word from the ccurrent word an the previous hidden state. RNNs can model probability distribution $P(w_t|w_{1:t-1})$ without the N-1 window approximation of feedforward NLM. The model quations are:

$$e_t = Ex_t, \qquad h_t = g(Uh_{t-1} + We_t), \qquad \hat{y}_t = \text{softmax}(Vh_t)$$

where x_t is a $|V| \times 1$ 1-hot representation of word w_t, E is a $d_h \times |V|$ learnable matrix with the word embeddings, U and W are $d_h \times d_h$ learnable matrices, h_t is a $d_h \times 1$ hidden vector at step t, V is a $|V| \times d_h$ learnable matirx

and $\hat{y}_t$ is a $|V| \times 1$ probability distribution.
The vector resulting from Vh_t records the logits over the vocabulary V, given the evidence provided by h_t. The softmax normalizes the logits, resulting in the estimated distribution $\hat{y}_t$ for the word at step t. More precisely, for each word $w \in V$, the element of $\hat{y}_t$ with index ind(w) estimates the probability that the next word is w:

$$\hat{y}_t[ind(w)] = P(w_{t+1} = w|w_{1:t})$$

The number of parameters of the model is $O(|V|)$, since d is a constant. Let y_t be the true distribution at

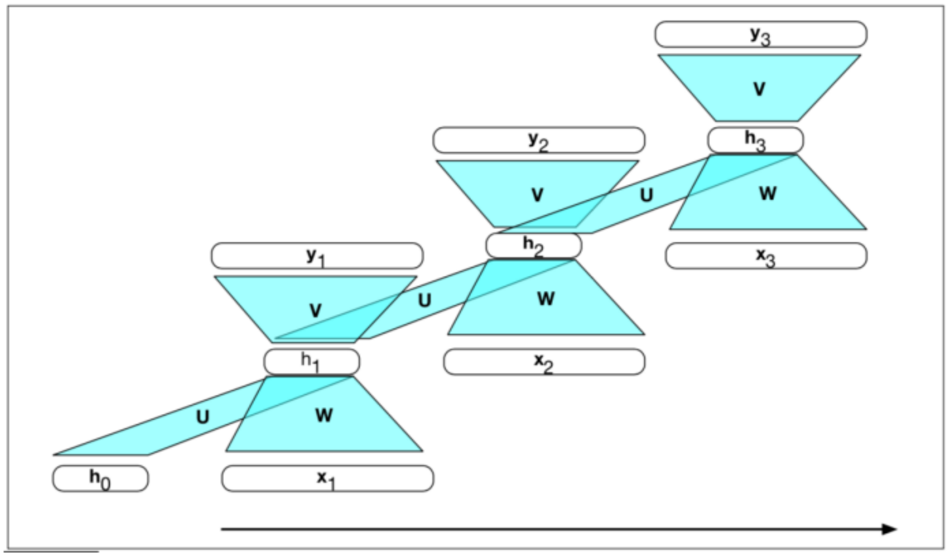


Figure 3.1: The recurrent NLM unrolled in time

step t. This is a 1-hot vector over V, obtained from the training set. We train the model to minimize the error in prediccting the true next word w_{t+1} in the training sequence, using cross-entropy as the loss ffunction:

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

At each step t durring training the prediction is based on the correct sequence of tokens $w_{1:t}$ and wwe ignore what the model predicted at previous steps. The idea that we always give the model the correct history sequence to predict the next word is called teacher forcing. Teacher forcing hase some disantantages: the model is never exposed to prediction mistakes; therefore at inference time the model is not able to recover from errors.

Character-level NLM

Characcter-level NLM improves modeling of uncommon and unknown words and reduce training parameters due to the small softmax. Performance usually worse than the word-level NLMs, since longer history is needed to predict the next word correcctly. A variety of solutions that combine character and word level information have been proposed, called character-aware LM.

3.5.3 Practical issues

Both the feedforward NLM and the recurrent NLM learn word embeddings E simultaneously with training the network. Alternatively, one can resort at freexzing: use pretrained word embeddings, for istance word2vec and hold E constant while training, and modify the remaining parameters in θ.
In the recurrent NLM model the columns of E provide the learned word embeddings, while the row of V provide a second set of learned word embeddings, that capture relevant aspects of word meaning and function.
Weigth tying, also known as parameter sharing, means that we impose $E^T = V$. Weigght tying can significantly reduce model size, and has an effect similar to regularization, preventing overfitting of the NLM.
RNNs suffer from the vanishing gradient problem: past events have weights that decrease exponetially with the distance from actual word w_t. Gated recurrent units (GRU) and long-short term memory (LSTM) neural networks are much better in capturing long distance relations.
The last step in NLMs, involving softmax over the entire vocabulary, dominates the ccomputation both at training and at test time. An effective alternative is hierarchical softmax, based on word clustering. Adaptive softmax is a simple variant of the hierarchical softmax, based on Zipf's law, especially tailored for GPUs.

