UA - University of Alicante

Faculty of Artificial Intelligence

Department of Computer Science and Artificial Intelligence

Natural Language Processing

Lecturers:
Prof. Miquel Esplà Gomis, Prof. Juan Antonio Pérez Ortiz

Author:
Christian Faccio

December 17, 2025

github.com/christianfaccio   christianfaccio@outlook.it

Preface

As a student of Artificial Intelligence, I've created these notes while attending the **Natural Language Processing** course.

The course provides a comprehensive introduction to the field of natural language processing, covering both theoretical concepts and practical applications. The notes encompass a variety of topics, which are divided in three main blocks:

1. Introduction to computational linguistics and natural language processing
2. Architectures for written-text processing
3. Architectures for speech

While these notes were primarily created for my personal study, they may serve as a valuable resource for fellow students and professionals interested in natural language processing.

Contents

1 Introduction to computational linguistics and natural language processing

Computational linguistics (CL) is a branch of linguistics that focuses on the theoretical understanding of language through computational models. In contrast, **natural language processing (NLP)** is an interdisciplinary field within artificial intelligence (AI) that aims to use computational models to process and generate language efficiently. NLP intersects with machine learning, statistics, and data science. While NLP is not primarily focused on linguistics, many of its approaches and tasks draw on linguistic theories to address the complexities of natural language. NLP cover a wide range of tasks, including part-of-speech tagging, named entity recognition, machine translation, speech recognition, and text summarization.

CL:

- Focuses on modeling human language using computational methods;
- Emphasizes theoretical understanding of language;
- Grounded in linguistic principles and theories;
- Examples include parsing syntactic structures and modeling phonetics.

NLP:

- Focuses on designing algorithms and systems to process natural language data;
- Driven by engineering and computational efficiency;
- Examples include machine translation, sentiment analysis, and chatbots.

While both share methods and tools such as syntax and semantics modeling or statistical and ML techniques.

1.1 Text Preprocessing

Texts come from diverse sources, languages, formats, scripts, and character encoding standards. A common preliminary step in preparing text for any NLP-related task is **preprocessing** it to make it suitable for the specific application. Typical preprocessing tasks include removing formatting, converting character encodings, and tokenizing. Additional steps often involve normalizing text, standardizing punctuation, and similar operations. A helpful introduction to these strategies and their implications for various NLP tasks can be found in the article *Comparison of text preprocessing methods* [2]. It focuses on tokenization at the word level. However, most neural-network-based approaches rely on subword-level tokenization, which involves splitting words into fragments ranging from single characters to character groups. Some popular subword-level tokenization techniques include byte-pair encoding (BPE), unigram, and SentencePiece.

A concise and intuitive explanation of these methods can be found in the Tokenizers section of the HuggingFace Transformers tutorial. Moreover, the Tiktokenizer is a tool that simulates the tokenization process of several well-known generative neural models. Select a model from the dropdown menu in the upper-right corner and input a short text to see an example of subword tokenization.

1.1.1 Format Cleaning

Raw text data usually contains different formatting elements, such as HTML tags and scripts, metadata and layout information from PDFs, or format marks from Markdown files. In most cases, data related to format adds noise to the text to be processed and should be removed.

There are different techniques for removing formatting:

- **Regular Expressions (Regex)**: Use patterns to identify and remove unwanted elements, like HTML tags (`<.*?>`);
- **Libraries and Tools** (usually in Python): `BeautifulSoup` for parsing and cleaning HTML or `PyPDF2` for extracting text from PDFs;
- **OCR Tools**: For scanned documents, Optical Character Recognition (OCR) tools like `Tesseract` can extract text while ignoring formatting.

1.1.2 Tokenization

Tokenization decomposes each text string into a sequence of words (technically **tokens**), which can represent sentences, words, characters or sub-words.

Sentence Tokenization This is the process of splitting a text into sentences, using most of the time **punctuation** marks as delimiters (e.g., periods, exclamation points, question marks). However, this can be challenging due to abbreviations, decimal points, and other punctuation uses that do not indicate sentence boundaries (remember also that not every language use punctuation, see Thai for example). Libraries like `NLTK` and `spaCy` provide pre-trained models for effective sentence tokenization.

Word Tokenization This involves splitting a text into words using **whitespaces**, **Regex** or **tailored tokenizers** that can handle specific languages or domains. Challenges include dealing with contractions (e.g., ”don’t” to ”do” and ”not”), hyphenated words, and special characters. Again, libraries like `NLTK` , `spaCy` , and `HuggingFace Tokenizers` offer robust word tokenization tools.

Character Tokenization This method breaks down text into individual characters. It is particularly useful for languages with complex morphology or when dealing with noisy text data, such as social media posts. Character tokenization can also be beneficial in certain NLP tasks like language modeling and text generation. However, it is almost never used alone in modern NLP applications.

Sub-word Tokenization This technique has become popular in neural-based NLP models, since it addresses issues with rare words and OOV words. Moreover, it is efficient for **morphologically rich languages** and mantains a balance between word and character tokenization. For this task, different algorithms have been developed, such as **Byte-Pair Encoding (BPE)**, **Unigram**,**WordPiece** and **SentencePiece**. These methods break down words into smaller units based on their frequency in the training corpus, allowing models to handle a wider variety of words and forms. For more details, refer to the HuggingFace Tokenizers documentation.

1.1.3 Text Normalization

This process aims at converting text into a standard form, reducing variablility in the text while preserving meaning. It also prepares text for consistent and effective processing in NLP tasks. Examples include converting text to lowercase, removing punctuation, expanding contractions (e.g., ”don’t” to ”do not”), correcting spelling errors, and standardizing formats for dates, numbers, and abbreviations.

However, modern text is usually encoded with **Unicode** standards, supporting a wide variety of scipts. This leads to data sparsity at character level.

Finally, text normalization varies depending on the specific NLP task and language. For instance, in sentiment analysis, preserving certain punctuation (like exclamation marks) may be important, while in machine translation, maintaining the original casing and punctuation is crucial for accuracy. More examples include:

- **Case-sensitive tasks**: Named Entity Recognition (NER), where capitalization can indicate proper nouns;
- **Removing Punctuation**: Useful in BoW models, but not always suitable for tasks like sentiment analysis;
- **Removing redundant text**: Removing duplicates sentences or paragraphs in a corpus is useful when training language models.

1.1.4 Identifying Stopwords

Stopwords are common words that carry little meaningful information and are often removed during text preprocessing to reduce noise and improve model performance. Examples include ”the”, ”is”, ”in”, ”and”, etc. However, the decision to remove stopwords depends on the specific NLP task. For instance, in sentiment analysis, words like ”not” can significantly alter the meaning of a sentence and should be retained. They can be detected in different ways, like using a **predefined stopwords list** (e.g., from `NLTK` or `spaCy`), by analyzing **word frequency** in the corpus to identify common words that may not contribute significantly to the task at hand or with **POS tagging** to identify function words that are typically considered stopwords.

Observation: ***Zipfian distribution of vocaboulary***

When the words in a corpus are ranked decreasingly they follow a **zipfian distribution** in which

$$freq(r) \propto \frac{1}{r}$$

In other words, a few words in most languages have a very high frequency and most of the words in a language have a very low frequency. This implies that removing stopwords can significantly reduce the vocabulary size without losing much information.

The implications of removing stopwords should be carefully considered based on the specific NLP task and the characteristics of the dataset being used:

- **Focus on meaningful terms**: Removing stopwords can help models focus on more meaningful terms that contribute to the overall context and meaning of the text;
- **Risk of losing context**: In some cases, stopwords can provide important context or nuance to the text. For example, in sentiment analysis, words like ”not” can significantly alter the meaning of a sentence;

- **Task-specific considerations**: Some tasks like sentiment analysis may benefit from retaining stopwords.

1.2 Morphological Parsing

Computational morphology refers to the design of software that analyzes or generates words not as atomic, indivisible units, but as the intricately structure objects linguistics have long recognized them to be. **Morphology** is the study of the stucture of words and the rules for word formation in a language. It focuses on the internal structure of words, including *morphemes*, which are the smallest meaningful units of language. In morphological parsing, we break down words into:

- **Lemmas**: Base forms of words (e.g., ”running” to ”run”);
- **Morphemes**: Smallest units of meaning (e.g., ”unhappiness” to ”un-”, ”happy”, ”-ness”).

This is essential for understanding word formation, meaning and grammatical roles.

In NLP, morphological parsing is used to simplify texts, extracting information relevant to understand meaning, generating morphologically-correct text and supporting language learners. For low-resource languages, valuable morphological resources are typically small or non-existent [5]. For richly inflected languages, morphological parsing is crucial to handle the complexity of word forms and their grammatical relationships. The canonical form of a word is called the *lemma*, and the set of all surface forms of it is called the *paradigm*. To help having annotated morphological data with a universal tagset, the **UniMorph** project has been created [4]. It provides a large-scale, multilingual database of morphological paradigms and annotations for over 100 languages, where each inflected form is associated with a lemma, that typically carries its underlying lexical meaning and a bundle of morphological features (e.g., tense, number, case). The database is organized in triplets of the form (lemma, inflected form, morphological features).

Lemma	Inflected Form	Morphological Features
run	running	V;PRS;PROG
run	ran	V;PST
child	children	N;PL
happy	happier	ADJ;COMP

Table 1.1: Example entries from the UniMorph database.

Whereas UniMorph contains type-level annotations, the **Universal Dependencies (UD)** project provides token-level annotations for sentences in many languages, including morphological features, syntactic dependencies, and part-of-speech tags. UD treebanks are widely used for training and evaluating NLP models on various tasks, including morphological analysis. The structure is useful for morphological tagging at the sentence level, where each word in a sentence is annotated with its morphological features.

Word	Lemma	POS Tag	Morphological Features
running	run	VERB	Tense=Pres;Aspect=Prog
ran	run	VERB	Tense=Past
children	child	NOUN	Number=Plur
happier	happy	ADJ	Degree=Comp

Table 1.2: Example entries from a Universal Dependencies treebank.

In the simplest setting, we simply wish to obtain a detailed morphological summary of a given word. For example, take the word *puppies* in English. A morphological parser should be able to identify that this word is the plural form of the noun *puppy*. Such a task is called **morphological tagging** and is very useful and valuable for ”downstream” tasks, such as **parsing** [1], recovering the syntactic structure of a sentence. This morhological summary might also include the word **segmentation**, which might break the word down into its morphemes: *puppy* + *-ies*. This suggests a more sophisticated alternative to stemming: **lemmatization**, or replacing inflected words with their lemmas. The inverse problem, **morphological generation**, is a key part on many generative systems.

Nowadays, **data-driven** methods are the most successful approaches to morphological parsing. These methods typically involve training machine learning models on annotated corpora, such as those provided by the UniMorph and Universal Dependencies projects. In this notes I will focus only on this approach, while the **knowledge-based** approaches can be found in [1].

1.2.1 Morphological Tagging

Morphological tagging is a sequence-labeling task similar to part-of-speech tagging. It considers words in context, assigning each word a set of morphological features based on its role in the sentence. For instance, in the sentence ”The cats are playing”, the word ”cats” would be tagged as a noun with plural number. Taggers are important building blocks for many other natural language processing tasks. PoS and morphological tags are used for different ”downstream” processing tasks, such as named entity recognition, syntactic parsing, and machine translation. Accurate morphological tagging can improve the performance of these tasks by providing additional linguistic information about the words in a sentence.

Tagging is a structured prediction problem, that requires us to simultaneously make a series of interdependent decisions to obtain the best overall prediction. One method to address this problem is the **Hidden Markov Model** (HMM), which tells a simple ”story” about how data are produced. It imagines that each tag is generated by the previous tag, and each word is then generated by its tag.

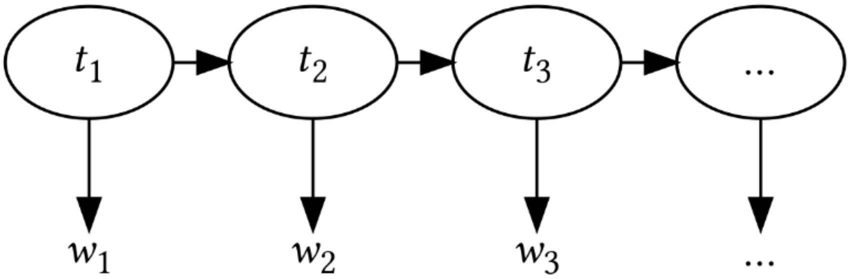


Figure 1.1: Graphical representation of a Hidden Markov Model for morphological tagging. Each circle represents a hidden state (morphological tag), and each square represents an observed word. Arrows indicate dependencies between states and observations.

Without going into much details, which you can find in [1], here the Viterbi algorithm can be used to efficiently find the most probable sequence of tags for a given sequence of words. More advanced models, such as Conditional Random Fields (CRFs) and neural network-based approaches (e.g., LSTMs, Transformers), have also been applied to morphological tagging with great success.

1.2.2 Morphological Segmentation

With segmentation, the goal is to split words into their smallest meaning-bearing units: **morphemes**. There are two types of segmentation:

- **Surface Segmentation**: Splits a word into morphemes in a way such that the concatenation of all parts exactly results in the original word. For example, the word ”unhappiness” can be surface-segmented into ”un-”, ”happy”, and ”-ness”. (Note that this is not necessarily meaningful for all languages);
- **Canonical Segmentation**: It is more complex as it aims to split a word into morphemes and to undo the orthographic changes which have occurred during word formation. As a result, each word is segmented into its *canonical* morphemes. For example, the word ”running” would be canonically segmented into ”run” and ”-ing”, restoring the base form of the verb.

1.2.3 Lemmatization, Inflection, Reinflection

Inflection and reinflection are concerned with generating inflected forms of a lemma. The former generates a word from a given lemma and a set of morphological features, while the latter generates a new inflected form from an existing inflected form and a target set of morphological features. For example, given the lemma ”run” and the features ”3rd person singular present”, an inflection system would generate ”runs”. Given the inflected form ”running” and the target features ”past tense”, a reinflection system would generate ”ran”. **Lemmatization** is the process of reducing an inflected word to its base or dictionary form, known as the lemma. For example, the words ”running”, ”ran”, and ”runs” would all be lemmatized to ”run”. It is essentially a special case of the reinflextion and a sort of tagging.Lemmatization is important for various NLP tasks, such as information retrieval and text analysis, as it helps to group together different forms of a word.

