EXPERT INSIGHT

In color

Building AI Agents with LLMs, RAG, and Knowledge Graphs

A practical guide to autonomous and modern AI agents

Salvatore Raieli | Gabriele Iuculano

packt

Building AI Agents with LLMs, RAG, and Knowledge Graphs

Portfolio Director: Gebin George

Relationship Lead: Ali Abdi

Project Manager: Prajakta Naik

Content Engineer: Mark D'Souza

Technical Editor: Irfa Ansari

Copy Editor: Safis Editing

Indexer: Tejal Soni

Production Designer: Alishon Falcon

Growth Lead: Kunal Sawant

First published: July 2025

Production reference: 1300625>

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK

ISBN 978-1-83508-706-0

www.packtpub.com

To Dorotea, Maria, Vincenzo, and Chiara, with love. A small thank you for the immense support.

– Salvatore Raieli

To Marta, for your strength when mine wavered, and for your light in difficult times. Thank you for walking with me through the storms. This book echoes the path we walked together.

– Gabriele Iuculano

The author acknowledges the use of cutting-edge AI, in this case, ChatGPT and Grammarly, with the sole aim of enhancing the language and clarity within the book, thereby ensuring a smooth reading experience for readers. It's important to note that the content itself has been crafted by the author and edited by a professional publishing team.

Contributors

About the authors

Salvatore Raieli is a senior data scientist in a pharmaceutical company with a focus on using AI for drug discovery against cancer. He has led different multidisciplinary projects with LLMs, agents, NLP, and other AI techniques. He has an MSc in AI and a PhD in immunology and has experience in building neural networks to solve complex problems with large datasets. He enjoys building AI applications for concrete challenges that can lead to societal benefits. In his spare time, he writes on his popularization blog on AI (on Medium).

Gabriele Iuculano boasts extensive expertise in embedded systems and AI. Leading a team as the test platform architect, Gabriele has been instrumental in architecting a sophisticated simulation system that underpins a cutting-edge test automation platform.

He is committed to integrating AI-driven solutions, focusing on predictive maintenance systems to anticipate needs and prevent downtimes. He obtained his MSc in AI from the University of Leeds, demonstrating expertise in leveraging AI for system efficiencies. Gabriele aims to revolutionize current business through the power of new disruptive technologies such as AI.

About the reviewers

Malhar Deshpande serves as the director and principal product owner of the AI Center of Excellence at Clean Harbors, where he leads AI initiatives, blending data science, machine learning, and generative AI to transform environmental services. With expertise in technology, innovation, and extensive experience in building AI teams, Malhar Deshpande is recognized for driving innovative solutions. He holds a Bachelor of Engineering, a master's in information systems, and an MBA from Northeastern University. As a technical reviewer, he is honored to contribute to this book, the AI and technology community, and the future of AI.

I am grateful to my parents, Mohan and Asha Deshpande, for their unwavering support and focus on education. Thanks to my wife, Shruti; my daughter, Tara; and my brother, Dr. Rupak, his wife, Dr. Riteeka, and their daughter, Samaira, for their love and encouragement throughout this journey.

Lalit Chourey is a seasoned software engineer with over a decade of experience in developing scalable backend services and distributed systems, specializing in AI infrastructure for LLM training. Currently a software engineer at Meta Platforms, Lalit leads a team in architecting robust systems for machine learning training. Previously at Microsoft, he led the development of several large-scale cloud services on Azure. Lalit holds a BTech in information technology from the National Institute of Technology, Bhopal, India.

Table of Contents

Part 1: The AI Agent Engine: From Text to Large Language Models

1

Analyzing Text Data with Deep Learning

Technical requirements

Representing text for AI

One-hot encoding

Bag-of-words

TF-IDF

Embedding, application, and representation

Word2vec

A notion of similarity for text

Properties of embeddings

RNNs, LSTMs, GRUs, and CNNs for text

RNNs

LSTMs

GRUs

CNNs for text

Performing sentiment analysis with embedding and deep learning

Summary

Part 2: AI Agents and Retrieval of Knowledge

4

Building a Web Scraping Agent with an LLM

Creating and Connecting a Knowledge Graph to an AI Agent

Further reading

Part 3: Creating Sophisticated AI to Solve Complex Scenarios

9

Creating Single- and Multi-Agent Systems

Preface

Building AI Agents with LLMs, RAG, and Knowledge Graphs introduces you to the evolving landscape of large language models (LLMs) and AI agents, offering both a theoretical foundation and practical guidance. It begins by explaining how text data can be represented and processed using deep learning, then progresses to modern architectures such as the Transformer model. From there, the book explores how LLMs are scaled and fine-tuned, and how their capabilities can be extended with tools, external memory systems, and agent-based frameworks. Technologies such as retrieval-augmented generation (RAG), GraphRAG, and multi-agent systems are explained in detail, with a focus on real-world applications and deployment. By the end of the book, you will have a clear understanding of how to build intelligent, tool-using AI agents and the role these systems play in shaping the future of AI.

Who this book is for

This book is intended for software engineers, data scientists, and researchers who want to understand and build applications using LLMs and AI agents. A basic understanding of Python programming and foundational concepts in machine learning is recommended to fully benefit from the content. While no deep expertise in NLP is required, familiarity with neural networks, REST APIs, and general software development practices will help you follow the examples and implement real-world systems. Whether you're looking to build intelligent agents, explore the inner workings of LLMs, or deploy AI applications at scale, this book provides both the theoretical background and practical guidance to get started.