The text generated by sampling our NLM should be **coherent**: text has to make sense; and **diverse**: the model has to be able to produce very different samples. A very popular method for modifying language model behavior is to change the **softmax temperature** τ:

$$\frac{\exp(\frac{V_i h_t}{\tau})}{\sum_j \exp \frac{V_j h_t}{\tau}}$$

where V_i denotes the i-th row of V.
Contrastitive evaluation is used to test specific linguistic constructions in NLM.

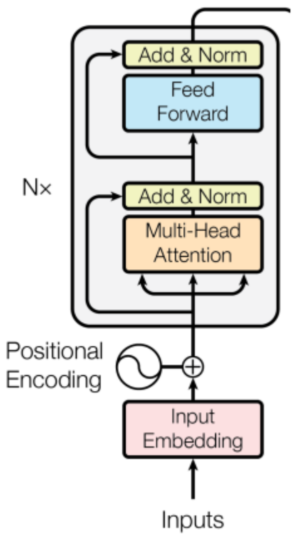
Chapter 4

Large Language Models

4.1 Transformer

The transformer is a neural network architecture modeling equencce to sentece transformation: it turns one sequence into another sequence. The original transformer model uses an encoder-decoder architecture. The transformer computes representations of its input and output entirely based on self-attention, without relying on sequential computation.

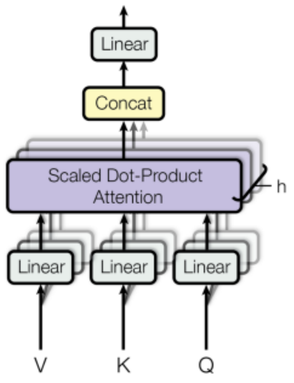
4.1.1 Encoder



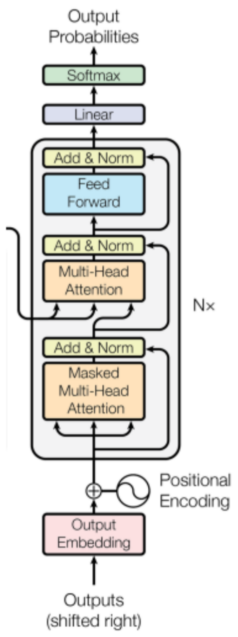
Thee Encoder has a tokenizer and an embedding layer with positional encoding. It has a stack of N identical blocks with two layers each. The first layer is a multi-head self-attention mechanism. The second layer is a fully connected feed-forward network. Bboth layers employ residual connection and normalization.
The encoder uses so-called scaled dot-product attention. Attention is computed on the basis of three different roles for the input tokens, produced through linear transformation. Query is the current focus in comparison, key is the importance to the given query, value is the contribuion to the given query. Query-key product needs to be scaled for gradient stability.

4.1.2 Multi-head attention

The input tokens can relate to each other in many different ways simultaneously. Multi-head attention implements parallel layers of self-attention that reside at the same depth.
Residual connection allows information to skip a layer, improvving learning. Layer normalization improves performance by keeping the values of a hidden layer in a range that facilitates graddient-based training.
Feed Forward: a simple MLP applied to each token individually; uses GeLU activation function.
The input length iss fixed, since attention is quadratic. Encoder output is a high level, contextualized version of the input tokens, serving as a basis for decoder computation.



4.1.3 Decoder



It has a stack of N identical blocks with three layers each. the first layer as in encoder, but with masked attention (auto-regressive). The second layer uses cross-attention (key and value of encoder stack). the thrid layer as in encoder. All layers employ residual connecction and normalization.
At training time, decoder uses masked self-attention. In this way, ech token only sees the alreaddy generated ones andd computation can be carried out in parallel. At generation time we need to generate one token at a time. This is why autoregressive decoding is extremely slow.
Moving layer normalization before multi-head attention and feedforwar layers stabilizes training. Fixed sinusoidal position embeddings sometimes replaced by unique, learned embeddings per position. Some applications use encoder-only, or else decoder-only components.

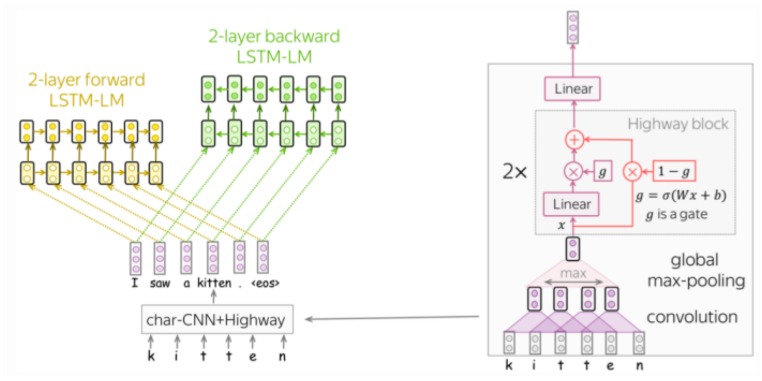
4.2 Contextualized Embeddings

Word embeddings such as word2vec or GloVe learn a single vector for each type in the vocabulary V. These are also called static embeddings. By contrast, contextualized embeddings represent each token by a different vector, acccording to the context the token appears in. Incorporating context into word embeddings has proven to be a watershed idea in NLP, opening the way to transfer learning. How can we learn to represent words along with the context they occur in? we train a neural network combining ideas from word embeddings an language model:

- predicct a worrd from left context
- predict a word from left and right context independently
- predict word from left and right context jointly

4.2.1 ELMo

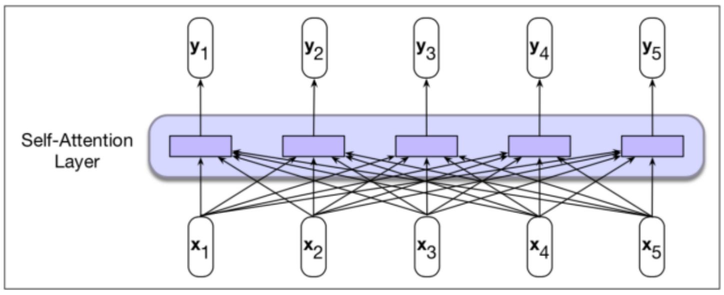
ELMo looks at the entire sentence before assigning each word its embedding. Tokens are processed by a character-level convolutional neural network producing word-level embeddings. These embeddings are processed by a left-tto-right and a right-to-left, 2-layers LSTM. For each word, the output embeddings at each layer are



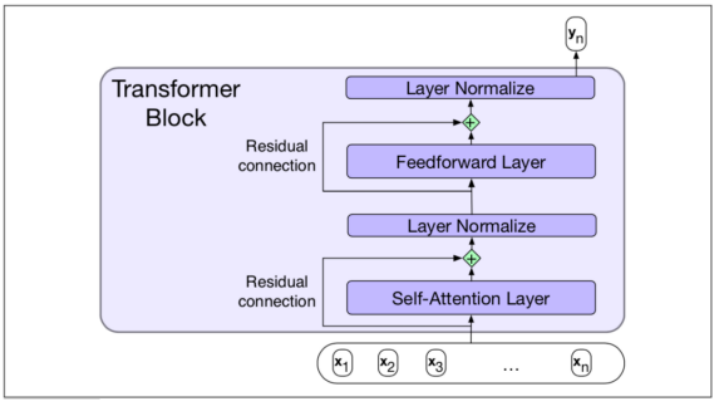
combined, producing contextual embeddings. When training, add output layer with linear transformation to $|V|$ dimension and softmax, and use cross-entropy as in neural language models. When encoding the input ignore the output layer and retain the word embedding represented by the last hidden layer.

4.2.2 BERT

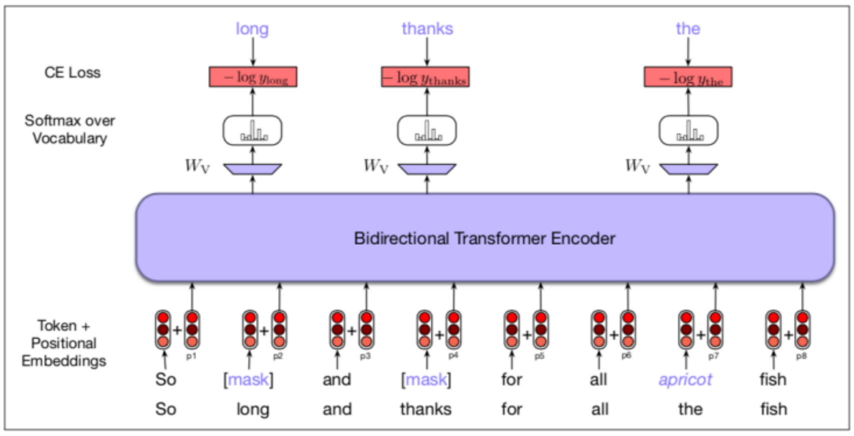
BERT produces word representations by jontly conditioning on both left and right context. The model is based on the encoder omponent of the transformer neural network. Tokenizer uses WordPiece algorithm. The bidirectional encoding in BERT is inherited from the self-attention mechanism of the transformer encoder. In-



put is segmented using wubword tokenization and combined with positional embeddings. Then input is passed through a series of standard transformer block consisting of self-attention and feedforward layers, augmented with residual connection and layer normalization. The model learns to perform a fill-in-the-blank task, tech-



nically called the cloze task. The approach is called masked language modeling. A random sample or 15% of the input tokes are replaced with the unique vocabualry token $[MASK]$ with 0.8 probability, replaced with another token randomly sampled with unigram probabilities with probability 0.1, or left unchanged with probability 0.1.


long
thanks
the
CE Loss
$-\log y_{\text{long}}$
$-\log y_{\text{thanks}}$
$-\log y_{\text{the}}$
Softmax over Vocabulary
W_V
Bidirectional Transformer Encoder
Token + Positional Embeddings
So [mask] and [mask] for all apricot fish
So long and thanks for all the fish