Most commonly, these operation refer to type-level tasks. The input consists of an input form together with the target morphosyntactic description (MSD).

$$\text{mutated V;3;SG;PRS} \rightarrow \text{mutates}$$

The token-level version of the task is often referred to as lemmatization or inflextion *in context*, meaning that the system has access to the sentential context in which the word appears. This is particularly useful for languages with high levels of homography, where the same surface form can correspond to different lemmas or morphological analyses depending on the context.

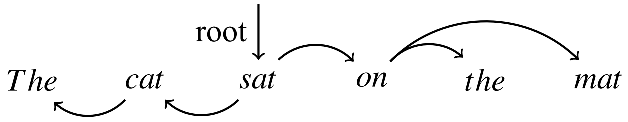
$$\text{mutate - The virus [MASK]} \rightarrow \text{mutates}$$

A drawback of this formulation is that typically many different inflected forms are possible with the same context. To overcome this problem, some approaches model the task as a sequence-to-sequence problem, where the input is the entire sentence with the target word marked, and the output is the inflected form of the target word. This allows the model to learn to generate the correct inflected form based on the context provided by the surrounding words.

1.3 Syntactic Parsing

Syntactic parsing is aimed at determining the structure of a sentence and provides representations that help understand relationships between words. Two main approaches exist: **constituency parsing** and **dependency parsing**.

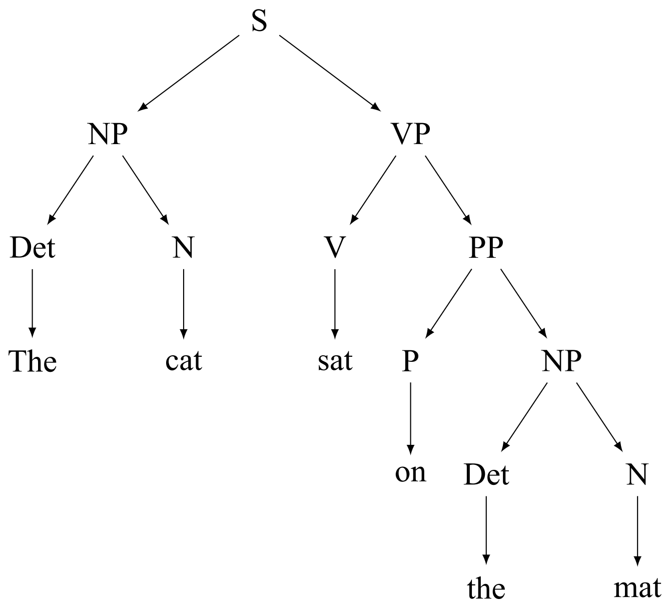
Depencency Structure Represents syntax as directed relationships between words, capturing dependencies directly. Each word is linked to its dependents, forming a tree structure where the root is typically the main verb. This approach is particularly useful for languages with flexible word order, as it focuses on the relationships between words rather than their positions in the sentence. Take for example sentence ”The cat sat on the mat”. The dependency structure would identify ”sat” as the root verb, with ”cat” as its subject and ”on the mat” as a prepositional phrase modifying the verb. Below is an illustration of the dependency parse tree for this sentence.



Dependency parsing means predicting linguistic structure from input sentences by establishing relationships between ”head” words and words which modify those heads. Dependency parsers can be built using various approaches, including rule-based methods, statistical models, and neural network-based techniques. Modern dependency parsers often leverage deep learning architectures, such as BiLSTMs or Transformers, to capture complex syntactic patterns in text. There are two main approaches:

- *Transition-based models*: These models build the dependency tree incrementally by making a series of decisions (transitions) based on the current state of the parse. They are typically faster and more efficient, making them suitable for real-time applications. They can easily condition on infinite context, but use greedy search algorithms that can cause short-term mistakes (see Shift-reduce parsing for more details);
- *Graph-based models*: These models calculate probabilities for each edge/constituent and perform dynamic programming to find the highest-scoring tree. They tend to be more accurate but computationally intensive, making them less suitable for real-time applications. They consider global context and optimize the entire tree structure, but can be slower and require more computational resources (see Jurafsky & Martin, Speech and Language Processing for more details).

Phrase Structure Represents syntax as nested phrases and is also known as constituency parsing. It breaks down sentences into hierarchical structures of phrases, such as noun phrases (NP) and verb phrases (VP). Each phrase can contain other phrases or words, forming a tree structure that reflects the grammatical organization of the sentence. For the same sentence ”The cat sat on the mat”, the phrase structure would identify ”The cat” as a noun phrase (NP) and ”sat on the mat” as a verb phrase (VP). Below is an illustration of the phrase structure parse tree for this sentence.



Dependency parsing is often preferred for its simplicity and direct representation of word relationships, while phrase structure parsing provides a more detailed hierarchical view of sentence structure. The choice between the two approaches depends on the specific NLP task and the linguistic characteristics of the language being analyzed, even if usually the former is easier to apply to languages with different word orders. They help NLP tasks in different ways:

- Splitting text into meaningful fragments;
- Disambiguating word meanings based on context;
- Knowledge-enhanced models (summarization, translation, etc.);
- Helping identifying named entities.

The project **Universal Dependencies (UD)** provides a standardized framework for dependency parsing across multiple languages, facilitating cross-linguistic research and applications. UD treebanks are widely used for training and evaluating dependency parsers, making them a valuable resource in the NLP community. Moreover, tools like the Stanza library or SciPy provide pre-trained models for efficient dependency parsing.

1.4 Semantic Representation of Text

Definition: ***Semantic compositionality principle***

The meaning of a complex expression (a **sentence**) is determined by the meanings of its constituent parts (**words**) and the way they are combined (**syntax**).

A **lexeme** is a unit of meaning in language, independent of inflectional forms. An example is the lexeme ”run”, which includes forms like ”runs”, ”running”, and ”ran”. Lexemes are crucial for understanding semantics, as they represent the core meaning of words. The **word sense**, instead, is the specific meaning of a word in a given context, which can vary based on usage. For example, the word ”bank” can refer to a financial institution or the side of a river, depending on the context. Understanding word senses is essential for tasks like word sense disambiguation, where the goal is to determine the correct meaning of a word based on its context.

Words are related by their meaning:

- **Synonymy**: Words with similar meanings (e.g., ”big” and ”large”);
- **Antonymy**: Words with opposite meanings (e.g., ”hot” and ”cold”);
- **Similarity**: Words that are related in meaning but not identical (e.g., ”car” and ”vehicle”);
- **Relatedness**: Words belong to the same *semantic field* (e.g., ”doctor” and ”hospital”).
- **Connotation**: The emotional or cultural associations of a word (e.g., ”home” connotes warmth and safety).

Vector Semantics is a method of representing word meanings using vectors in a high-dimensional space. Words are represented as points in this space, where the distance between them reflects the semantic similarity of words and words appearing in similar contexts are closer in the vector space.

There are three main approaches to vector semantics:

- **Bag of Words (BoW)**: Represents documents as a vector of word counts, ignoring grammar, word order and context. Given a corpus (e.g. ”The cat sat on the mat.”), we represent each document as a vector of word counts (the length of the vector is equal to the size of the vocaboulary, determined through tokenization). In this case, the BoW representation would be [1, 1, 1, 1, 1, 1] for the words [”The”, ”cat”, ”sat”, ”on”, ”the”, ”mat”] respectively. This creates a sparse and high-dimensional vector space, which can be computationally expensive to work with;
- **TF-IDF**: Weights words by their importance in the document and corpus, reducing the impact of frequent but uninformative words (e.g. ”the”). The first step is to define the **term frequency (TF)**:

  $$\text{TF} = \frac{\text{Number of occurrences of the term in the document}}{\text{Total terms in the document}}$$

  Then, we define the **inverse document frequency (IDF)**:

  $$\text{IDF} = \log \frac{\text{Total number of documents}}{\text{Number of documents containing the term}}$$

  Finally, the **TF-IDF** score is computed as:

  $$\text{TF-IDF} = \text{TF} \times \text{IDF}$$

  It is important to take into consideration that TF-IDF may give low scores to semantically important words, and produces, as BoW, sparse and high-dimensional vectors;
- **Embeddings**: Dense vector representations of text in a continuous vector space, they capture the semantic and syntactic relationships between words. Unlike BoW or TF-IDF, embeddings are **dense** (low-dimensional) and are learned automatically from data rather than being based on simple counting or weighting. Different models and strategies exist to generate embeddings, such as **Word2Vec**, **GloVe**, and **FastText**. More recently, contextual embeddings from models like **BERT** and **GPT** have become popular, as they capture the meaning of words in context, allowing for more nuanced representations.

  Neural Networks are used to learn embeddings by training on large corpora of text. The networks learn to predict words based on their context (or vice versa), adjusting the vector representations of words to minimize prediction errors. This process results in embeddings that capture semantic relationships, such as synonyms being close together in the vector space. For a recall on neural networks, refer to [3].

  To compare two vector representations, a measure of **distance** or **similarity** has to be computed, usually with the Euclidean Distance (straight-line distance between vectors) or with the Cosine Similarity (cosine angle between bectors in the vector space).

  $$\text{cosine similarity} = \frac{\vec{u} \cdot \vec{v}}{||\vec{u}||\,||\vec{v}||}$$

2 Architectures for written-text processing

2.1 Logistic Regression

Before diving deeper into Neural Networks, it is useful to understand their littele brother: **Logistic Regression**. It can be used to classify an observation onto one of two classes (binary classification), or into one of many classes (multinomial classification). This section builds from the book Speech and Language Processing [3].

Logistic Regression is a discriminative model, meaning that it models the conditional probability $P(y|x)$ directly, where y is the class label and x is the input feature vector. The model assumes a linear relationship between the input features and the log-odds of the class probabilities. A machine learning system for classification is based on four components:

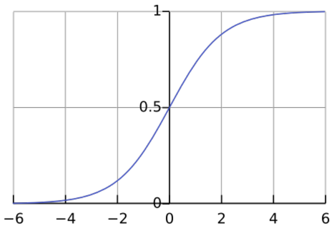
1. A **feature representation** of the input data, in the form of a vector of real-valued features $x = [x_1, x_2, \ldots, x_n]$;
2. A classification function that computes $\hat{y}$, the estimated class, via $p(y|x)$ (see the **sigmoig** and **softmax** functions);
3. An objective function for learning the model parameters from training data, minimizing error on the training set (in this case **cross-entropy loss function**);
4. An algorithm for optimizing the objective function, such as **gradient descent**.

Starting from a single input observation x represented as a vector of n features $[x_1, x_2, \ldots, x_n]$, the classifier output can be 1 or 0 in the **binary classification**. Logistic regression solves this task by learning, from a training set, a vector of **weights** W and a **bias term** b (which will be broadcasted to a vector in this case). The weight w_i represents how important feature x_i is to the classification decision.

$$z = \left(\sum_{i=1}^{n} w_i x_i \right) + b = \mathbf{w}^\top \mathbf{x} + b$$

To create a probability, then, we'll pass z through the **sigmoid** function $\sigma(z)$, also called the **logistic function**. It has a domain of $(-\infty, +\infty)$ and a range of $(0, 1)$, making it suitable for modeling probabilities.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



We can then derive the probability of class 1 as:

$$P(y=1|x) = \sigma(z) = \frac{1}{1+e^{-(\mathbf{w}^\top \mathbf{x}+b)}}$$

Observation:

The sigmoid function has the property

$$\sigma(-z) = 1-\sigma(z)$$

which implies that

$$P(y=0|x) = 1-P(y=1|x) = \sigma(-z)$$

The **decision boundary** is the value of z for which we make a decision about which class to assign to a test instance x. Usually we set a threshold of 0.5 for the probability of class 1. This corresponds to $z=0$, since $\sigma(0)=0.5$. Therefore, if $z \geq 0$, we classify the instance as class 1; otherwise, we classify it as class 0.

As for now, we only consider a single example, but in practice we have to handle many of them. The most efficient approach is to use matrix multiplication to compute the outputs for all m examples at once. The input data is still represented as a vector of features, but now the output must be a **one-hot vector** of K values representing the classes. This way, only one of the K values is 1, indicating the correct class, while all other values are 0.

$$\underbrace{\mathbf{Z}}_{(K,1)} = \underbrace{\mathbf{W}}_{(K,n)} \underbrace{\mathbf{x}}_{(n,1)} + \underbrace{\mathbf{b}}_{(K,1)}$$

Moreover, the **softmax** function is used in this case. It is a generalization of the sigmoid function for multi-class classification problems, where there are more than two classes. The softmax function takes a vector of real-valued scores (logits) and converts them into a probability distribution over multiple classes. Given a vector of logits $\mathbf{z} = [z_1, z_2, \ldots, z_k]$, the softmax function computes the probability of each class i as follows:

$$P(y=i|\mathbf{z}) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

Applying the softmax function to the logistic regression model, we have to separate weight vectors $\mathbf{w}_i$ and bias b_i for each of the K classes.

$$p(y_i|\mathbf{x}) = \frac{e^{\mathbf{w}_i^\top \mathbf{x}+b_i}}{\sum_{j=1}^{K} e^{\mathbf{w}_j^\top \mathbf{x}+b_j}}$$

where $\mathbf{w}$ has shape $[n,K]$ and $\mathbf{b}$ has shape $[K,1]$.

$$\underbrace{\hat{y}}_{(K,1)} = softmax(\underbrace{\mathbf{W}}_{(K,n)} \underbrace{\mathbf{x}}_{(n,1)} + \underbrace{\mathbf{b}}_{(K,1)})$$

2.1.1 Cross-entropy Loss Function

We now need a loss function that expresses, for an observation x, how close the classifier output $\hat{y}$ is to the correct output y. We do this via a loss function that prefers the correct class labels of the training examples to be *more likely*. We therefore choose the parameters $\mathbf{W}$ and $\mathbf{b}$ that maximize the likelihood of the correct class labels in the training data. This is equivalent to minimizing the **cross-entropy loss function**, which is defined as follows for a single training example:

$$L(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$$

Observation: ***Cross-entropy Loss = Negative Log-Likelihood***

The information an event x carries is defined as $I(x) = -\log P(x)$. The cross-entropy between two probability distributions p and q over the same set of events measures the average number of bits needed to identify an event drawn from the set, if a coding scheme used for the set is optimized for an estimated probability distribution q, rather than the true distribution p. It is defined as:

$$H(p, q) = -\sum_{x} p(x) \log q(x)$$

We can notice that the cross-entropy loss function is equivalent to the negative log-likelihood of the correct class label (look at the binary classification to understand better).