What this book covers

Chapter 1, *Analyzing Text Data with Deep Learning*, introduces how to process and represent natural language in a format suitable for machine learning models. It covers various text encoding techniques, from basic one-hot encoding and bag of words to more advanced representations such as TF-IDF and word2vec. The chapter then explores key deep learning architectures for sequential data, such as RNNs, LSTMs, GRUs, and CNNs, and demonstrates how to apply them to text classification tasks. By the end of this chapter you will understand how these foundations enable modern language models such as ChatGPT.

Chapter 2, *The Transformer: The Model Behind the Modern AI Revolution*, introduces attention mechanisms and explains how they evolved into the transformer architecture. The chapter highlights the limitations of earlier models such as RNNs and LSTMs, and shows how transformers overcame them to become the foundation of modern NLP. Key topics include self-attention, masked language modeling, training techniques, and internal model visualization. The chapter concludes by demonstrating real-world applications and laying the groundwork for understanding today's LLMs.

Chapter 3, *Exploring LLMs as a Powerful AI Engine*, examines how the large-scale training of transformer models gave rise to today's LLMs. The chapter explores their evolution, capabilities, and limitations, including techniques such as instruction tuning, fine-tuning, and alignment. It also introduces more compact and efficient LLM variants, multimodal models that handle multiple data types, and understanding challenges such as hallucinations, ethical concerns, and prompt engineering.

Chapter 4, *Building a Web Scraping Agent with an LLM*, introduces the concept of AI agents as an extension of LLMs, aimed at overcoming their ability to perform actions. The chapter explores the key characteristics of agents, and distinctions between single and multi-agent systems. It also presents the main libraries used for building agents and guides you through the creation of a web-scraping agent capable of retrieving information from the internet.

Chapter 5, *Extending Your Agent with RAG to Prevent Hallucinations*, explores how RAG could overcome key limitations of LLMs, such as outdated knowledge and hallucinations. The chapter explains how RAG enables an LLM to access external information sources through embedding and vector databases, thereby improving accuracy and adaptability. It also compares RAG with fine-tuning and demonstrates its practical use by building a movie recommendation agent.

Chapter 6, *Advanced RAG Techniques for Information Retrieval and Augmentation*, expands on the basic RAG architecture by introducing enhancements at every stage of the pipeline—data ingestion, indexing, retrieval, and generation. The chapter explores modular RAG, techniques for scaling

systems with large datasets and user bases, and key concerns such as robustness and privacy. It also highlights current challenges and open questions surrounding the future development of RAG-based systems.

Chapter 7, *Creating and Connecting a Knowledge Graph to an AI Agent*, explores how to structure textual knowledge into knowledge graphs (KGs) to enhance information retrieval and reasoning in AI agents. The chapter introduces the concept of GraphRAG, where KGs are used to augment LLMs with structured contextual data. It covers how LLMs can be used to build KGs by extracting entities and relationships, how to use graphs for querying and reasoning, and discusses the benefits, limitations, and future directions of combining different approaches.

Chapter 8, *Reinforcement Learning and AI Agents*, explores how agents can learn by interacting with dynamic environments, adjusting their behavior based on experience. It introduces the fundamentals of reinforcement learning, explains how agents make decisions and improve over time, and demonstrates how neural networks can be used to guide behavior. The chapter concludes by discussing how LLMs can be combined with reinforcement learning to build more capable AI systems.

Chapter 9, *Creating Single- and Multi-Agent Systems*, explores how LLMs can be extended with tools and other models to form autonomous agents. It introduces the concept of single-agent and multi-agent systems, shows how LLMs can interact with APIs or external models, and presents key examples such as HuggingGPT. The chapter also covers agent coordination strategies, real-world applications in complex domains, and emerging business paradigms such as SaaS, MaaS, DaaS, and RaaS.

Chapter 10, *Building an AI Agent Application*, addresses the challenges of scaling and deploying AI agents in real-world applications. It introduces Streamlit as a rapid prototyping framework to create both frontend and backend components of an agent-based system. The chapter also covers key operational aspects such as asynchronous programming, containerization with Docker, and best practices for building scalable, production-ready AI solutions.

Chapter 11, *The Future Ahead*, explores the transformative potential of AI agents across industries such as healthcare and beyond. Building on the advancements discussed in earlier chapters, it reflects on the remaining technical and ethical challenges facing LLMs and agent systems. The chapter concludes by examining open questions and future directions in the development and deployment of intelligent AI agents.

To get the most out of this book

You should have a basic understanding of Python and be familiar with fundamental programming concepts such as functions, classes, and modules. A general knowledge of machine learning and neural networks (such as what a model is and how training works) will help in following the deeper technical content. While prior experience with deep learning frameworks or LLMs is not required, it will enhance your ability to apply the techniques discussed. The book is designed to be progressive, so concepts are introduced step by step, but a technical mindset is essential.

Software/hardware covered in the book	**Operating system requirements**
Python 3.10+	Windows, macOS, or Linux
PyTorch/Transformers	Windows, macOS, or Linux
Streamlit	Windows, macOS, or Linux
Docker	Windows, macOS, or Linux

For readers without access to a local GPU, using Google Colab is a convenient option. A Google Colab Pro account is recommended, as it provides access to more powerful GPUs such as NVIDIA T4 or A100, which can greatly improve performance when running embedding models, fine-tuning, or working with agents.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at https://github.com/PacktPublishing/Modern-AI-Agents/tree/main. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and X/Twitter handles. Here is an example: "The `process_frame` function is used to preprocess frames from the game to make them more suitable for training an RL agent."