Next sentence prediction

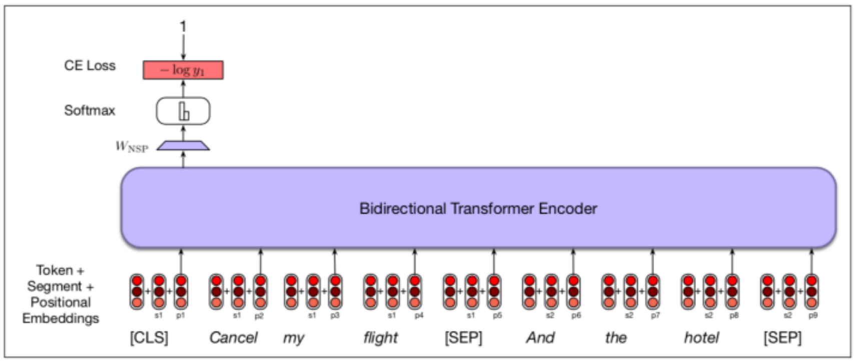
Many important downstream tasks involve relationship between pairs of sentences:

- paraphrase detection: detecting if two sentences have similar meanings, also called paraphrase identification.
- semantic textual similaritty: detecting the degree of semantic similarity between two sentences.
- sentence entailment: detecting if the meaning 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 understands sentencce relationships, BERT introduces a second learning objective called next sentece prediction. NSP is a binary decision task that can be trivvially generated from any monolingual corpus. NSP produces sentence embeddings that can be used for tasks involving relationship between pair of sentences. In NSP, the model is presented with pairs of sentences with:

- token $[CLS]$ prepended to the first sentence.
- token $[SEP]$ placed between the two sentences and after the rightmost token of the second sentence.
- segment embeddings making the first and second sentences are added to each word and positional embeddings.

Each pair consist of an actual pair of sentences from the training corpus and a pair of unrelated sentences randomly selected. The output embedding associated with the $[CLS]$ token reprensets the next sentence predicition.


1
CE Loss
$-\log y_1$
Softmax
W_{NSP}
Bidirectional Transformer Encoder
Token + Segment + Positional Embeddings
[CLS] Cancel my flight [SEP] And the hotel [SEP]

4.2.3 Practical issues

The result of the MLM and NSP pre-training processes consits of the encoder, which is used to produce contextual embeddings for tokens in novel sentences. It is common to compute a representation for word token w_t by averaging the output embeddings at t from each off the topmost layers of the model. $[CLS]$ embedding also used as sentence embedding for tasks involvving single sentence classification.

4.3 Large Language Model

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

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

LLMs are at the basis of modern applications of **generative artificial intelligence**.

4.3.1 GPT family

GPT-2

GPT-2 is a left-to-right language model. Based on transformer's decoder-only, no cross-attention layer. Layer normalization moved to the input of each sub-block. Tokenization: BPE with vocabulary size from 32,000 to 64,000.

GPT-3

GPT-3 has the same basic architecture as GPT-2: transformer's decoder-only, including pre-normalization. The model adds several optimization techniques, includin **attention sparsity** at half of the layers, computing only a subset of the elements of the attention matrix. GPT-3 has been trained on Common Crawl dataset, WebText dataset, book corpora and English-language Wikipedia. Because of such a large textual collection, novel capabilities of LLMs emerge, collectively called **in-context learning**.
Few-shot learning is the ability of an LLM to solve a problem on the basis of a few examples. **Zero-shot learning** is the ability of an LLM to solve a new problem solely on the basis of natural language instructions. Using few-shot and zero-shot settings, evaluation has been carried out on question answering, machine translla-tion, reading comprehension and common sense reasoning. An enhanced version of GPT-3 named GPT-3.5 has been used as a basis for chatBot chatGPT.

GPT-4

GPT-4 is a multimodal large language model. The model is capable of processing image and text inputs and producing text outputs. The model was then fine-tuned using Reinforcement Learning from Human Feedback. Two version with context windows of 98,192 and 32,768 tokens. On a suite of traditional NLP benchmarks, GPT-4 outperforms most state-of-the-art systems. Also evaluated on a variety of exams originally designed for humans. When instructed to do so, GPT-4 can interact with **external interfaces**. A large focus of the GPT-4 project was building a deep learning stack that scales **predictably**: for very large training it is not feasible to do extensive model-specific tuning.

4.4 Other LLMs

4.4.1 T5

T5 is an encoder-decoder based on the transformer architeture. T5 recasts text-based language problems into text-to-text format. T5 uses a **denoising** objective function, which generalizes MLM: randdom substrings of tokens are masked; replaced by a singletoken called sentinel, which is unique over the example; the entire sentence must be reconstructed.
T5 achieves state-of-the-art results on many NLP benchmarks such as summarization, question answering, and text classification.

4.4.2 OPT

OPT suite of decoder-only pretrained transformers ranging from 125M to 175B parameters. OPt-175B is comparable to GPT-3, while requiring only (1/7)-th of the carbon footprint to develop. First time for a language technology system of this size to release the code needed to train and to use the model.

4.4.3 Gopher & PaLM

Gopher is an autoregressive language model smaller but significantly more accurate than some ultra-large language software.
PaLM is an encoder-decoder architecture using pathways, a new ML system which enables highly efficient training across multiple TPU Pods.

4.4.4 LaMDA

LaMDA is a family of conversational large language models. In june 2022, LaMDA gained widespread attention when a Google engineer made claims that the chatBot had become sentient.