The cross-entropy loss function can be written also as:

$$L(\hat{y}, y) = -\sum_{i=1}^{K} y_i \log \hat{y}_i$$

We now need an algorithm to minimize the loss function over the training set. A common choice is **gradient descent**, which iteratively updates the model parameters in the direction of the negative gradient of the loss function with respect to the parameters. The update rule for the weights and bias is as follows:

$$\mathbf{W} := \mathbf{W} - \eta \nabla_{\mathbf{W}} L(\hat{y}, y)$$

where η is the learning rate, a hyperparameter that controls the step size of each update.

Warning:

To use the gradient descent algorithm, we need to be able to compute the gradient of the loss function with respect to the model parameters. This is done using the **backpropagation** algorithm, which efficiently computes the gradients by applying the chain rule of calculus, but also introduces the need to have proper differentiable functions in the model.

The gradient descent algorithm can be applied in two main ways: **batch gradient descent** and **stochastic gradient descent (SGD)**. In batch gradient descent, the gradients are computed using the entire training set, while in SGD, the gradients are computed using a single training example at a time. A compromise between these two approaches is **mini-batch gradient descent**, where the gradients are computed using a small subset of the training set (mini-batch) at each iteration.

It's now time to generalize the loss function from 2 to K classes. We represent both y and $\hat{y}$ as vectors, and the loss function is the sum of the logs of th K output classes, each weighted by their probability y_i:

$$
\begin{aligned}
L(\hat{y},y) &= -\sum_{i=1}^{K} y_i \log \hat{y}_i \\
&= -\log \hat{\mathbf{y}}_{\mathbf{c}} \quad \text{(where c is the correct class)} \\
&= -\log \frac{\exp \mathbf{w}_c\mathbf{x} + b_c}{\sum_{j=1}^{K} \exp \mathbf{w}_j\mathbf{x} + b_j}
\end{aligned}
$$

Moreover, we can derive the partial derivative of the loss with respect to $\mathbf{w}_{k,i}$ as:

$$
\frac{\partial L}{\partial \mathbf{w}_{k,i}} = (\hat{y}_i - y_i)x_k
$$

Advanced Concept: ***Deriving the Gradient Equation***

The **chain rule** is a fundamental rule in calculus that allows us to compute the derivative of a composite function. It states that if we have two functions $f(g(x))$, then the derivative of the composite function with respect to x is given by:

$$
\frac{d}{dx} f(g(x)) = f'(g(x)) \cdot g'(x)
$$

Using the chain rule, we can derive the partial derivative of the loss function with respect to $\mathbf{w}_{k,i}$ as follows:

$$
\begin{aligned}
\frac{\partial L}{\partial \mathbf{w}_{k,i}} &= -\frac{\partial}{\partial \mathbf{w}_{k,i}} \log \hat{y}_c \\
&= -\frac{1}{\hat{y}_c} \cdot \frac{\partial \hat{y}_c}{\partial \mathbf{w}_{k,i}} \\
&= -\frac{1}{\hat{y}_c} \cdot \frac{\partial}{\partial \mathbf{w}_{k,i}} \left(\frac{e^{\mathbf{w}_c\mathbf{x}+b_c}}{\sum_{j=1}^{K} e^{\mathbf{w}_j\mathbf{x}+b_j}} \right) \\
&= -\frac{1}{\hat{y}_c} \cdot \left(\frac{e^{\mathbf{w}_c\mathbf{x}+b_c} x_k(\delta_{i,c} - \hat{y}_i)}{\sum_{j=1}^{K} e^{\mathbf{w}_j\mathbf{x}+b_j}} \right) \\
&= -(1)x_k(\delta_{i,c} - \hat{y}_i) \\
&= (\hat{y}_i - y_i)x_k
\end{aligned}
$$

where $\delta_{i,c}$ is the Kronecker delta, which is 1 if $i = c$ and 0 otherwise.

2.2 Embeddings

Vector semantics is the standard way to represent word meaning in NLP. It derives from two ideas: using a point in a 3D space to represent the connotation of a word and defining the meaning of a word by its **distribution** in language use, considering the neighborhood of words that tend to occur near it.

Definition: ***Embeddings***

An **embedding** is simply a multidimensional vector to represent words or other discrete items. The idea is to map each word to a point in a continuous vector space, where semantically similar words are mapped to nearby points.

They are **dense** vectors, meaning that the number of dimensions is much lower than the vocabulary size $|V|$, and these dimensions don't have a clear interpretation. Having dense vectors instead of sparse ones helps in different tasks, since we have to learn much less weights. Here we will dive deeper into one model: the **skip-gram with negative sampling (SGNS)**, usually referred to also as Word2Vec.

Observation: ***Cosine Similarity***

To measure similarity between two target words v and w, we need a metric that takes two vectors (of the same dimensionality) and gives a measure of similarity. We consider here the **cosine similarity**, which measures the angle between the two vectrs. It is based on the dot product, in fact it is a **normalized dot product**, corrected since the normal one favors long vectors ($|\mathbf{v}|$):

$$cosine(\mathbf{v}, \mathbf{w}) = \frac{\mathbf{v} \cdot \mathbf{w}}{|\mathbf{v}||\mathbf{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}}$$

The cosine value ranges from 1 for vectors pointing in the same direction, through 0 for orthogolan vectors, to -1 for vectors pointing in opposite directions.

Word2Vec embeddings are **static embeddings**, meaning that the method learns one fixed embedding for each word in the vocaboulary, contrary to the **contextual embeddings** that will be later explained. Its intuition is to train a classifier on a binary prediction task ("Is word w likely to show up near word c?") and then use the learned weights (initialized randomly) as embeddings. It seems like a cycle, but remember that the embeddings are the weights and the whole procedure of learning them is the algorithm. An important consideration is that we can apply **self-supervision** in this case, since we can check in an online way if our prediction is corrext just by using running text.

Given a text, we want to train a classifier such that, given a tuple (w, c) of a target word w paired with a candidate context word c, it will return the probability that c is a real context word.

$$P(+|w, c)$$

To compute this probability, we use embedding similarity: a word is likely to occur near the target if its embeddings (weights) are simiar to the ones of the target. We then consider the **dot product**:

$$Similarity(w, c) \approx \mathbf{c} \cdot \mathbf{w}$$

To have a probability, we then use the **sigmoid** function:

$$P(+|w,c) = \sigma(\mathbf{c}\cdot\mathbf{w}) = \frac{1}{1+e^{-\mathbf{c}\cdot\mathbf{w}}}$$

Since we also need the total probability of all the cases to be 1, and simplifying using the independence assumption, we have that:

$$P(+|w,c_{1:L}) = \prod_{i=1}^{L} P(+|w,c_i) = \prod_{i=1}^{L} \sigma(\mathbf{c}_i\cdot\mathbf{w})$$

Skip-gram stores **two embeddings** for each word: the **target embedding** (used when the word is the target word w) and the **context embedding** (used when the word is a context word c). This way, we can have different representations for words depending on their role in the prediction task.

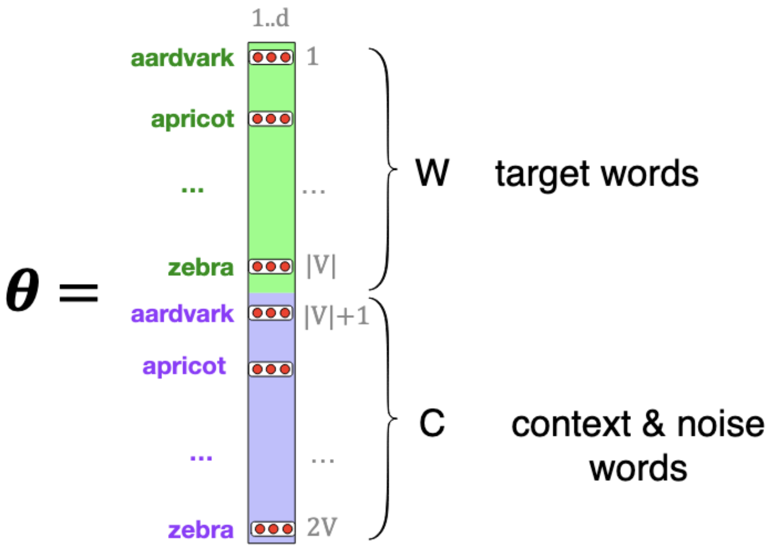


Figure 2.1: Skip-gram architecture.

The learning algorithm takes as input a corpus of text and a chosen vocabulary size N. It randomly initializes the embeddings and then iteratively shifts them for each word to be more like the embeddings of word that occur nearby in texts, and less like the embeddings of words that do not. Since we need **negative samples**, the algorithm first determines the correct pairs and then randomly samples k words from the vocabulary to create negative pairs. The loss function to minimize is then:

$$\begin{aligned}
L_{CE} &= -\log\left[P(+|w,c_{pos})\prod_{i=1}^{k}P(-|w,c_{neg_i})\right] \\
&= -\left[\log P(+|w,c_{pos}) + \sum_{i=1}^{k}\log P(-|w,c_{neg_i})\right] \\
&= -\left[\log P(+|w,c_{pos}) + \sum_{i=1}^{k}\log(1-P(+|w,c_{neg_i}))\right] \\
&= -\left[\log\sigma(\mathbf{c}_{pos}\cdot\mathbf{w}) + \sum_{i=1}^{k}\log\sigma(-\mathbf{c}_{neg_i}\cdot\mathbf{w})\right]
\end{aligned}$$

This loss function is constructed such that it:

- Maximizes the similarity between the target word, context word pairs (w, c_{pos}) drawn from the positive examples;
- Minimizes the similarity between the target word, context word pairs (w, c_{neg}) drawn from the negative examples.

We minimize this loss function using stochastic gradient descent (SGD).

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \left[[\sigma(\mathbf{c}_{pos} \cdot \mathbf{w}^t) - 1]\, \mathbf{c}_{pos} + \sum_{i=1}^{k} [\sigma(\mathbf{c}_{neg_i} \cdot \mathbf{w}^t)]\, \mathbf{c}_{neg_i} \right]$$

For proofs and more details, refer to [3].

2.3 Feedforward Neural Networks

For this session, I will be short since you have probably already seen this concepts in other courses. Consider this just a quick recap of the main ideas and if you still have doubts or need more in-depth proofs, check the book [3].

A **feedforward neural network (FNN)** is a multilayer network in which the units are connected with no cycles. They are sometimes called **multi-layer perceptrons (MLP)**. Three nodes are present in a FNN:

- input units (features) → $\mathbf{x}$;
- hidden units (intermediate computations) → $\mathbf{h}$;
- output units (predictions) → $\hat{\mathbf{y}}$.

Each layer here is **fully connected**, meaning that each unit in each layer takes as input the outputs from all the units in the previous layer, and there is a link between every pair if units from two adjacent layers. Thus, each hidden unit sums over all the input units. We represent the parameters for the entire hidden layer by combining the weight vector and bias for each unit i into a single weight matrix $\mathbf{W}$ and a single bias vector b.

The computation only has three steps: multiplying the weight matrix by the input vector $\mathbf{x}$, adding the bias vector $\mathbf{b}$ and applying the activation function g.

$$\underbrace{\mathbf{h}}_{d_h \times 1} = \sigma \left(\underbrace{\mathbf{W}}_{d_h \times n_0} \underbrace{\mathbf{x}}_{n_0 \times 1} + \underbrace{\mathbf{b}}_{d_h \times 1} \right)$$

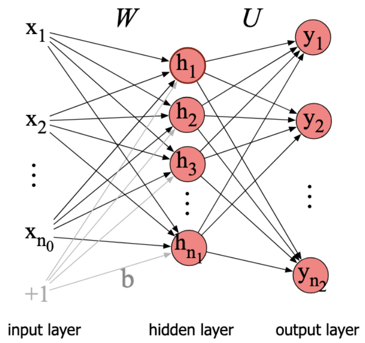


Figure 2.2: Feedforward Neural Network with one hidden layer.

Like the hidden layer, the output layer has a weight matrix (**U**), but some models don't include a bias vector **b**. The weight matrix is then multiplied by its input vector **h** to produce the intermediate output **z**:

$$\underbrace{\mathbf{z}}_{|V|\times 1} = \underbrace{\mathbf{U}}_{|V|\times d_h} \underbrace{\mathbf{h}}_{d_h\times 1}$$

However, **z** cannot be the output of a classifier, since its values are unbounded. We therefore need to apply the **softmax** function to obtain a probability distribution over the output classes:

$$\underbrace{\hat{\mathbf{y}}}_{|V|\times 1} = softmax(\underbrace{\mathbf{z}}_{|V|\times 1}) = softmax(\underbrace{\mathbf{U}}_{|V|\times d_h} \underbrace{\mathbf{h}}_{d_h\times 1})$$

Observation: ***Replacing the bias unit***

Instead of having a separate bias vector **b**, we can add an extra input unit x_0 that is always equal to 1. This way, the bias term can be absorbed into the weight matrix **W**, simplifying the notation. The same can be done for the output layer.

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x}+\mathbf{b}) \quad \longrightarrow \quad \mathbf{h} = \sigma(\mathbf{W}'\mathbf{x}')$$

and so, instead of using a vector **x** of size n_0, we use a vector $\mathbf{x}'$ of size $n_0 + 1$, where the first element is always 1.

$$\mathbf{h}_j = \sigma\left(\sum_{i=0}^{n_0} \mathbf{W}_{ji}\mathbf{x}'_i\right)$$

Let's now consider **language modeling**, meaning the task of predicting upcoming words from prior word context. We can apply a FFN to language modeling by taking as input the representation of some number of previous words (the context), and outputting a probability distribution over possible next words.

$$P(w_t|w_1,\dots,w_{t-1}) \approx P(w_t|w_{t-N+1,\dots,w_{t-1}})$$

The representations of the previous words can be their embeddings, concatenated together to form the input vector **x**. The output layer will then produce a probability distribution over the vocabulary, representing the likelihood of each word being the next word in the sequence. The model can be trained using a cross-entropy loss function, comparing the predicted probabilities with the actual next word in the training data. But let's start from the beginning. For the task of **forward inference**, meaning executing a forward pass on the network and computing the output probabilities, we represent each word by a one-hot vector, and multiply it by the embedding matrix **E** to obtain its embedding vector. We then concatenate (**pooling**) the vectors to form the input vector for the FFN.