A block of code is set as follows:

```
self.critic_linear = nn.Linear(512, 1)
self.actor_linear = nn.Linear(512, num_actions)
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

Any command-line input or output is written as follows:

```
streamlit run https://raw.githubusercontent.com/streamlit/my_apps/ master/my_app.py
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "Once we have our tokens ready, we can enter our question and click **Submit**."

> ***TIPS OR IMPORTANT NOTES***
>
> *Appear like this.*

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customercare@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share Your Thoughts

Once you've read *Building AI Agents with LLMs, RAG, and Knowledge Graphs*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

https://packt.link/free-ebook/978-1-83508-706-0

2. Submit your proof of purchase

3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1: The AI Agent Engine: From Text to Large Language Models

This part lays the foundation for understanding how modern AI agents process and generate language. It begins by exploring how raw text can be represented in numerical form suitable for deep learning models, introducing techniques such as word embeddings and basic neural architectures. The focus then shifts to the Transformer model and explains how attention mechanisms revolutionized natural language processing. Finally, it examines how large language models (LLMs) are built by scaling transformers, discussing training strategies, instruction tuning, fine-tuning, and the evolution toward models capable of general-purpose reasoning. Together, these chapters provide the technical and conceptual groundwork for building intelligent AI agents.

This part has the following chapters:

1
Analyzing Text Data with Deep Learning

Language is one of the most amazing abilities of human beings; it evolves during the individual's lifetime and is capable of conveying a message with complex meaning. Language in its natural form is not understandable to machines, and it is extremely challenging to develop an algorithm that can pick up the different nuances. Therefore, in this chapter, we will discuss how to represent text in a form that is digestible by machines.

In natural form, text cannot be directly fed to a **deep learning** model. In this chapter, we will discuss how text can be represented in a form that can be used by **machine learning** models. Starting with natural text, we will transform the text into numerical vectors that are increasingly sophisticated (one-hot encoding, **bag of words** (**BoW**), **term frequency-inverse document frequency** (**TF-IDF**)) until we create vectors of real numbers that represent the meaning of a word (or document) and allow us to conduct operations (word2vec). In this chapter, we introduce deep learning models, such as **recurrent neural networks** (**RNNs**), l**ong short-term memory** (**LSTM**), **gated recurrent units** (**GRUs**), and **convolutional neural network** (**CNNs**), to analyze sequences and discuss their strengths as well as the problems associated with them. Finally, we will assemble these models all together to conduct text classification, showing the power of the learned approaches.

By the end of this chapter, we will be able to take a corpus of text and use deep learning to analyze it. These are the bases that will help us understand how a **large language model** (**LLM**) (such as ChatGPT) works internally.

In this chapter, we'll be covering the following topics:

- Representing text for AI
- Embedding, application, and representation
- RNNs, LSTMs, GRUs, and CNNs for text
- Performing sentiment analysis with embedding and deep learning

Technical requirements

In this chapter, we will use standard libraries for Python. The necessary libraries can be found within each of the Jupyter notebooks that are in the GitHub repository for this chapter: https://github.com/PacktPublishing/Modern-AI-Agents/tree/main/chr1. The code can be executed on a CPU, but a GPU is advised.

Representing text for AI

Compared to other types of data (such as images or tables), it is much more challenging to represent text in a digestible representation for computers, especially because there is no unique relationship between the meaning of a word (signified) and the symbol that represents it (signifier). In fact, the meaning of a word changes from the context and the author's intentions in using it in a sentence. In addition, native text has to be transformed into a numerical representation to be ingested by an algorithm, which is not a trivial task. Nevertheless, several approaches were initially developed to be able to find a vector representation of a text. These vector representations have the advantage that they can then be used as input to a computer.

First, a collection of texts (**corpus**) should be divided into fundamental units (words). This process requires making certain decisions and process operations that collectively are called **text normalization**. A sentence, therefore, is divided into words by exploiting the natural division of spaces (**text segmentation**); each punctuation mark is also considered a single word. In fact, punctuation marks are considered to be the boundaries of sentences and convey important information (change of topic, questions, exclamations).

The second step is the definition of what a word is and whether some terms in the corpus should be directly joined under the same vocabulary instance. For example, "He" and "he" represent the same instance; the former is only capitalized. Since an algorithm does not include such nuances, one must normalize the text in lowercase. In some cases, we want to conduct more sophisticated normalizations such as **lemmatization** (joining words with the same root: "came" and "comes" are two forms of the verb) or **stemming** (stripping all suffixes of words).

Tokenization is the task of transforming a text into fundamental units. This is because, in addition to words, a text may also include percentages, numbers, websites, and other components. We will return to this later, but in the meantime, we will look at some simpler forms of tokenization.