4.4.5 Orca

Orca: aiming at enhancing the capability of smaller models through imitation learning, drawing on the outputs generated by larger LM. Orca learns from righ signals from GPT-4, including eexplanation traces, step-by-step thought processes, and other complex instructions.

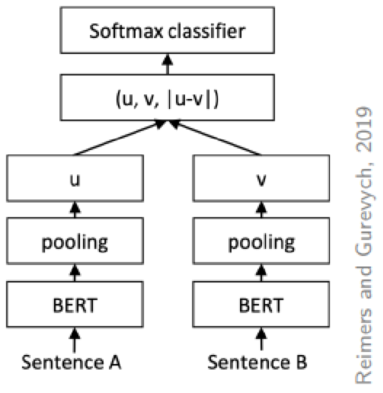
4.5 Multi-lingual LLMs

Multilingual large language models are typically based on the encoder part of the transformer architecture. A MLLM is pretrained using large amounts of unlabeled data from multipe languages, ranging from 10 to 100. Very useful in transfer learning, where low resource languages can benefit from high resource languages.
During pre-training, MLLMs use two different sources of data: large monolingual corpora in individual languages and parallel corpora between some languages. Objective functions can be based on monolingual data alone by pooling together monolingual corpora from multiple languages and applyng masked language modeling. Surprisingly, the encoder learns word representations across languages that are aligned, withut the need for any paraallel corpora.
Objective functions based on parallel corpora explicitly force representations for corresponding words across languages to be close to each other in the multilingual encoder space. Using MLM, to predict a masked word $[MASK]$ for language A the system can rely on word for language A surrounding $[MASK]$ or rely on an aligned worrd that has not been asked, that is, a word representing a translation in language B of the masked word.
Since parallel corpora are generally much smaller than monolingual dat, the parallel objecctives are often used in conjuction with monolingual models. Curse of multilinguality: similar to models that are trained on many tasks, the more languages a model is pre-trained on, the less model capacity is available to learn representations for each language.

4.6 SBERT

In some sentence-pair regression tasks, training BERT on aall sentence pairs is computationally unfeasible. Alternatively individual sentence embeddings can be computed by averaging the BERT output layer, knwon as BERT embeddings, or by using the output of the $[CLS]$ token. The above methods result in bad sentence embeddings, often worse than averaged GLoVE embeddings.
Sentence-BERT is an efficient and effective method to compute sentence embeddings. SBERT uses a siamese neural network, a special network that contains two identical subnetworks sharing their parameters. Siamese networks are most commonly used to comput similarity scores for the inputs, and have many applications in computer vision.

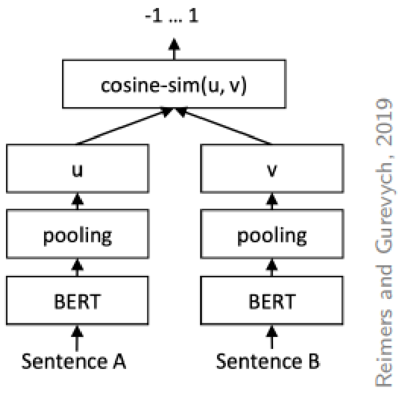
4.6.1 Training



SBERT concatenates vectors u, v, and the element-wise difference $|u - v|$ and uses trainable matrix $W \in \mathbb{R}^{3n \times k}$ to compute

$$o = \text{softmax}(W[u; v; |u - v|])$$

with n dimension of the embeddings and k number of labels. BERT is fine-tuned. parameter updating is mirrored across both ssub-networkks. Optimization uses cross-entropy loss. The two embeddings are compared



using a cosine similarity function, which will output a similarity score for the two sentences.

4.7 Miscellanea

Emergent abilities of large language models are characterized by sharpness (transiitioning seemingly instantaneously from not present to present) and unpredictability (transitioning at seemingly unforseeable model scales). This issue is still under debate and controversial.

4.7.1 Hallucination

Hallucination refers to a phenomenon where a LLM generates text that is incorrect, nonsensical, or not real. Since LLMs are not databases or search engines, they would not cite where their response is based on. Since LLMs need to be creative, they must be able to invent scenarions that are not factual. To avoid hallucinations use controlled generation: providing enough details and constraints in the prompt to the model.

4.7.2 Mixture of Experts

Mixture of Experts is a macchine learning technique where multiple expert networks are used to divide a problem space into homogeneous regions. MoE differs from ensemble techniques in that typically only one model will be run, rather than combining results from all models. this techniques has been recently used in LLMs. The result is a sparsely-activated model with a large number of parameters but a constant computational cost.

4.8 Fine-tuning

To make practical use of LLMs, we need to interface these models with downstream applications. This process is calledadaptation; it uses supervised learning for the task of interest. Adaptation is the prevalent paradigm today in natural language processing.

4.8.1 Adaptation

Two most common forms of adaptation are: feature extracion, where you freeze the pre-trained parameters of the language model and train paramters of the model for the task at hand; fine-tuning, where you make adjustments to the pre.-trained parameters.
Retraining the whole model reuslts in so-called catastrophic forgetting.