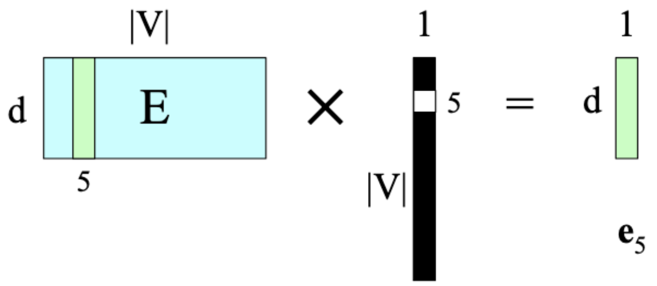


Figure 2.3: Creating the word embeddings.

The steps are the following:

1. Pool the embeddings of the context words to form the input vector **x**;
2. Multiply the vector by the weight matrix **W**, add the bias vector **b** and apply the activation function σ to obtain the hidden layer **h**;
3. Multiply the hidden layer by the output weight matrix **U** to obtain the intermediate output **z**;
4. Apply the softmax function to **z** to obtain the output probabilities $\hat{\mathbf{y}}$.

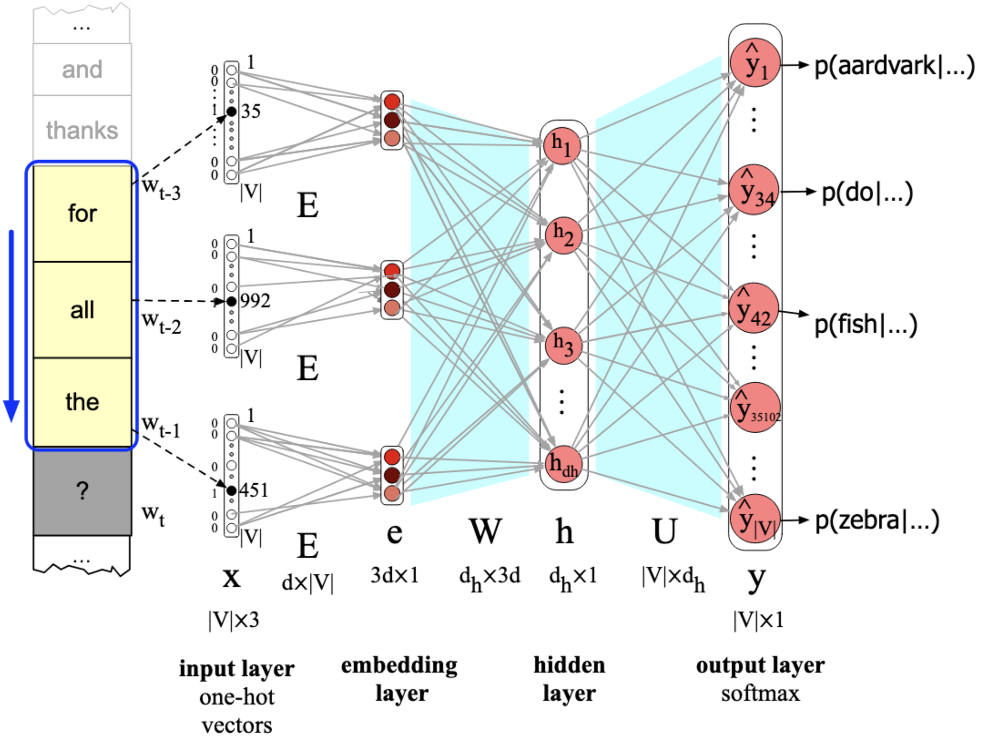


Figure 2.4: Feedforward Neural Network for language modeling.

To train the model and learn the parameters (weights and biases), we can use the **cross-entropy loss** as loss function, and then using the **gradient descent** algorithm to minimize the loss over the training set. The gradients can be computed using the **backpropagation** algorithm, which efficiently computes the gradients by applying the chain rule of calculus. It is based on computation graphs, i.e., representations of the processes of computing mathematical expressions, in which the computation is broken down into separate operations, each of which is modeled as a node in the graph.

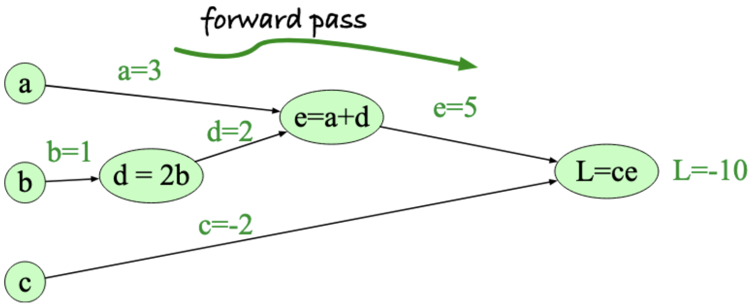


Figure 2.5: Computation graph for a simple feedforward neural network.

On this graph, both a **forward pass** and a **backward pass** can be done, with the first that is used to compute results and the second used to compute gradients. Recalling the chain rule, we can compute the gradient of the loss function with respect to each parameter by multiplying the gradients along the paths from the output node (loss) to the parameter node. This way, we can efficiently compute the gradients for all parameters in the network, allowing us to update them using gradient descent:

$$\frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{dx}$$

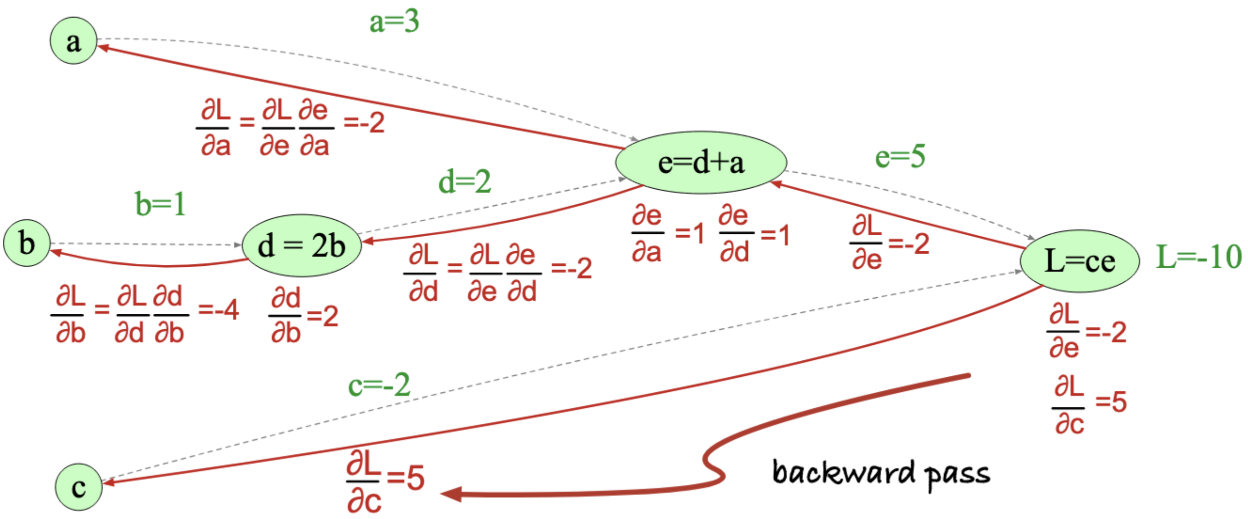

a
a=3
$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$=-2
e=d+a
e=5
b=1
d=2
b
d = 2b
$\frac{\partial e}{\partial a}$=1
$\frac{\partial e}{\partial d}$=1
$\frac{\partial L}{\partial e}$=-2
L=ce
L=-10
$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial d}\frac{\partial d}{\partial b}$=-4
$\frac{\partial d}{\partial b}$=2
$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}$=-2
$\frac{\partial L}{\partial e}$=-2
$\frac{\partial L}{\partial c}$=5
c=-2
c
$\frac{\partial L}{\partial c}$=5
backward pass

Figure 2.6: Backpropagation algorithm.

Observation: ***Common derivatives***

Just a reminder for some common derivatives regarding the activation functions:

- Sigmoid:

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))$$

- Tanh:

$$\frac{d\tanh(z)}{dz} = 1 - \tanh^2(z)$$

- ReLU:

$$\frac{d\mathrm{ReLU}(z)}{dz} = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

Finally, a small change has to be done for training a neural language model, where in this case the parameters to learn are not only the weights and biases of the FFN, but also the word embeddings. During backpropagation, we also compute the gradients of the loss function with respect to the embedding matrix **E**, and update it using gradient descent:

$$\mathbf{E} := \mathbf{E} - \eta \nabla_{\mathbf{E}} L(\hat{y}, y)$$

This way, the model learns embeddings that are useful for the language modeling task, capturing semantic and syntactic properties of words based on their context in the training data.

2.4 Recurrent Neural Networks

A **Recurrent Neural Network** is any network that contains a cycle within its network connections, meaning that the value of some unit is directly or indirecly dependent on its own earlier outputs as an input.

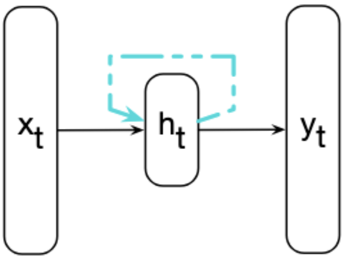
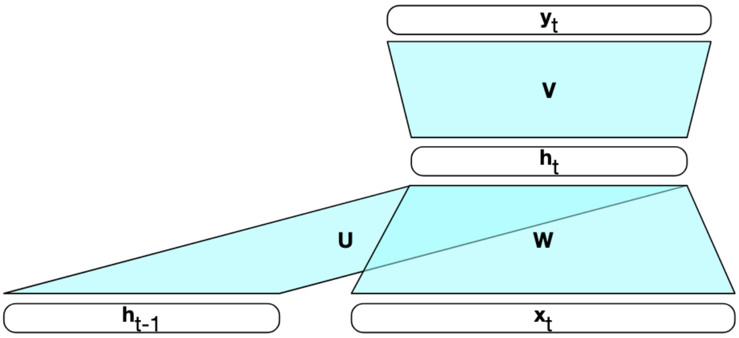


Figure 2.7: Simple RNN.

As with classic feedforward NN, an input vector representing the current input ($\mathbf{x_t}$) is multiplied by a weight matrix and then passed through a non-linear activation function to compute the values for a layer of hidden units. This hidden layer is then used to calculate a corresponding output $\mathbf{y}_t$. The key difference from a FNN is in the recurrent link shown in Fig.2.7 as a dashed line, which basically augments the input to the computation at the hidden layer with the value of the hidden layer from the preceding point in time. The hidden layer from the previous time step provides a sort of **memory**, or context, that encodes earlier processing and informs the decisions to be made at later points in time.

Consequently, there is a new set of weights **U** that connect the hidden layer from the previous step to the current hidden layer.



Forward inference is almost identical to that of a FNN, with the addition of the recurrent connection. At each time step t, the hidden layer $\mathbf{h}_t$ is computed using both the current input $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$:

$$\mathbf{h}_t = \sigma(\mathbf{W}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{b})$$

The output layer is then computed as:

$$\mathbf{z}_t = \mathbf{V}\mathbf{h}_t + \mathbf{c}$$

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

The training process for RNNs is similar to that of FNNs, but this time we use the **Backpropagation Through Time** algorithm, since:

- To compute the loss function for the output at time t we need the hidden layer from time $t-1$;
- The hidden layer at time t influences both the output at time t and the hidden layer at time $t+1$ (hence also the output and loss at time $t+1$).

This means that when we compute the gradients during backpropagation, we need to consider the dependencies across multiple time steps. The Backpropagation Through Time algorithm unfolds the RNN over time, treating it as a deep feedforward network with shared weights across time steps. The gradients are then computed using the chain rule, taking into account the contributions from all time steps.

Finally, for language modeling tasks, forward inference proceeds exactly as described for FNNs, with the addition of the recurrent connection. The input at each time step is the embedding of the current word, and the output is a probability distribution over the vocabulary for the next word. The model can be trained using the cross-entropy loss function, comparing the predicted probabilities with the actual next word in the training data.

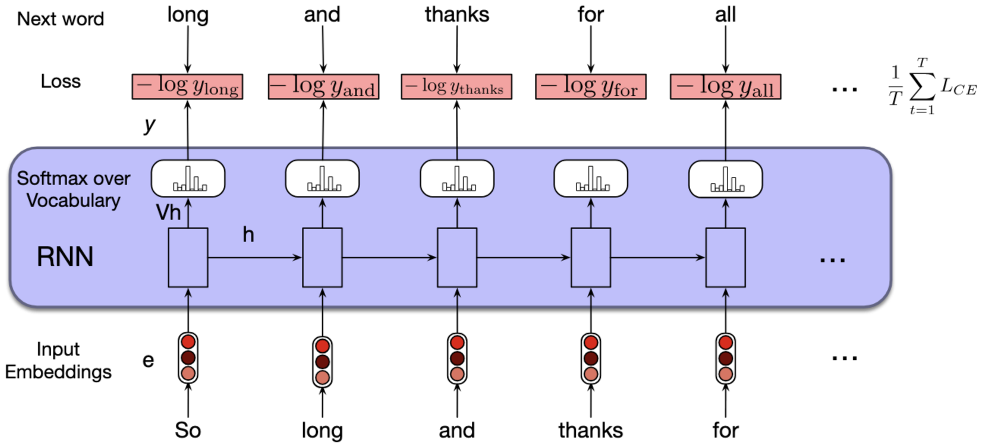


Figure 2.8: Recurrent Neural Network for language modeling.