One-hot encoding

In traditional **natural language processing** (**NLP**), text representation is conducted using discrete symbols. The simplest example is one-hot encoding. From a sequence of text in a corpus (consisting of n different words), we obtain an n-dimensional vector. In fact, the first step is to compute the set of different words present in the whole text corpus called vocabulary. For each word, we obtain a vector as long as the size of the vocabulary. Then for each word, we will have a long vector composed mainly of zeros and ones to represent the word (one-hot vectors). This system is mainly used when we want a

matrix of features and then train a model. This process is also called **vectorization**; here's a sparse vector for the following two words:

$$restaurant = [0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0]$$

$$pizzeria = [0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]$$

There are different problems associated with this representation. First, it captures only the presence (or the absence) of a word in a document. Thus, we are losing all the semantic relationships between the words. Second, an average language has about 200,000 words, so for each word, we would have a vector of length 200,000. This leads to very sparse and high-dimensional vectors. For large corpora, we need high memory to store the vectors and high computational capacity to handle them. In addition, there is no notion of similarity. The two words in the preceding example are two places that sell food, and we would like the vectors representing these words to encode this similarity. If the vectors had a notion of similarity, we could conduct clustering, and the synonyms would be in the same cluster.

In order to obtain such a matrix, we must do the following:

- Standardize the text before tokenization. In this case, we simply transform everything into lowercase.
- We construct a vocabulary constituted of unique words and save the vocabulary so that in a case from a vector, we can get the corresponding word.
- We create an array and then populate it with `1` at the index of the word in the vocabulary; `0`s elsewhere.

Let's take a look at how this works in code:

```
import numpy as np
def one_hot_encoding(sentence):
    words = sentence.lower().split()
    vocabulary = sorted(set(words))
    word_to_index = {word: i for i,
        word in enumerate(vocabulary)}
    one_hot_matrix = np.zeros((
        len(words), len(vocabulary)), dtype=int)
    for i, word in enumerate(words):
        one_hot_matrix[i, word_to_index[word]] = 1
    return one_hot_matrix, vocabulary
```

Let's look at a specific example:

```
sentence = "Should we go to a pizzeria or do you prefer a restaurant?"
one_hot_matrix, vocabulary = one_hot_encoding(sentence)
print("Vocabulary:", vocabulary)
print("One-Hot Encoding Matrix:\n", one_hot_matrix)
```

We get the following output:

```
Vocabulary: ['a', 'do', 'go', 'or', 'pizzeria', 'prefer', 'restaurant?', 'should', 't
o', 'we', 'you']
One-Hot Encoding Matrix:
 [[0 0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 0]
 [1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0 0]]
```

> ***IMPORTANT NOTE***
>
> *Observe how choosing another sentence will result in a different matrix and how, by increasing the length of the sentence, the matrix grows proportionally to the number of different words. Also, note that for repeated words, we get equal vectors. Check the preceding output.*

Even if it is a simple method, we have obtained a first representation of text in a vectorial form.

Bag-of-words

In the previous section, we discussed one-hot encoding and some of the problems associated with this form of text representation. In the previous example, we worked with a single sentence, but a corpus is made up of thousands if not millions of documents; each of these documents contains several words with a different frequency. We want a system that preserves this frequency information, as it is important for the classification of text. In fact, documents that have similar content are similar, and their meaning will also be similar.

BoW is an algorithm for extracting features from text that preserves this frequency property. BoW is a very simple algorithm that ignores the position of words in the text and only considers this frequency property. The name "bag" comes precisely from the fact that any information concerning sentence order and structure is not preserved by the algorithm. For BoW, we only need a vocabulary and a way to be able to count words. In this case, the idea is to create document vectors: a single vector represents a document and the frequency of words contained in the vocabulary. *Figure 1.1* visualizes this concept with a few lines from *Hamlet*:

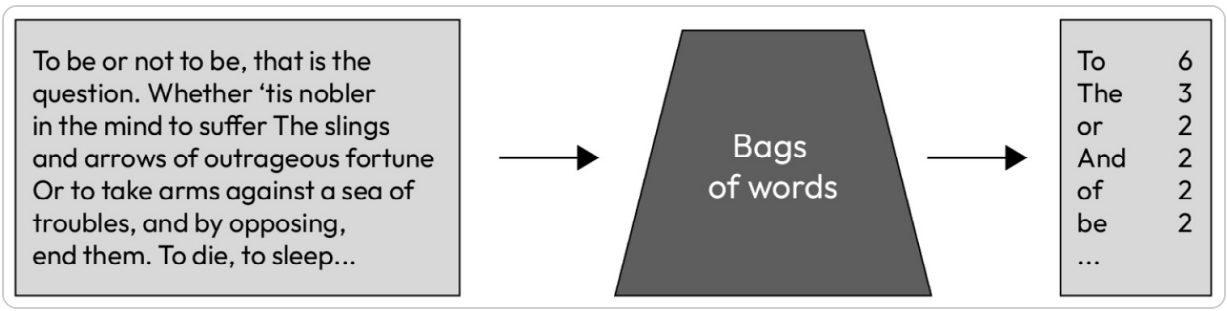

Figure 1.1 – Representation of the BoW algorithm

Even this representation is not without problems. Again, as the vocabulary grows, so will the size of the vectors (the size of each vector is equal to the length of the vocabulary). In addition, these vectors tend to be scattered, especially when the documents are very different from each other. High-dimensional or sparse vectors are not only problematic for memory and computational costs but for algorithms as well (the longer the vectors, the more weight you need in the algorithm, leading to a risk of overfitting). This is called the **curse of dimensionality**; the greater the number of features, the less meaningful the distances between examples. For large corpora, some solutions have been proposed, such as ignoring punctuation, correcting misspelled words, stemming algorithms, or ignoring words with high frequency that don't add information (articles, prepositions, and so on).