4.8.2 Adapters

When working with huge pre-trained models, fine-tuning may still be inefficient. Alternatively, one could fixx the pre-trained model, and train only small, very simple components called adapters. With adapter modules transfer become very efficient: the largest part of the pre-trained model is shared between all downstream tasks.

4.8.3 LoRA

LoRA is a popular fine-tuning strategy, altternative to adapters. it drastically redues the number of trainable parameters. LoRa iis based on the idea of **intrinsic ddimensions**: there exists a low dimension reparameterization that is as effective foor fine-tuning as the full parameter space. LoRA is known for its high efficiency, and for avoiding catastrophic forgetting. We can think of **weight update** as follows:

$$W \leftarrow W + \Delta W, W \in \mathbb{R}^{d\times d}$$

In LoRA we update the weights using a decomposition of ΔW into two low-rrank matrices:

$$\Delta W = BA, B \in \mathbb{R}^{d\times r}, A \in \mathbb{R}^{r\times d}$$

where r is some low rank.

4.9 Transfer Learning

The pre-train/fine-tune paradigm is a special case of a machine learning approach called transfer learning. The general idea of **transfer learning** is to reuse information from a previously learned source task for the learning of a target task.

4.9.1 Prompt Learning

fine-tuning LLMs on task-speccific downstream applications has become standard practice in NLP. However, the GPT-3 model with 175B parameters has brough a new way of using LLMs for downstram tasks. We can handle a wide range of tasks with only a few examples, by formulating the original task as a word prediction problem, while not updating the parameters in the underlying model.
A **prompt** is a piece of text inserted in the input examples, so that the original task can be forumalted as a masked language modeling problem. In **prompt learning** we give the LM a few example prompts to demonstrate a task. We then append a test prompt and ask the LM to make a prediction, conditionally to the example prompts.
The advantage of prompt learning is that, given a suite of apppropriate prompts, a signle 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.

4.10 Retrieval-augmented generation

Retrieval-augmented generation is a two step process combining external knowledge search with prompting. **Retrieval**: given a uery, some information retrieval neural modle fetches relevant documents/passages. **Generation**: fetched documents are wrapped into a prompt and passed to the LLM to generate relevant response.

Chapter 5

Part-of-Speech Tagging

Part-of-Speech tags are lexical categories such as noun, verb, adjective, adverb, pronoun, preposition, article, etc. We call tagset the set of all POS tags used by some model. Different languages, different grammatical theories, and diifferent applications may require different tagset. POS tags fall into two broad categories: closed class and open class. Closed class includes preposition, pronouns, article, etc. New words in this class are ralily coined. Open class consists of four major groups: nouns, verbs, adjectives and adverbs. New words apppear almost always in this class.
Words are ambiguous: depending on the context in which they appear, they may have different meaning. The task of part-of-speech tagging involves the assignment of the proper 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. The input is a sequence $x_1, ..., x_n$ of tokenized words, and the output is a sequence $y_1, ..., y_n$ of tags, with y_i the tag assigned to x_i. In POS tagging we need to output a whole sequence of tags $y_1, ..., y_n$ for the input string, not just a category. POS tagging is therefore a structured prediction task, not a classification task. The number of output structures can be exponentially large in the lenght of the input, which makes structured prediction more challenging than classification.
The accuracy of a part-of-speech tagger is the percentage of test set tags that match human gold labels.

5.1 Hidden Markov Model

Hidden Markov Models first applied in speech recognition. HMM is a generative model: it models how a class could generate some input data. You might use the model to generate examples. Let $w_{1:n} = w_1, ..., w_n$ be an input sequence of words, and let $\mathcal{Y}(w_{1:n})$ be the set of all possible tag sequences $t_{1:n} = t_1, ..., t_n$ for $w_{1:n}$. The goal of POS tagging is to choose the most most probable tag sequence $\hat{t}_{1:n} \in \mathcal{Y}(w_{1:n})$

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

The term $P(t_{1:n}|w_{1:n})$ is referred to as the posterios probability and can be difficult to model/compute. We break down the posterior probability using Bayes rule:

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

where we have used the fact that $w_{1:n}$ is given, so $P(w_{1:n})$ is a constant. 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 $P(w_{1:n}|t_{1:n})P(t_{1:n})$, whicch equals the joint probability $P(t_{1:n}, w_{1:n})$.
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:

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

The factor $P(w_i|t_i)$ is referred to as the emission probability. The 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:

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

5.2 Probability Estimation

Assume a tagged corpus, where each word has been tagged with its gold label. We implement supervised learning using the relative requency estimator. Tor the transition probability we obtain:

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

Similarly. for the emission probability we obtain:

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

5.3 HMMs as automata

We can formally define an HMM as a special type of probabilistic finite state automaton which generates sentences. The states represent hidden information, that is, the POS tags which are not observed. The transition function is defined acccording to the transition probabilities. Each state generates a word according to the emission probabilities. The generated output is an observable word sequence. The HMM definition is the following:

- finite set of output symbols V
- finite set of states Q, with initial state q_0 and final state q_f
- transition probabilities $a_{q,q'}$ for each pari q,q', $q \in Q \setminus \{q_f\}, q' \in Q \setminus \{q_0\}$
- emission probabilities $b_q(u)$ for each pair q,u, $q \in Q \setminus \{q_0, q_f\}, u \in V$

Trainssition and emission probabilities are subject to: $\sum_{q'} a_{q,q'} = 1$ for all $q \in Q \setminus \{q_f\}$, and $\sum_u b_q(u) = 1$ for all $q \in Q \setminus \{q_0, q_f\}$. 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.