2.4.1 LSTM

One of the biggest issues with training RNNs is the **vanishing gradient problem**, which occurs when the gradients of the loss function with respect to the model parameters become very small as they are propagated back through time. This can lead to slow convergence during training, and in some cases, the model may fail to learn long-term dependencies in the data. **Long short-term memory (LSTM)** networks are used to address this issue. They divide the context management problem into two sub-problems:

- removing information no longer needed from the context;
- adding information likely to be needed for later decision making.

LSTMs accomplish this by first adding an explicit content layer to the architecture and second by using *gates* to control the flow of information into and out of the units that comprise the network layers. The gates consist of a feedforward layer, followed by a sigmoid activation function and followed by a pointwise multiplication with the layer being gated. This creates a sort of binary mask. Two types of gates are used in LSTMs:

- **Forget gate**: deletes information from the cotext that is no longer needed. It computes a weighted sum of the previous state's hidden layer and the current input and passes that thorough a sigmoid. This mask is then multiplied element-wise by the context vector to remove the information from context that is no longer required.

$$\mathbf{f}_t = \sigma(\mathbf{U}_t\mathbf{h}_{t-1} + \mathbf{W}_t\mathbf{x}_t)$$

$$\mathbf{k}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t$$

Then, we compute the actual information needed to extract from the previous hidden state and current inputs:

$$\mathbf{g}_t = \tanh(\mathbf{U}_g\mathbf{h}_{t-1} + \mathbf{W}_g\mathbf{x}_t)$$

- **Add gate**: selects the information to add to the current context:

$$\mathbf{i}_t = \sigma(\mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{W}_i\mathbf{x}_t)$$

$$\mathbf{j}_t = \mathbf{g}_t \odot \mathbf{i}_t$$

Finally, we update the context vector:

$$\mathbf{c}_t = \mathbf{k}_t + \mathbf{j}_t$$

The final gate is the **output gate**, which controls the information to output from the LSTM unit:

$$\mathbf{o}_t = \sigma(\mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{W}_o\mathbf{x}_t)$$

$$\mathbf{h}_t = \tanh(\mathbf{c}_t) \odot \mathbf{o}_t$$

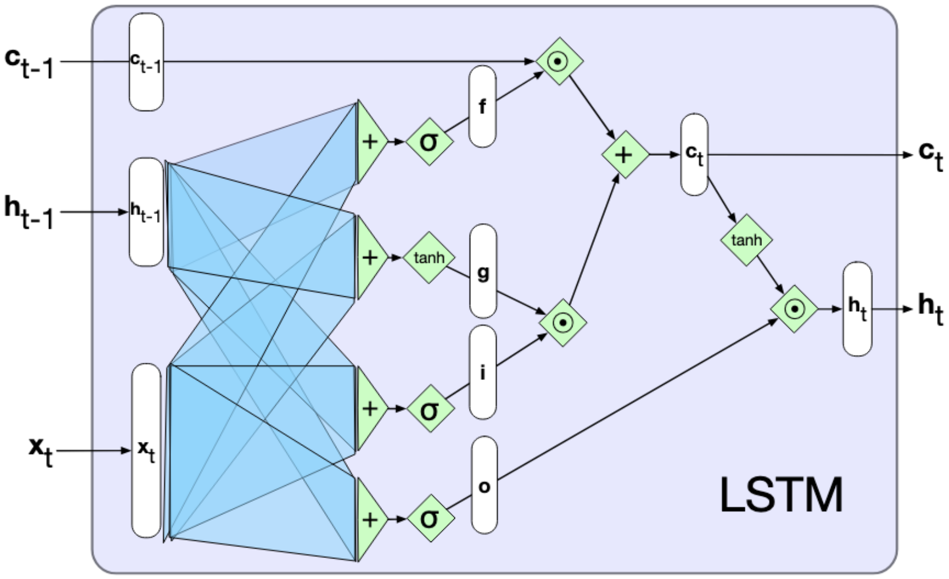


Figure 2.9: LSTM architecture.

2.5 Transformer

Even if the gates allow LSTMs to handle more distant information than RNNs, they don't completely solve the underlying problem: passing information through an extended series of recurrent connections leads to information loss and difficulties in training. **Transformers** map sequences of input vectors $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ to sequences of output vectors $(\mathbf{y}_1, \ldots, \mathbf{y}_n)$ of the same length. They are made of stacks of transformer blocks, which are multilayer networks made by combining simple linear layers, FFNs and **self-attention layers**. Moreover, the computation performed for each item is independent of all the other computations, allowing for parallelization during training.

2.5.1 Self-attention layer

At the core of an attention-based approach is the ability to *compare* an item of interest to a collection of other items in a way that reveals their relevance in the current context. The result of these comparisons is then used to compute an output for the current input. The simplest form of comparison is the dot product:

$$score(\mathbf{x}_i \mathbf{x}_j) = \mathbf{x}_i \mathbf{x}_j \quad \forall j \leq j$$

Then to make an effective use of these scores, they have to be normalized with a softmax to create a vector of weights that indicate the proportional relevance of each input to the input element that is the current focus of attention:

$$\alpha_{ij} = softmax(score(\mathbf{x}_i, \mathbf{x}_j)) = \frac{\exp(\mathbf{x}_i \mathbf{x}_j)}{\sum_{k=1}^{n} \exp(\mathbf{x}_i \mathbf{x}_k)}$$

Finally, the output for the current input element is computed as a weighted sum of the input elements, using the attention weights as coefficients:

$$\mathbf{y}_i = \sum_{j=1}^{n} \alpha_{ij} \mathbf{x}_j$$

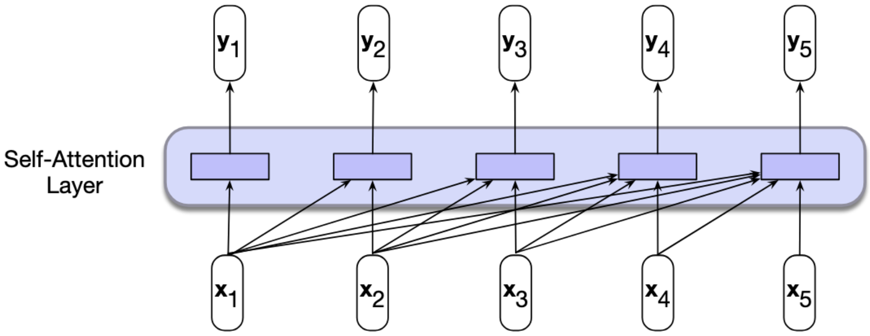


Figure 2.10: Self-attention layer.

Each input embedding plays three distinct roles in the self-attention computation:

- As a **query** vector, representing the item for which we are computing attention;
- As a **key** vector, representing each item in the collection being attended to;
- As a **value** vector, representing the actual content of each item in the collection.

To implement these three roles, we use three different weight matrices ($\mathbf{W}_Q$, $\mathbf{W}_K$, $\mathbf{W}_V$) to project the input embeddings into three different vector spaces:

$$\mathbf{q}_i = \mathbf{W}_Q \mathbf{x}_i \quad \mathbf{k}_i = \mathbf{W}_K \mathbf{x}_i \quad \mathbf{v}_i = \mathbf{W}_V \mathbf{x}_i$$

The attention scores are then computed using the query and key vectors:

$$score(\mathbf{q}_i, \mathbf{k}_j) = \mathbf{q}_i \cdot \mathbf{k}_j$$

However, since the dot products can grow large in magnitude, leading to small gradients during training, we scale the scores by the square root of the dimensionality of the key vectors (d_k):

$$score(\mathbf{q}_i, \mathbf{k}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}$$

The attention weights are then computed using the softmax function:

$$\alpha_{ij} = softmax\left(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}\right) = \frac{\exp\left(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}\right)}{\sum_{l=1}^{n} \exp\left(\frac{\mathbf{q}_i \cdot \mathbf{k}_l}{\sqrt{d_k}}\right)}$$

Finally, the output for the current input element is computed as a weighted sum of the value vectors:

$$\mathbf{y}_i = \sum_{j=1}^{n} \alpha_{ij} \mathbf{v}_j$$

Since each output is computed independently, this process can be parallelized by taking advantage of efficient matrix multiplication routines and pack the input embeddings of the N tokens into a single matrix $\mathbf{X} \in \mathbb{R}^{N \times d}$. We then multiply it by the key, query and value matrices (all of dimensionality $d \times d$) to produce the matrices $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q \quad \mathbf{K} = \mathbf{X}\mathbf{W}_K \quad \mathbf{V} = \mathbf{X}\mathbf{W}_V$$

So, the complete self-attention computation can be summarized as:

$$SelfAttention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

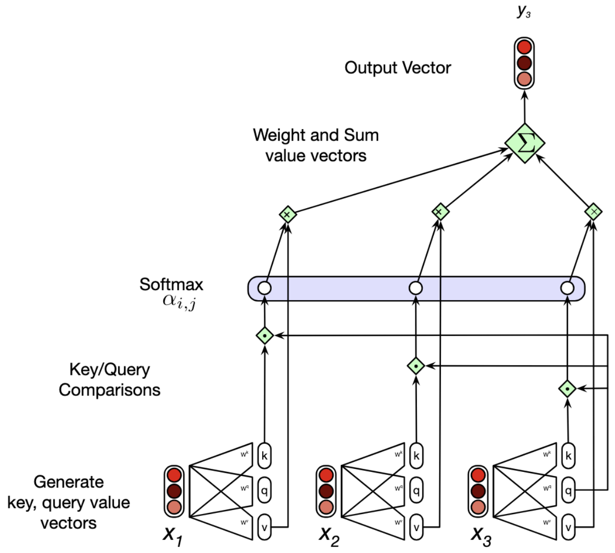


Figure 2.11: Masked self-attention mechanism.

Observation: ***Matrix dimensions***

The inputs and outputs of transformers, as well as the intermediate vectors after the various layers, all have the same dimensionality $1 \times d$. For now you can assume that the dimensionalities of the transform matrices are all $d \times d$, but consider that in **Multi-head attention mechanisms**, which will be explained later, the dimensionalities change and every head has its own set of matrices with different dimensions.

Finally, a mask has to be applied to the upper triangle of the score matrix during training, to prevent the model from attending to future tokens. This is done by setting the scores for all positions (i, j) where $j > i$ to $-\infty$ before applying the softmax function. This way, the attention weights for these positions will be zero, effectively preventing the model from attending to future tokens.

2.5.2 Transformer Blocks

Every transformer block consists of a single attention layer followed by a fully-connected feedforward layer with **residual connections** and **layer normalizations** following each, as shown in Fig.2.12.

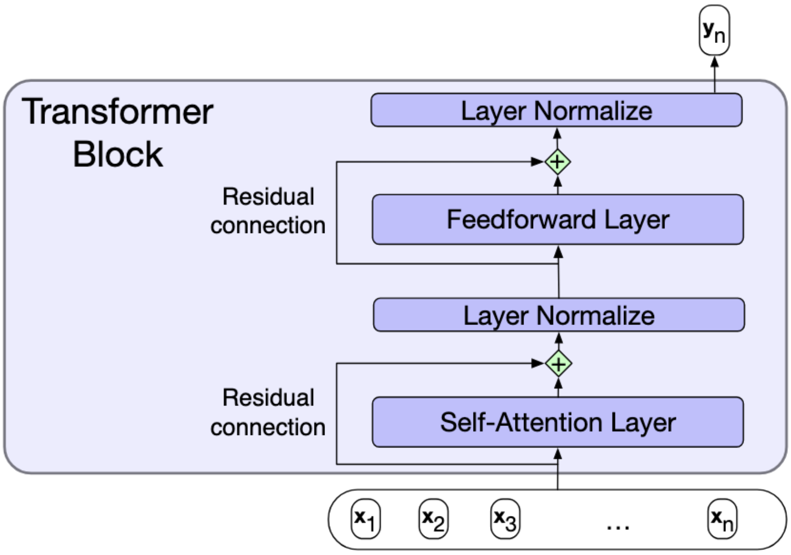


Figure 2.12: Transformer block architecture.

Residual connections are connections that pass information from a lower layer to a higher layer without going through the intermediate layer. That said, the output vector is the sum of the input vector and the hidden vector computed with the attention or feedforward layer:

$$\mathbf{y} = \mathbf{x} + Layer(\mathbf{x})$$

Layer normalization, on the other hand, is a technique used to normalize the activations of a layer across the features dimension. It helps to stabilize the training process and improve convergence by reducing internal covariate shift. The layer normalization can be applied before or after the residual connection (the original paper uses post-norm, but pre-norm is more common nowadays). The layer normalization is computed as:

$$\text{LayerNorm}(\mathbf{x}) = \frac{\mathbf{x} - \mu}{\sigma} \odot \gamma + \beta$$

where μ and σ are the mean and standard deviation of the elements in $\mathbf{x}$, and γ and β are learnable parameters that scale and shift the normalized output.

Since different words in a sentence can relate to each other in many different ways simultaneously, **multi-head attention layers** are used to learn different aspects of the relationships that exist among inputs at the same level of abstractions. They reside in parallel layers at the same depth, each with its own set of parameters. Each head i has its own set of key, query and value matrices ($\mathbf{W}_K^i$, $\mathbf{W}_Q^i$, $\mathbf{W}_V^i$) to project the input embeddings into three different vector spaces. The outputs of all heads are then concatenated and projected back into the original space using a final weight matrix $\mathbf{W}_O$:

$$MultiHead(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2 \oplus \cdots \oplus \mathbf{head}_h)\mathbf{W}_O$$

Just a reminder that in multi-head attention, instead of using the model dimension d, the query and key embeddings have dimensionality d_k, and the value embeddings have dimensionality d_v.