In order to get a BoW matrix for a list of documents, we need to do the following:

- Tokenize each document to get a list of words.
- Create our vocabulary of unique words and map each word to the corresponding index in the vocabulary.
- Create a matrix where each row represents a document and each column, instead, a word in the vocabulary (the documents are the examples, and the words are the associated features).

Let's look at the code again:

```
import numpy as np
def bag_of_words(sentences):
    """
    Creates a bag-of-words representation of a list of documents.
    """
    tokenized_sentences = [
        sentence.lower().split() for sentence in sentences
    ]
    flat_words = [
        word for sublist in tokenized_sentences for word in sublist
    ]
    vocabulary = sorted(set(flat_words))
    word_to_index = {word: i for i, word in enumerate(vocabulary)}
    bow_matrix = np.zeros((
        len(sentences), len(vocabulary)), dtype=int)
    for i, sentence in enumerate(tokenized_sentences):
        for word in sentence:
            if word in word_to_index:
                bow_matrix[i, word_to_index[word]] += 1
    return vocabulary, bow_matrix
```

Here's an example:

```
corpus = ["This movie is awesome awesome",
          "I do not say is good, but neither awesome",
          "Awesome? Only a fool can say that"]
vocabulary, bow_matrix = bag_of_words(corpus)
print("Vocabulary:", vocabulary)
print("Bag of Words Matrix:\n", bow_matrix)
```

This prints the following output:

```
Vocabulary: ['a', 'awesome', 'awesome?', 'but', 'can', 'do', 'fool', 'good,', 'i', 'i
s', 'movie', 'neither', 'not', 'only', 'say', 'that', 'this']
Bag of Words Matrix:
 [[0 2 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1]
 [0 1 0 1 0 1 0 1 1 1 0 1 1 0 1 0 0]
 [1 0 1 0 1 0 1 0 0 0 0 0 0 1 1 1 0]]
```

> ***IMPORTANT NOTE***
>
> *Note how in the example, the word "**`awesome`**" is associated with a review with a positive, neutral, or negative meaning. Without context, the frequency of the word "awesome" alone does not tell us the sentiment of the review.*

Here, we have learned how to transform text in a vectorial form while keeping the notion of frequency for each word.

TF-IDF

In the previous section, we obtained a document-term matrix. However, the raw frequency is very skewed and does not always allow us to discriminate between two documents. The document-term matrix was born in information retrieval to find documents, though words such as "good" or "bad" are not very discriminative since they are often used in text with a generic meaning. In contrast, words with low frequency are much more informative, so we are interested more in relative than absolute frequency:

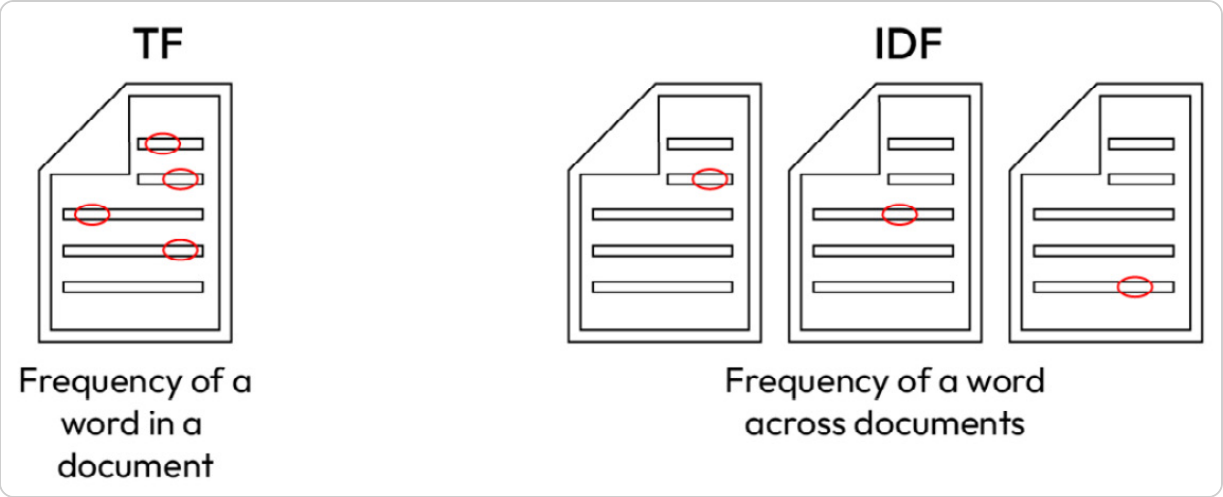

Figure 1.2 – Intuition of the components of TF-IDF

Instead of using raw frequency, we can use the logarithm in base 10, because a word that occurs 100 times in a document is not 100 times more relevant to its meaning in the document. Of course, since vectors can be very sparse, we assign 0 if the frequency is 0. Second, we want to pay more attention to words that are present only in some documents. These words will be more relevant to the meaning of the document, and we want to preserve this information. To do this, we normalize by IDF. IDF is defined as the ratio of the total number of documents in the corpus to how many documents a term is present in. To summarize, to obtain the TF-IDF, we multiply TF by the logarithm of IDF.