5.4 Viterbi Algorithm

5.4.1 Decoding

Decoding problem for HMMs: Given a sequence of observations $w_{1:n}$ find the most probable sequence of states/tags $\hat{t}_{1:n}$

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

Also called the inference problem. In order to compute $\hat{t}_{1:n}$ we can use the following naive algorithm:

- enumerate all sequences 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 tiem, since there can be exponentially many $t_{1:n}$ in $\mathcal{Y}(w_{1:n})$!. The classical decoding algorithm for HMMs is the Viterbi algorithm, an instance of dynamic programming. 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, exploiting dynamic programming.
Let $w_{1:n} = w_1, ..., w_n$ be the input sequence. In what follows:

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

We use a two-dimensional table $vt[q,i]$ denoting the probability of the best path to get to state q after scanning $w_{1:i}$. We use a two-dimensional table $bkpt[q,i]$ for retrieving the best path. The initialization step is: for all q $vt[q,1] = a_{q_0,q}b_q(w_1)$ and $bkpt[q,1] = q_0$.
The recursive step: for all i=2,...,n and for all q:
$vt[q,i] = \max_{q'} vt[q', i-1]a_{q',q}b_q(w_i)$ and $bkpt[q,i] = \arg\max_{q'} vt[q', i-1]a_{q',q}b_q(w_i)$.
The termination step is:
$vt[q_f, n+1] = \max_{q'} vt[q', n]a_{q',q_f}$ and $bktp[q_f, n+1] = \arg\max_{q'} vt[q', n]a_{q',q_f}$.
After the execution of the algorithm we have:

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

where $\hat{t}_{1:n} = q_1, ..., q_n$ 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.

5.5 Forward Algorithm

With the goal of developing an unssupervised algorithm for the estimation of HMM, we now develop some auxiliary algorithms. Cosider the probability of the sequence $w_{1:n}$, defined as:

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

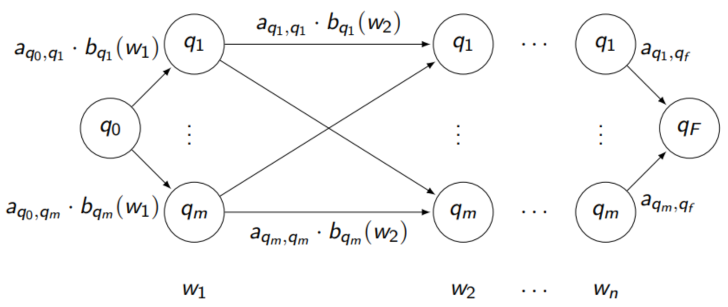
The forward algorithm computes $P(w_{1:n})$ in polynomial time, exploiting dynamic programming. The forward algorithm is very similar to Viterbi's algorithm, but using summation instead of maximisation. No use of backpointers, since we do not need to retrieve an optimal sequence. Let $w_{1:n} = w_1, ..., w_n$ 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 $w_{1:i}$. formally, this it the joint probability of $w_{1:i}$ and state q at i, and can be written as:

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

Each of the above quantities is called forward probability.
Initialization step: for all q
$\alpha[q,1]0a_{q_0,q}b_q(w_1)$
Recursive step: for all i=2,...,n and for all q
$alpha[q,i] = \sum_{q'} \alpha[q', i-1]a_{q',q}b_q(w_i)$
Termination step:
$\alpha[q_f, n+1] = \sum_{q'} \alpha[q', n]a_{q',q_f}$.
After execution of the algorithm we have

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

5.5.1 Trellis


$a_{q_0,q_1} \cdot b_{q_1}(w_1)$
q_1
$a_{q_1,q_1} \cdot b_{q_1}(w_2)$
q_1
$\cdots$
q_1
a_{q_1,q_f}
q_0
q_F
$a_{q_0,q_m} \cdot b_{q_m}(w_1)$
q_m
q_m
q_m
a_{q_m,q_f}
$a_{q_m,q_m} \cdot b_{q_m}(w_2)$
w_1
w_2
w_n

An intuitive interpretation of the forward algorithm is in terms of expansion of the HMM into a trellis, defined as follows: Given input $w_{1:n} = w_1, ..., n$:

- 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
- fir eacch pair of nodes q, q' associated with tokens w_{i-1}, w_i, respectively, introduce an arc with weight provided by the product $a_{q,q'}b_{q'}(w_i)$

- introduce special arcs for node q_f

Each path trhough the trellis corresponds to a sequencce $t_{1:n}$ of states consistent with input $w_{1:n}$. In the forward algorithm we compute one value for each node:

- the value at initial node q_0 is 1
- the value at each intermediate node is $\alpha[q,i]$, computed by summing a sequence of values, one for each incoming edge
- for each incoming edge, this value is obtained by multiplying edge value and value of previous node
- value at final node q_f is the desired probability $P(w_{1:n})$

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

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

Each of the above quantities is called **bcckward probability**.
Initialization step: for all q:
$\beta[q,n] = a_{q,q_f}$
Recursive step: for all i=n-1,...,1 and for all q:
$\beta[q,i] = \sum_{q'} \beta[q,i] a_{q,q'} b_{q'}(w_{i+1})$
Termination step:
$\beta[q_0,0] = \sum_{q'} \beta[q',q] a_{q_0,q'} b_{q'}(w_1)$.
After execution of the algorithm we have:

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

5.6.1 Duality

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

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

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.

5.7 Forward-backward Algorithm

We have already seen that, given a corpus annotated with POS tags, we can do supervised training of HMM using the relative frequency estimator. 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. Wit the forward-backward algorithm we can train HMM with tags as internal state.
Assume we have a single unannotated sentence $w_{1:n}$ to train the model- Let vector θ be an **assignment** for all 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'). c(q,u) is the expectation of the emission of symbol $u \in V$ at state q. 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 parameter do not change much anymore.

5.7.1 E-step

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 i instance

$$\alpha[q,i]a_{q,q'}b_{q'}\beta[q',i+1]$$

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

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

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

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

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

$$c(q,w_i,i) = \frac{\alpha[q,i]\beta[q,i]}{P8w_{1:n}}$$

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 occurence of a state must be followed by a transition and that an occurence of a state must be associated with an emission. Then it is not difficult to see that

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

5.7.2 M-step

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

$$\hat{a}_{q,q'} = \frac{c(q,q')}{c(q)}, \qquad \hat{b}_q(u) = \frac{c(q,u)}{c(q)}$$

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

5.8 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. 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 $x_{1:n} = x_1, ..., x_n$ be the input word sequence, and let $\mathcal{Y}(x_{1:n})$ be the set of all possible tag sequences $y_{1:n} = y_1, ..., y_n$ for $x_{1:n}$. CRFs solve the problem

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

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 the log-linear functions over a set of relevan local features, and these features are aggregated and normalised to produce a global probability. We can think of CRF as a multinomial logistic regression model, but applied to a full sequence pair $(x_{1:n}, y_{1:n})$ rather than a pair (x,y) of individual tokens. 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:

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

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

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

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 constrain above characterises linear chain CRF. This limitation makes it possible to use the Viterbi algorithm.

5.8.1 Local Features

For a predicate x, we write $\mathbb{I}\{x\}$ to denote 1 if x is true and 0 otherwise. What features to use is a decision of the system designer. To avoid feature handwriting, specific features are automatically instantiated from feature templates. Here are some templates using information from $y_{i-1}, y_i, x_{1:n}, i$:

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

It is important to use special features for unknown words. Word shape features are used to represent letter patterns. Prefix and suffix features are used to represent morphological patterns. The result of the above feature templates can be very a very large set of features. Generally, features are thrown out if they have count smaller than some cutoff in the trainign set.

5.8.2 Inference

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

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

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

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

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

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:

$$vt[q, i] = \max_{q'} vt[q', i-1] a_{q',q} b_q(w_i)$$

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

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

5.8.3 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. As in the case of multinomial logistic regression, **L1 and L" regularization** is important. To optimize the objective function, we use **stochastic gradient descent**. the local nature of linear-chain CRFs can be exploited to efficiently compute the necessary derivatives.
Let $D = \{(y^{(h)}_{1:n_h}, x^{(h)}_{1:n_h})\}$ be a trainig set, where each $x^{(h)}_{1:n_h}$ is a sentence and each $y^{(h)}_{1:n_h}$ is the associated sequence label. Let also w be the parameter vector. the **objective function** is:

$$\mathcal{L}(D, w) = \frac{\lambda}{2}||w||^2 - \sum_{h=1}^{N} \log P(y^{(h)}_{1:n_h}, x^{(h)}_{1:n_h})$$

$$= \frac{\lambda}{2}||w||^2 - \sum_{h=1}^{N} \log \frac{1}{Z(x^{(h)}_{1:n_h})} \exp\left(\sum_{k=1}^{K} w_k F_k(y^{(h)}_{1:n_h}, x^{(h)}_{1:n_h})\right)$$

$$= \frac{\lambda}{2}||w||^2 - \sum_{h=1}^{N}\left(\sum_{k=1}^{K} w_k F_k(y^{(h)}_{1:n_h}, x^{(h)}_{1:n_h})\right) + \sum_{h=1}^{N} \log Z(x^{(h)}_{1:n_h})$$

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

$$\sum_{y_{1:n} \in \mathcal{Y}(x_{1:n})} P8y_{1:n|x_{1:n}} F_k(x_{1:n}, y_{1:n})$$

In **practice**, feature expectations are computed under the hood by miodern software libraries using automatic differentiation of the objective function. Alternatively, feature expectations can be efficiently computed using the **forward-backward algorithm**.

5.9 Neural POS tagger