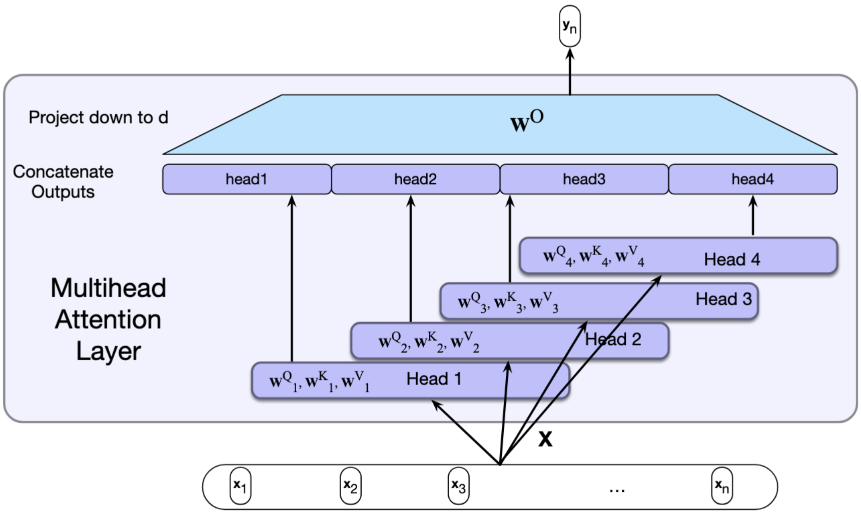

y_n
Project down to d
W^O
Concatenate Outputs
head1
head2
head3
head4
W^Q_4, W^K_4, W^V_4 Head 4
W^Q_3, W^K_3, W^V_3 Head 3
W^Q_2, W^K_2, W^V_2 Head 2
W^Q_1, W^K_1, W^V_1 Head 1
Multihead Attention Layer
X
x_1
x_2
x_3
...
x_n

Figure 2.13: Multi-head attention mechanism.

Positional embeddings are also added to the word embeddings to give a sense of word order. They are learned as well during training.

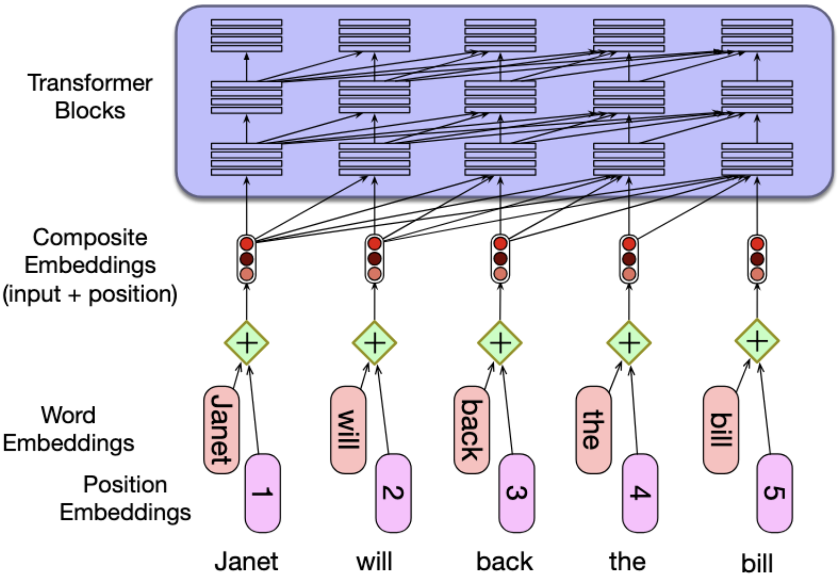


Figure 2.14: Transformer architecture.

⚠ **Warning:**

A potential problem using absolute position embeddings is that there will be plenty of training examples for the initial positions in the inputs and fewer at the outer length limits. This could lead to poor generalization during testing. An alternative approach is to use a **static function** that maps integer inputs to real-valued vectors in a way that captures the inherent relationships among the positions.

2.6 Encoder-Decoder Models

In the previous section, the basic block of the transformer architecture was presented. However, the first use of the architecture was for a different case: transforming one sequence of tokens into another to enable machine translation between languages. For this use case, a two-phase system was designed: encoding and decoding, where the transformer from the previous section corresponds to the second phase.

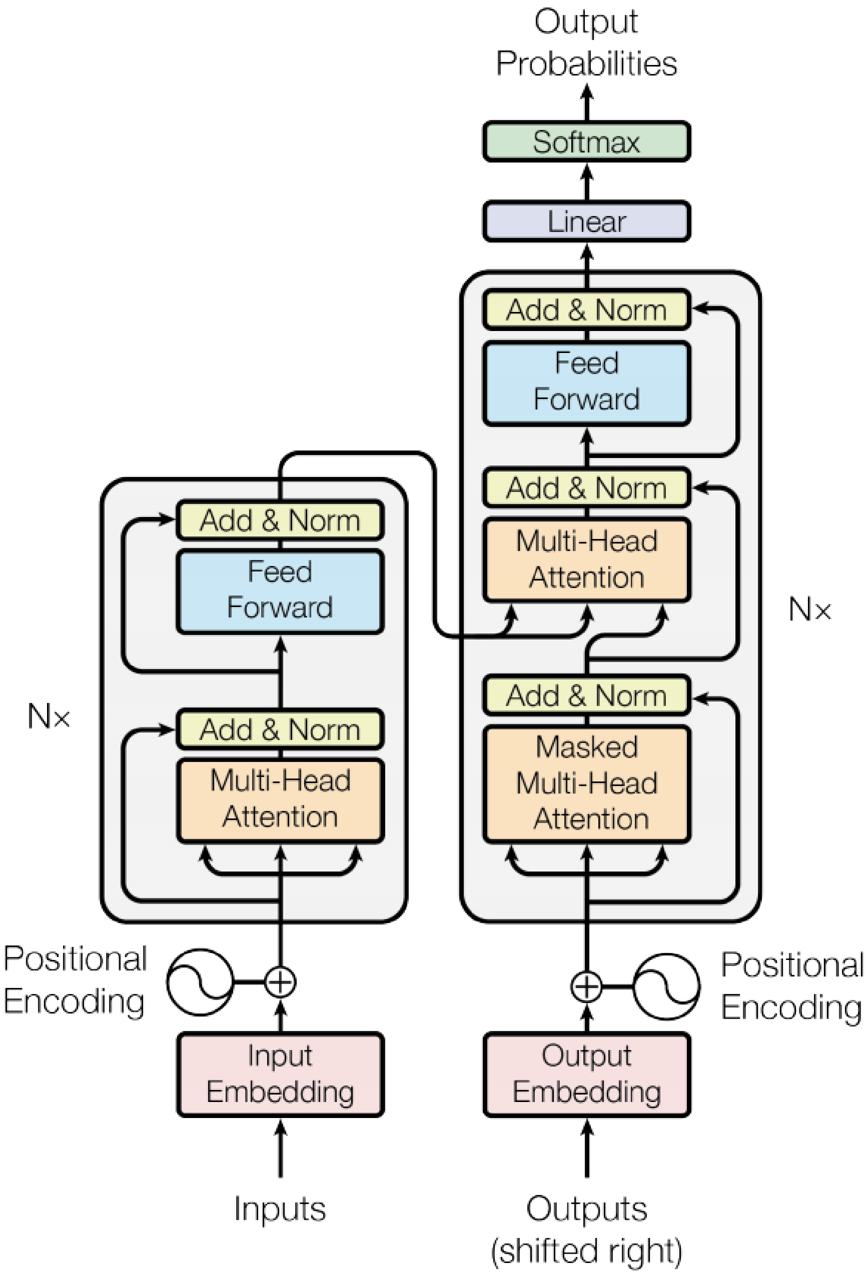
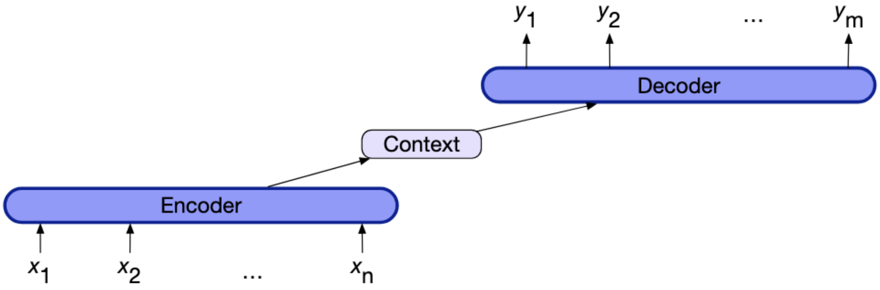


Figure 2.15: Encoder-Decoder architecture described in the original transformer paper.

As shown in Fig.2.15, the transformer architecture described before is just the decoder part (on the right), for which we can remove the mixed multi-head attention if we don't consider the encoder. The **encoder**, on the other part, is used for different purposes. Together, they form the **encoder-decoder architecture**, which is widely used for sequence-to-sequence tasks such as machine translation, text summarization, and question answering. It is able to generate contextually appropriate, arbitraty length output sequences. The encoder, in fact, takes an input sequence and creates a contextualized representation of it, called the **context**. It is then passed to the decoder which generates a task-specific output sequence.



Machine Translation (MT) systems generally use a fixed vocabulary generated with the **BPE** or **wordpiece** algorithms. Both of them are **subword tokenization** methods, meaning that they break down rare words into smaller, more frequent subword units. This allows the model to handle out-of-vocabulary words and reduces the overall vocabulary size, making training more efficient. Both also have a **token learner**, which takes row training corpus and induces a vocabulary, and a **token segmenter**, which takes a row test sentence and segments it into the tokens in the vocabulary. BPE token learner begins with a vocabulary that is just the set of individual characters. It then examines the training corpus, chooses the two symbols that are most frequently adjacent, adds a new merger symbol to the vocabulary and replaces every adjacent occurrence of the two symbols with the new symbol. This process continues until k merges have been done creating k novel tokens. Once we've learned our vocabulary, the token segmenter can segment new sentences by greedily matching the longest possible tokens from the vocabulary. Wordpiece, instead, first initializes the lexicon with characters, then it trains an n-gram language model on the training corpus and, considering the set of possivle new wordpieces made by concatenating two wordpieces from the current lexicon, chooses the one that most increases the language model probability of the training corpus. Repeat the last two steps until there are V wordpieces.

2.6.1 Beam Search

Choosing the single most probable token to generate at each step in Machine Translation is called **greedy decoding**. This is not optimal, since the token that looks good to the decoder now might not turn out later to have been the wrong choice. Thus, decoding in MT and other sequence generation problems generally uses a method called **beam search**. In it, instead of choosing the best token to generate at each timestep, we keep k possible tokens at each step. This fixed-size memory footprint k is called the *beam width*. These initial k outputs are the search frontier and these k initial words are called *hypotheses*.

At subsequent steps, each of the k best hypotheses is extended incrementally by being passed to distinct decoders, which each generate a softmax over the entire vocabulary to extend the hypothesis to every possible next token. Each of these $k \times V$ hypotheses is scored by $P(y_i|x, y_{<i})$: the product of the probability of current word choice multiplied by the probability of the path that led to it. We then prune the $k \times V$ hypotheses down to the k best ones, so there are never more than k hypotheses at the frontier of the search, and never more than k decoders.

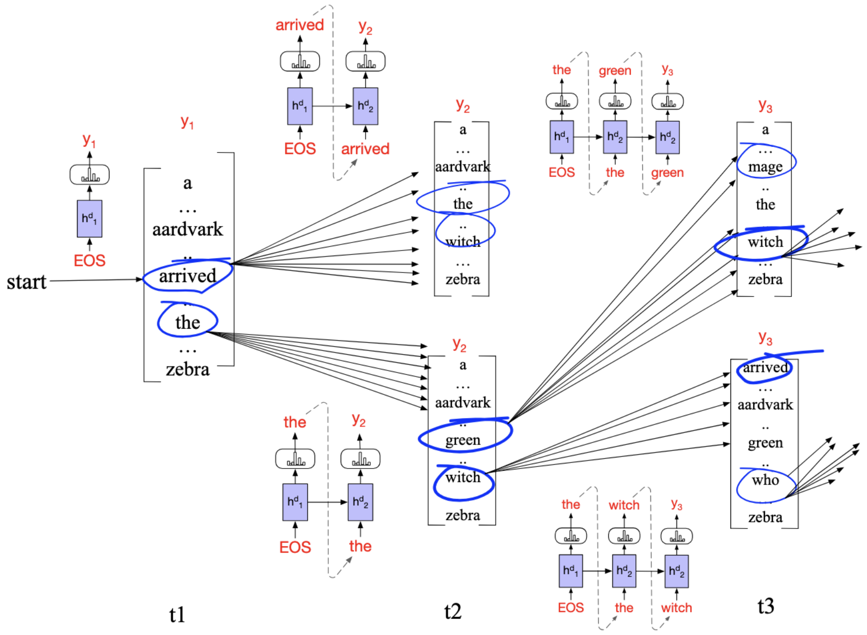


Figure 2.16: Beam search with beam width of 2.

⚠ Warning:

One problem arises from the fact that the completed hypotheses may have different lengths. Because models generally assign lower probabilities to longer strings, a naive algorithm would also choose shorter strings for *y*.

2.6.2 Pre-trained Models

If we want to assign the correct named-entity tag to each word in a sentence, or other sophisticated labels like the parse tags, we'll want to be able to take into account information from the right context as we process each element. **Birirectional encoders** are designed to do just that by allowing the self-attention mechanism to range over the entire input. They underlies models like BERT and its descendants like RoBERTa or SpanBERT.

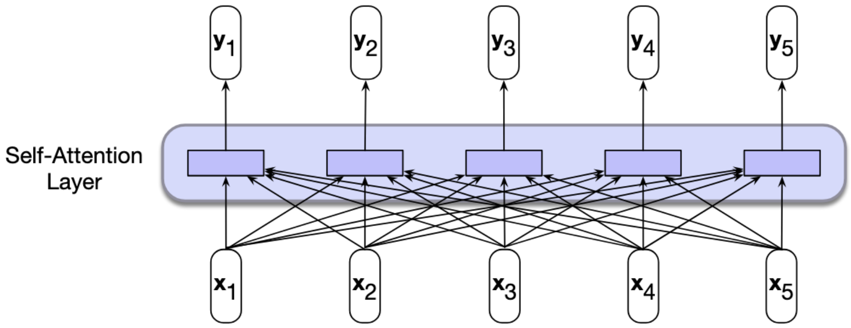


Figure 2.17: Bidirectional encoder self-attention mechanism.

The focus of bidirectional encoders is on computing contextualized representations of the tokens in an input sequence that are generally useful across a range of downstream applications. This contextualization is accomplished through the use of the same self-attention mechanism used in causal models, the only difference is that here we skip the masking step, allowing the model to contextualize each token using information from the entire input.