This is demonstrated in the following code block:

```
import numpy as np
def compute_tf(sentences):
    """Compute the term frequency matrix for a list of sentences."""
    vocabulary = sorted(set(
        word for sentence in sentences
        for word in sentence.lower().split()))
    word_index = {word: i for i, word in enumerate(vocabulary)}
    tf = np.zeros((
        len(sentences), len(vocabulary)), dtype=np.float32)
    for i, sentence in enumerate(sentences):
        words = sentence.lower().split()
        word_count = len(words)
        for word in words:
            if word in word_index:
                tf[i, word_index[word]] += 1 / word_count
    return tf, vocabulary
def compute_idf(sentences, vocabulary):
    """Compute the inverse document frequency for a list of sentences."""
    num_documents = len(sentences)
    idf = np.zeros(len(vocabulary), dtype=np.float32)
    word_index = {word: i for i, word in enumerate(vocabulary)}
    for word in vocabulary:
        df = sum(
            1 for sentence in sentences
            if word in sentence.lower().split()
        )
```

```
            idf[word_index[word]] = np.log(
                num_documents / (1 + df)) + 1  # Smoothing
        return idf
    def tf_idf(sentences):
        """Generate a TF-IDF matrix for a list of sentences."""
        tf, vocabulary = compute_tf(sentences)
        idf = compute_idf(sentences, vocabulary)
        tf_idf_matrix = tf * idf
        return vocabulary, tf_idf_matrix
    vocabulary, tf_idf_matrix = tf_idf(corpus)
    print("Vocabulary:", vocabulary)
    print("TF-IDF Matrix:\n", tf_idf_matrix)
```

This generates the following output:

```
Vocabulary: ['a', 'awesome', 'awesome?', 'but', 'can', 'do', 'fool', 'good,', 'i', 'i
s', 'movie', 'neither', 'not', 'only', 'say', 'that', 'this']
TF-IDF Matrix:
 [[0.         0.4        0.         0.         0.         0.
  0.         0.         0.         0.2        0.28109303 0.
  0.         0.         0.         0.         0.28109303]
 [0.         0.11111111 0.         0.1561628  0.         0.1561628
  0.         0.1561628  0.1561628  0.11111111 0.         0.1561628
  0.1561628  0.         0.11111111 0.         0.        ]
 [0.20078073 0.         0.20078073 0.         0.20078073 0.
  0.20078073 0.         0.         0.         0.         0.
  0.         0.20078073 0.14285715 0.20078073 0.        ]]
```

> ***IMPORTANT NOTE***
>
> *In this example, we used the same corpus as in the previous section. Note how word frequencies changed after this normalization.*

In this section, we learned how we can normalize text to decrease the impact of the most frequent words and give relevance to words that are specific to a subset of documents. Next, we'll discuss embedding.

Embedding, application, and representation

In the previous section, we discussed how to use vectors to represent text. These vectors are digestible for a computer, but they still suffer from some problems (sparsity, high dimensionality, etc.). According to the distributional hypothesis, words with a similar meaning frequently appear close together (or words that appear often in the same context have the same meaning). Similarly, a word can have a different meaning depending on its context: "I went to deposit money in the *bank*" or "We went to do a picnic on the river *bank*." In the following diagram, we have a high-level representation of the embedding process. So, we want a process that allows us to start from text to obtain an array of vectors, where each vector corresponds to the representation of a word. In this case, we want a model

that will then allow us to map each word to a vector representation. In the next section, we will describe the process in detail and discuss the theory behind it.

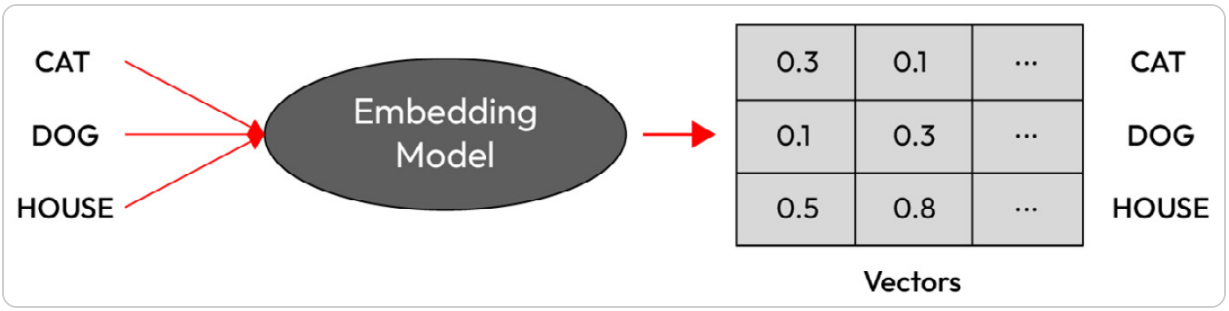

Figure 1.3 – High-level representation of the embedding process

We would, therefore, like to generate vectors that are small in size, composed of real (dense) numbers, and that preserve this contextual information. Thus, the purpose is to have vectors of limited size that can represent the meaning of a word. The scattered vectors we obtained earlier cannot be used efficiently for mathematical operations or downstream tasks. Also, the more words there are in the vocabulary, the larger the size of the vectors we get. Therefore, we want dense vectors (with real numbers) that are small in size and whose size does not increase as the number of words in the vocabulary increases.

In addition, these vectors have a distributed representation of the meaning of the word (whereas in sparse vectors, it was local or where the `1` was located). As we will see a little later, these dense vectors can be used for different operations because they better represent the concept of similarity between words. These dense vectors are called **word embeddings**.