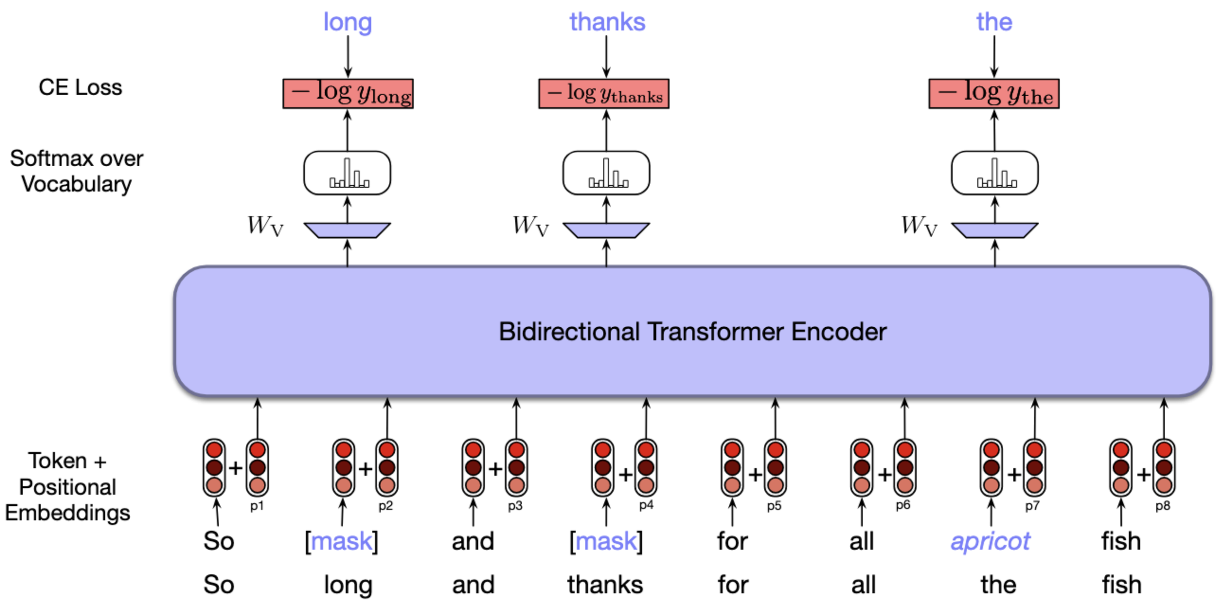


Figure 2.18: Masked Language Modeling training process.

Observation:

In BERT and descendants, the WordPiece algorithm is used for tokenization, creating subwords. This will require that for some NLP tasks that require notions of words we will occasionally need to map subwords back to words.

For training, bidirectional encoders learn to perform a fill-in-the-blank task, technically called the **cloze task**. That is, given an input sequence with one or more elements missing, the learning task is to predict the missing elements. More precisely, during training the model is deprived of one or more elements of an input sequence and must generate a probability distribution over the vocabulary for each of the missing items. The original approach to training bidirectional encoders is called **Masked Language Modeling (MLM)**, which uses unannotated text from large corpus. The model is presented with a series of sentences from the training corpus where a random sample of tokens from each training sequence is selected for use in the learning task. Once chosen, a token is used in one of three ways:

- It is replaces with the unique vocabulary token `[MASK]` → 80%;
- It is replaces with another token from the vocabulary, randomly sampled based on token unigram probabilities → 10%;
- It is left unchanged → 10%.

In BERT, 15% of the input tokens in a training sequence are sampled for learning.

Observation:

Note that all of the input tokens play a role in the self-attention process, but only the sampled tokens are used for learning.

To produce a probability distribution over the vocabulary for each of the masked tokens, the output vector from the final transformer layer for each of the masked tokens is multiplied by a learned set of classification weights $\mathbf{W}_V \in \mathbb{R}^{|V| \times d_h}$ and then through a softmax to yield the required predictions over the vocabulary:

$$\hat{\mathbf{y}}_i = softmax(\mathbf{W}_V \mathbf{h}_i)$$

For many NLP applications, the natural unit of interest may be larger than a single word or token. Question answering, syntactic parsing, coreference and semantic role labeling applications all involve the identification and classification of constituents or phrases. This suggests a **span-oriented** masked learning objective mighe provide improved performance on such tasks. A span is a contiguous sequence of one or more words selected from a training text, prior to subword tokenization. In span-based masking, a set of randomly selected spans from a trainig sequence are chosen. Once a span is chosen for masking, all the words within the span are substituted according to the same regime used in BERT. Moreover, the SpanBERT learning objective augments the MLM objective with a boundary oriented component called the **Span Boundary Objective (SBO)**, which relies on a model's ability to predict the words within a masked span from the words immediately preceding and following it.

$$L(x) = L_{MLM}(x) + L_{SBO}(x)$$

$$L_{SBO}(x) = -\log P(x|x_s, x_e, p_x)$$

where s denotes the position of the word before the span and e denotes the word after the end.

$$s = FFNN([y_{s-1}; y_{e+1}; p_{i-s+1}])$$

$$z = softmax(Es)$$

The final loss is the sum of the MLM loss and the SBO loss.

An important class of applications involves determining the relationship between pairs of sentences. These include tasks like paraphrase detection, entailment or discourse coherence. To capture the kind of knowledge required for applications such as these, BERT introduced a second learning objective called **Next Sentence Prediction (NSP)**, where the model is presented with pairs of sentences and is asked to predict whether each pair consists of an actual pair of adjacent sentences from the training corpus or a pair of unrelated sentences. To facilitate this, BERT introduces two new tokens to the input representation. After tokenizing the input with the subword model, the token `[CLS]` is prepended to the input sentence pair, and the token `[SEP]` is placed between the sentences and after the final token of the second sentence.

During training, the output vector from the final layer associated with the `[CLS]` token represents the next sentence prediction. A learned set of classification weights $\mathbf{W_{NSP}} \in \mathbb{R}^{2 \times d_h}$ is used to produce a two-class prediction from the raw `[CLS]` vector.

$$\hat{\mathbf{y}}_i = softmax(\mathbf{W_{NSP}} \mathbf{h}_i)$$

The power of pretrained language models lies in their ability to extract generalizations from large amounts of text. To make practical use of these generalizations, we need to create interfaces from these models to downstream applications through a process called **fine-tuning**. Most common applications are:

- **Sequence Classification**: An additional vector is added to the model to stand for the entire sequence. This vector is sometimes called the *sentence embedding* since it refers to the entire sequence. In BERT, the `[CLS]` token plays the role of this embedding. This unique token is added to the vocabulary and is prepended to the start of all input sequences. The output vector in the final layer of the model for the `[CLS]` input represents the entire input sequence and serves as the input to a *classifier head*, a logistic regression or NN classifier that makes the relevant decision. A key difference from what we've seen earlier with neural classifier is that this loss can be used to not only learn the weights of the classifier, but also to update the weights for the pretrained language model itself.

- **Pair-wise Sequence Classification**: Fine-tuning an application for one of these tasks proceeds just as with pretraining using the NSP objective. During fine-tuning, pairs of labeled sentences from the supervised learning data are presented to the model. The output vector associated with the prepended `[CLS]` token represents the model's view of the input pair. And as NSP training, the two inputs are separated by the `[SEP]` token.
- **Sequence Labelling**: Here, the final output vector corresponding to each input token is passed to a classifier that produces a softmax distribution over the possible set of tags. Again, assuming a simple classifier consisting of a single feedforward layer followed by a softmax, the set of weights to be learned for this additional layer is $\mathbf{W_K} \in \mathbb{R}^{k \times d_h}$, where k is the number of possible tags for the task. A complication with this approach arises from the use of subword tokenization. Supervised training data for tasks like named entity recognition is typically in the form of BIO tags associated with text segmented at the word level. To deal with this, we need a way to assign BIO tags to subword tokens dueing training and a corresponding way to recover word-level tags from subwords during decoding. For training, we can just assign a gold-standard tag associated with each word to all of the subwords token derived from it. For decoding, the simplest approach is to use the argmax BIO tag associated with the first subword token of a word.

2.7 Prompting

One key emergent ability from GPT-2 is **zero-shot learning**, i.e., the ability to do many tasks with no examples and no gradient updates, by simply:

- Specifying the right sequence in prediction problem (e.g. question answering) → ”Passage: Tom Brady ... Q: Where was Tom Brady born? A: ...”;
- Comparing probabilities of sequences (e.g. classification tasks) → ”The cat couldn't fit into the hat because it was too big.”, does ”it” refers to the cat or to the hat?

GPT-2 beats state-of-the-art models on language modeling benchmarks with **no task-specific fine-tuning**. Moreover, with bigger models (GPT-3), we can see **few-shot learning**, where the model is given a few examples of the task in the prompt, and it can generalize to new examples without any gradient updates. It is also called **in-context learning**. Therefore, there exist another method to prompt the model, using this ability to learn in context without having to fine-tune the model and update weights.

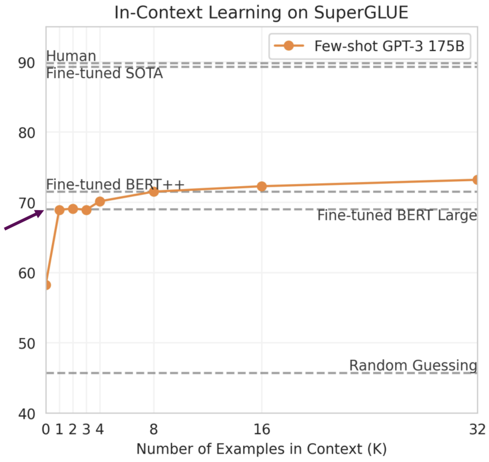


Figure 2.19: Prompting methods: zero-shot, one-shot and few-shot learning.

However, some tasks seem to hard for even large LMs to learn through prompting alone, see for example operations with large numbers, where tasks require multi-step reasoning. Therefore, **chain-of-thought prompting** has evolved, where the idea is to demonstrate the process we want the model to follow, by providing intermediate reasoning steps in the prompt. This way, the model can decompose complex problems into simpler sub-problems, and solve them step by step.

Example: ***Chain-of-thought prompting***

Model Input
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: The answer is 11.
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
Model Output
A: The answer is 27. **Wrong!**
Using chain-of-thought prompting:
Model Input
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: He starts with 5 tennis balls. He buys 2 cans, each with 3 balls, so he gets 2 * 3 = 6 more balls. In total, he has 5 + 6 = 11 tennis balls.
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
Model Output
A: They start with 23 apples. They use 20 for lunch, leaving them with 23 - 20 = 3 apples. They then buy 6 more apples, so they have 3 + 6 = 9 apples. **Correct!**

This chain-of-thought prompt is an emergent property of model at scale (e.g. GPT-3 with 175B parameters). Smaller models don't seem to benefit from it.

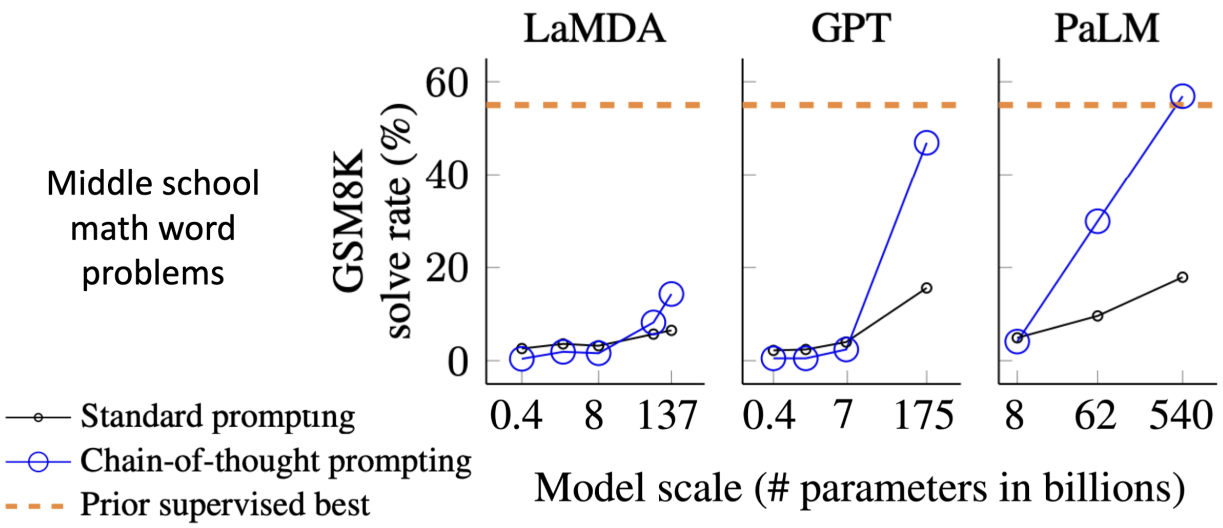
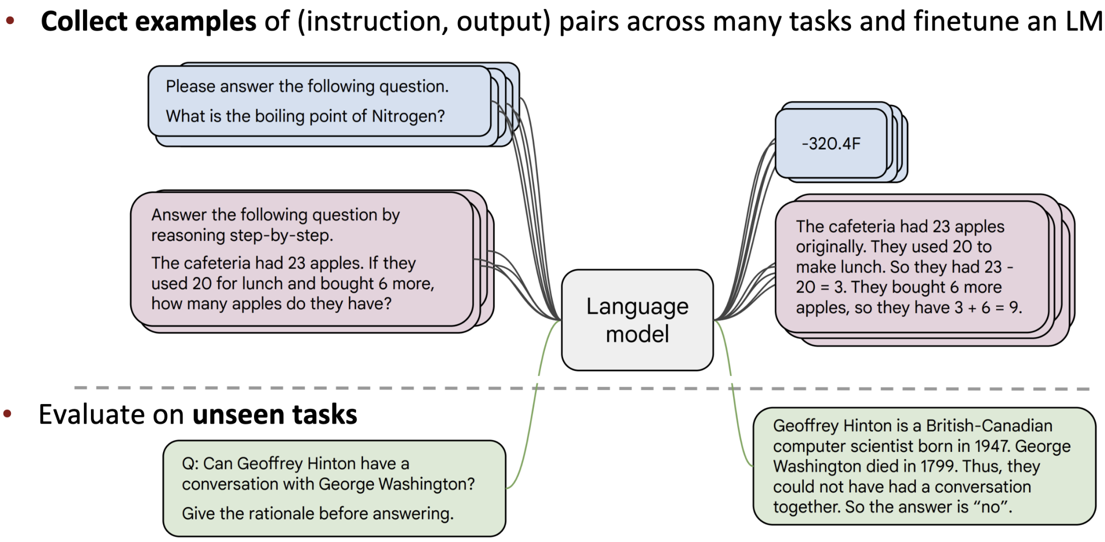


Figure 2.20: Chain-of-thought prompting performance on arithmetic reasoning tasks.

An even better approach that was tested is the use of phrases like "Let's think step by step", with which the performance improves even more.

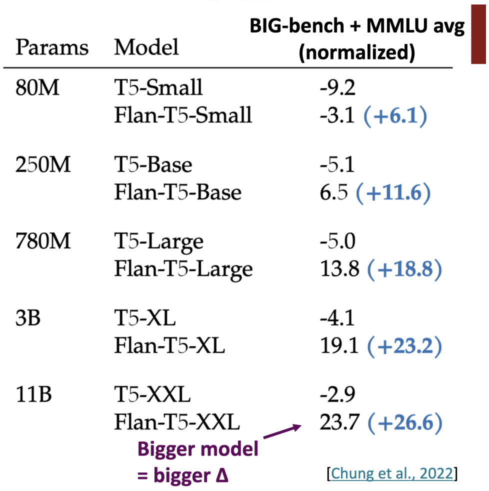
The negative sides of this approach are in the limits to what we can fit in context and in the complex tasks that still need gradient steps. Moreover, language modeling does not mean alignment with human assistance, since they are just made to predict the next most probable token. Therefore, we can do fine-tuning on some tasks, i.e. **instruction fine-tuning**. To do this, we collect examples of (instruction, output) tuples across many tasks and fine-tune an LM. We then evaluate on unseen tasks. As always, the key is to use more data and a model that can scale.



- **Collect examples** of (instruction, output) pairs across many tasks and finetune an LM
- Evaluate on **unseen tasks**

Please answer the following question.
What is the boiling point of Nitrogen?
Answer the following question by reasoning step-by-step.
The cafeteria had 23 apples. If they used 20 for lunch and bought 6 more, how many apples do they have?
Language model
-320.4F
The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9.
Q: Can Geoffrey Hinton have a conversation with George Washington?
Give the rationale before answering.
Geoffrey Hinton is a British-Canadian computer scientist born in 1947. George Washington died in 1799. Thus, they could not have had a conversation together. So the answer is "no".

Figure 2.21: Instruction fine-tuning.



Params	Model	BIG-bench + MMLU avg (normalized)
80M	T5-Small Flan-T5-Small	-9.2 -3.1 (+6.1)
250M	T5-Base Flan-T5-Base	-5.1 6.5 (+11.6)
780M	T5-Large Flan-T5-Large	-5.0 13.8 (+18.8)
3B	T5-XL Flan-T5-XL	-4.1 19.1 (+23.2)
11B	T5-XXL Flan-T5-XXL	-2.9 23.7 (+26.6)

Bigger model = bigger Δ

[Chung et al., 2022]

Figure 2.22: Performance of prompting vs. fine-tuning.

⚠ **Warning**: ***Limitations***

This approach remains first of all **expensive**: we need to collect much ground-truth data. Moreover, there are two main problems:

- Tasks like open-ended creative generation have no right answer;
- Language modeling penalizes all token-level mistakes equally, but some errors are worse than others. Even with instruction fine-tuning, a mismatch between the LM objective and the objective of satisfying human preferences exists.

For those limitations, **Reinforcement Learning with Human Feedback (RLHF)** has been developed. Imagine having a way to get a *human reward* regarding an answer, where higher is better. We want to maximize the expected reward of samples from our LM:

$$\mathbb{E}_{\hat{s}\sim P_\theta(s)}\left[R(\hat{s})\right]$$

We update the parameters of the LM with gradient ascent:

$$\theta_{t+1} := \theta_t + \alpha\nabla_\theta\mathbb{E}_{\hat{s}\sim P_\theta(s)}\left[R(\hat{s})\right]$$

Policy gradient methods in RL (e.g. REINFORCE) give us a way to estimate and optimize this objective. The main problem is the almost infinite search space. Since we want

$$\nabla_\theta\mathbb{E}_{\hat{s}\sim P_\theta(s)}\left[R(\hat{s})\right] = \nabla_\theta\sum_s R(s)p_\theta(s) = \sum_s R(s)\nabla_\theta p_\theta(s)$$

we can use the log-derivative trick to take the gradient of $\log p_\theta(s)$

$$\nabla_\theta \log p_\theta(s) = \frac{1}{p_\theta(s)}\nabla_\theta p_\theta(s) \quad \rightarrow \quad \nabla_\theta p_\theta(s) = \nabla_\theta \log p_\theta(s)p_\theta(s)$$

and plug it back in the previous equation:

$$\nabla_\theta\mathbb{E}_{\hat{s}\sim P_\theta(s)}\left[R(\hat{s})\right] = \sum_s R(s)\nabla_\theta \log p_\theta(s)p_\theta(s) = \mathbb{E}_{\hat{s}\sim P_\theta(s)}\left[R(\hat{s})\nabla_\theta \log p_\theta(\hat{s})\right]$$

Now we can approximate the expectation with Monte Carlo sampling:

$$\mathbb{E}_{\hat{s}\sim P_\theta(s)}\left[R(\hat{s})\nabla_\theta \log p_\theta(\hat{s})\right] \approx \frac{1}{N}\sum_{i=1}^{N} R(\hat{s}_i)\nabla_\theta \log p_\theta(\hat{s}_i) \quad \hat{s}_i \sim P_\theta(s)$$

giving us the update rule:

$$\theta_{t+1} := \theta_t + \frac{\alpha}{N}\sum_{i=1}^{N} R(\hat{s}_i)\nabla_\theta \log p_\theta(\hat{s}_i)$$

The problem is that collecting human feedback is expensive and slow. Therefore, we can model their prefereices as a separate (NLP) problem. We can train an LM to predict human preferences from an annotated dataset, then optimize for the predicted reward instead of the actual human reward. This is called **reward modeling**. The second main problem is that human judgements are noisy and miscalibrated. It has been demonstrated, in fact, that instead of asking for direct ratings, asking for **pairwise comparisons** between two outputs leads to more reliable results. Therefore, the reward model is trained to predict which of two outputs is preferred by humans. Given two outputs $\hat{s}_1$ and $\hat{s}_2$, the probability that $\hat{s}_1$ is preferred over $\hat{s}_2$ is modeled as:

$$J_{RM}(\phi) = -\mathbb{E}_{(s^w,s^l)\sim D}\left[\log\sigma(RM_\phi(s^w) - RM_\phi(s^l))\right]$$

where s^w is the winning sample, s^l is the losing sample, and σ is the sigmoid function.

Now, to put all together:

- Initialize a copy of the model $p_\theta^{RL}(s)$ with parameters θ;
- Optimize the following reward objective with policy gradient:

$$R(s) = RM_\phi(s) - \beta\log\left(\frac{p_\theta^{RL}(s)}{p^{PT}(s)}\right)$$

  where $p^{PT}(s)$ is the pretrained LM, and β is a hyperparameter that controls the strength of the KL regularization; We pay a price when $p_\theta^{RL}(s) > p^{PT}(s)$.

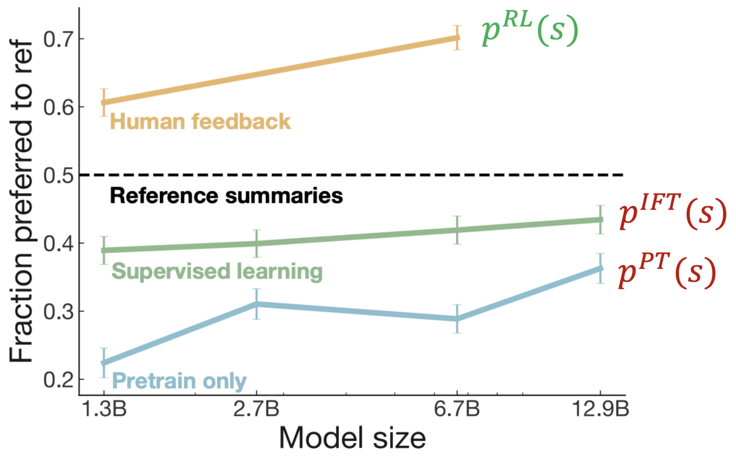


Figure 2.23: RLHF provides gains over pretraining + finetuning.

⚠ **Warning**: ***Limitations of RLHF***

RLHF is not a silver bullet. Some limitations are:

- Reward hacking: the model may find ways to exploit weaknesses in the reward model;
- Catastrophic forgetting: the model may forget how to perform well on the original language modeling task;
- Scalability: collecting human feedback and training reward models can be expensive and time-consuming.

2.8 Multilingual Models

So far, in our exploration of transformer-based natural language processing models, we have primarily considered systems that work on a single language or, at least, we have not explicitly considered the possibility of using a single model for multiple languages. However, as we will see in this section, models can handle multiple languages simultaneously, and there are significant advantages to doing so. One of these advantages stems from a phenomenon known as **transfer learning**, where knowledge gained about one language in the representations learned by the neural network can be applied to other languages. This is particularly valuable for *low-resource* languages, that is, languages with fewer training resources, as they may end up benefiting from the knowledge gained by the model from other languages.

Training a multilingual transformer model is not very different from training a monolingual model. The main difference lies in the training data, which is a mixture of texts in different languages. The tokenizer is created using the entire multilingual corpus with systems such as SentencePiece or byte pair encoding (BPE). If the languages are related, some tokens may be shared, which helps the neural network learn joint representations, also known as **cross-lingual embeddings**. For example, consider *centro*, *centru*, *centre*, and *center* in Spanish, Asturian, Catalan, and English, respectively; if these words are tokenized as centr and the remaining suffix, the model can learn a representation for *centr* from Spanish, Catalan, or English texts and apply it to Asturian inputs, even if the word *centru* has not been seen in the Asturian training data. Names of people or places can also be easily shared across languages. In the common case of data imbalance, where there are significantly more texts in some languages than in others, it is common to use a rebalancing formula to upsample languages with less data, ensuring that their texts are overrepresented in the training or evaluation datasets, as well as in the corpus used to train the tokenizer. Interestingly, even without shared tokens, words like *atristuráu* (Asturian) and *sad* (English) might end up with similar embeddings in certain layers of the model. This is another example of the language-independent nature of some of the representations learned. Additionally, depending on the task, language-specific tokens can be used as the first token to indicate the language of the text. This practice is more common in encoder-decoder models (e.g., to indicate the target language) than in decoder-only models. These special tokens are added to the vocabulary, prepended to every sentence during both training and inference, and learned in the same way as the `MASK` token in BERT-like pretraining. Decoder-like language models are usually trained with multilingual data from the beginning but do not require language-specific tokens. The language used in the prompt is sufficient to guide the generation process toward the desired language.

Early pre-trained models, such as BERT, were English-centric. Over time, variants emerged for other languages, like CamemBERT for French, GilBERTo for Italian, or BERTo for Spanish. These models, however, were still essentially monolingual. A turning point came with pre-trained multilingual models like mBERT or XLM-R, which support around a hundred languages, and more recently, Glot500, which extends to 500 languages. These models are self-supervisedly trained with neutral tasks, such as masked language modeling, making them general-purpose models that can be fine-tuned for specific tasks in any of the supported languages. There are also encoder-decoder multilingual models like mBART, mT5 or NLLB-200. More recently, multilingual decoder-only models trained as large language models and covering a wide range of languages have started to emerge, such as EMMA-500, EuroLLM or Aya. A notable phenomenon here is the zero-shot generalization ability of the model. For example, in a named entity recognition task, a model fine-tuned only with English texts can be applied to texts in other languages without requiring labeled data in those languages, thanks to the multilingual representations learned during pretraining. This is particularly useful for low-resource languages.

⚠ **Warning**: ***Challenges***

Multilingual models often face the challenge known as the curse of *multilinguality* (a term inspired by the *curse of dimensionality* in statistics and machine learning). This phenomenon refers to a decline in performance for individual languages as the model expands to accommodate a larger number of languages. Languages with more resources often benefit the least from the multilingual model and may even exhibit lower performance compared to a monolingual model trained with the same data. Several techniques have been proposed to mitigate the curse of multilinguality. One approach is the use of **adapters**, small trainable modules (e.g., a feedforward network), one for each language, inserted at specific points within the transformer layers. During training, common parameters are learned as usual, while adapter parameters are updated only for their associated language. This allows the model to learn a mix of cross-lingual and language-specific parameters. Adapters are also used to fine-tune pre-trained models for new tasks or languages without retraining the entire model. This preserves the original model's weights, updating only the adapter parameters. Other alternatives for using monolingual models in multilingual scenarios, such as *translate-train* or *translate-test*, are also possible. In the first case, training data available in one language is translated into other languages, and a multilingual model is trained with both the source and translated data. In the second case, test data is translated into the language of a monolingual model at inference time. In both cases, machine translation systems perform the translation. Finally, several multilingual datasets are available for different tasks, such as the universal dependencies treebanks, the XNLI dataset for natural language inference, or the Seed or FLORES+ corpora for machine translation.

3
Architectures for speech

Bibliography

[1] Mark Aronoff et al. *What is morphology?* John Wiley & Sons, 2022.

[2] Christine P Chai. “Comparison of text preprocessing methods”. In: *Natural language engineering* 29.3 (2023), pp. 509–553.

[3] Daniel Jurafsky et al. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*.

[4] Christo Kirov et al. “UniMorph 2.0: universal morphology”. In: *arXiv preprint arXiv:1810.11101* (2018).

[5] Adam Wiemerslage et al. “Morphological Processing of Low-Resource Languages: Where We Are and What’s Next”. In: *arXiv preprint arXiv:2203.08909* (2022).