This concept was introduced in 2013 by Mikolov with a framework called **word2vec**, which will be described in detail in the next section.

Word2vec

The intuition behind word2vec is simple: predict a word *w* from its context. To do this, we need a **neural network** and a large corpus. The revolutionary idea is that by training this neural network to predict which words *c* are needed near the target word *w*, the weights of the neural network will be the embedding vectors. This model is self-supervised; the labels in this case are implicit, and we do not provide them.

Word2vec simplifies this idea by making the system extremely fast and effective in two ways: by turning the task into binary classification (Is the word *c* needed in the context of the word *w*? Yes or no?) and using a logistic regression classifier:

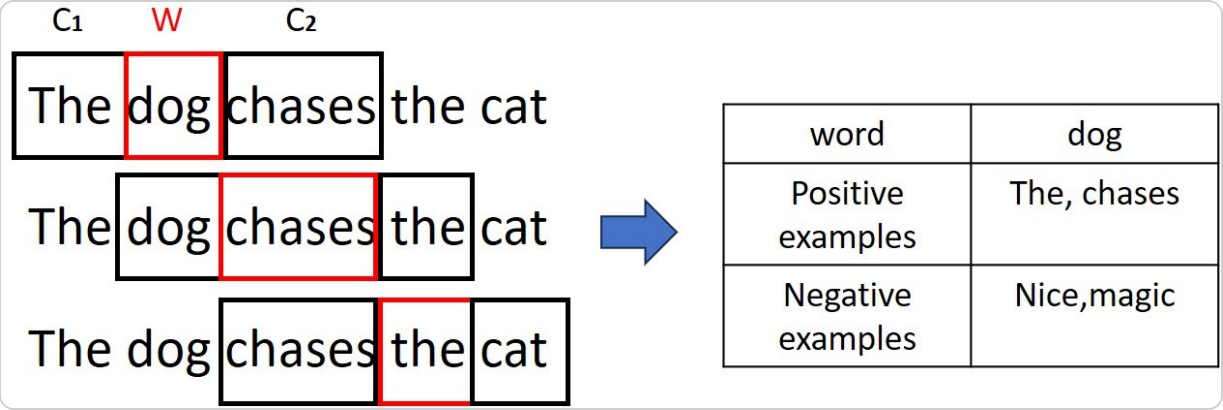

Figure 1.4 – In word2vec, we slide a context window (here represented as a three-word context window), and then we randomly sample some negative words

Given a text *t*, we scroll a window *c* (our context) for a word *w* in the center of our window; the words around it are examples of the positive class. After that, we select other random words as negative examples. Finally, we train a model to classify the positive and negative examples; the weights of the model are our embeddings.

Given a word *w* and a word *c*, we want the probability that the word *c* is in the context of *w* to be similar to its embedding similarity. In other words, if the vector representing *w* and *c* are similar, *c* must often be in the context of *w* (word2vec is based on the notion of context similarity). We define this embedding similarity by the dot product between the two embedding vectors for *w* and *c* (we use the sigmoid function to transform this dot product into a probability and thus allow comparison). So, the probability that *c* is in the context of *w* is equal to the probability that their embeddings are similar:

$$P(+|w,c) = \sigma(\boldsymbol{c} \bullet \boldsymbol{w}) = \frac{1}{1 + exp(-\boldsymbol{c} \bullet \boldsymbol{w})}$$

This is done for all words in context *L*. To simplify, we assume that all words in the context window are independent, so we can multiply the probabilities of the various words *c*. Similarly, we want to ensure that this dot product is minimal for words that are not in the context of word *w*. So, on the one hand, we maximize the probability for words in the context, and on the other hand, we minimize the probability for words that are not in the context. In fact, words that are not in the context of *w* are randomly extracted during training, and the process is the same:

$$P(+|w,c) = \prod_{i=1}^{L} \frac{1}{1 + exp(-\boldsymbol{c_i} \bullet \boldsymbol{w})}$$

$$P(-|w,c) = \prod_{i=1}^{L} \frac{1}{1 + exp(\boldsymbol{c_i} \bullet \boldsymbol{w})}$$

For simplicity, we take the logarithm of probability:

$$logP(+|w,c) = \sum_{i=1}^{L} \log\left(\frac{1}{1 + exp(-\boldsymbol{c_i} \bullet \boldsymbol{w})}\right)$$

$$logP(-|w,c) = \sum_{i=1}^{L} \log\left(\frac{1}{1 + exp(\boldsymbol{c_i} \bullet \boldsymbol{w})}\right)$$

The matrix of weights *w* is our embedding; it is what we will use from now on. Actually, the model learns two matrices of vectors (one for *w* and one for *c*), but the two matrices are very similar, so we take just one. We then use cross-entropy to train the models and learn the weights for each vector:

LCE=-logP+w,cpos+∑i=1LlogP-w,cneg

This is represented visually in the following diagram:

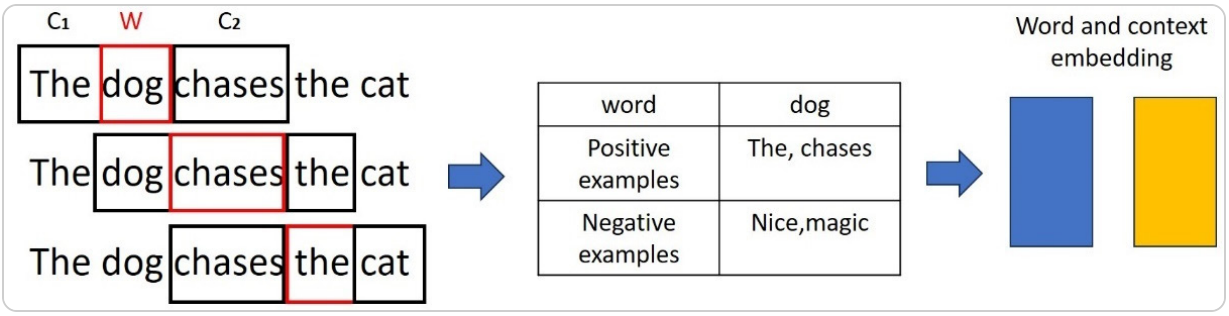

Figure 1.5 – Word and context embedding

The following choices affect the quality of embedding:

- *Data quality is critical*. For example, leveraging Wikipedia allows better embedding for semantic tasks, while using news improves performance for syntactic tasks (a mixture of the two is recommended). Using Twitter or other social networks can insert bias.
- At the same time, *a larger amount of text improves embedding performance*. A large amount of text can partially compensate for poor quality but at the cost of much longer training (for example, Common Crawl is a huge dataset downloaded from the internet that is pretty dirty, though).
- *The number of dimensions is another important factor*. The larger the size of the embedding, the better its performance. 300 is considered a sweet spot because, beyond this size, number performance does not increase significantly.
- *The size of the context window also has an impact*. Generally, a context window of 4 is used, but a context window of 2 allows for vectors that better identify parts of speech. In contrast, long context windows are more useful if we are interested in similarity more broadly.

In Python, we can easily get an embedding from lists of tokens using the following code:

```
from gensim.models import Word2Vec
```

```
model = Word2Vec(sentences=list_of_tokens,
                 sg=1,
                 vector_size=100,
                 window=5,
                 workers=4)
```

> ***IMPORTANT NOTE***
>
> *The complete code is present in the GitHub repository. We used an embedding of 100 dimensions and a window of 5 words.*

Once we have our embedding, we can visualize it. For example, if we try **clustering** the vectors of some words, words that have similar meanings should be closer together:

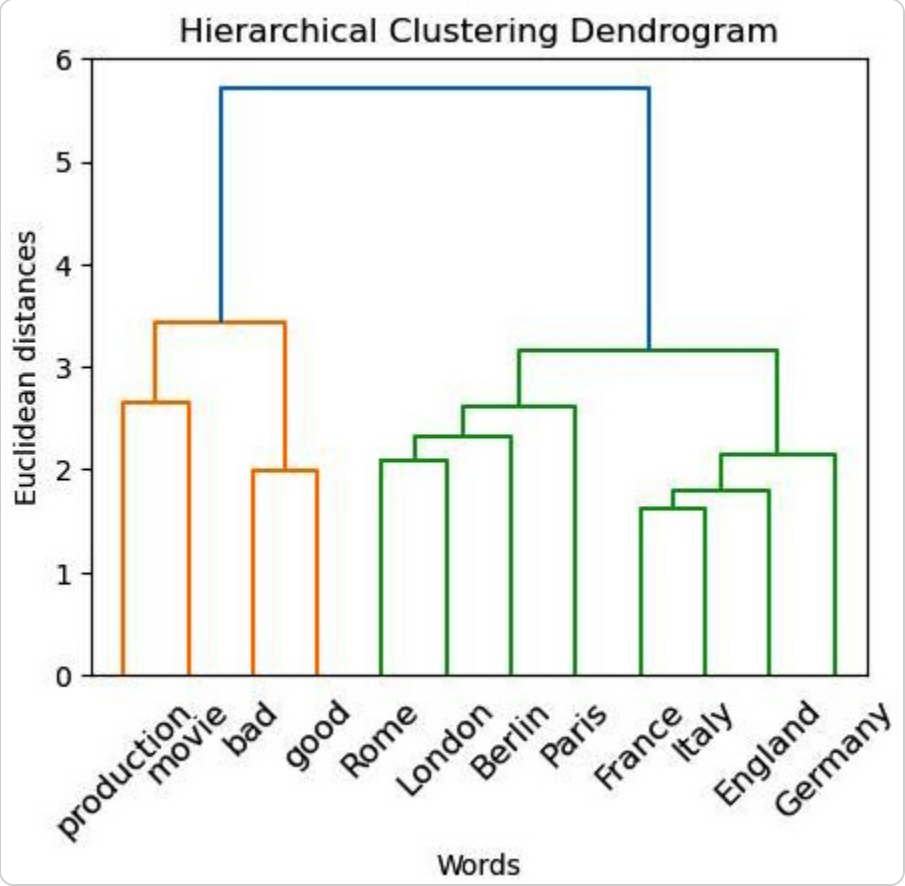

Figure 1.6 – Clustering of some of the vectors obtained from the embedding

Another way to visualize vectors is to use dimensionality reduction techniques. Vectors are multidimensional (100-1,024), so it is more convenient to reduce them to two or three dimensions so that they can be visualized more easily. Some of the most commonly used techniques are **principal component analysis** (**PCA**) and **t-distributed stochastic neighbor embedding** (**t-SNE**). **Uniform Manifold Approximation and Projection** (**UMAP**), on the other hand, is a technique that has become the first choice for visualizing multidimensional data in recent years:

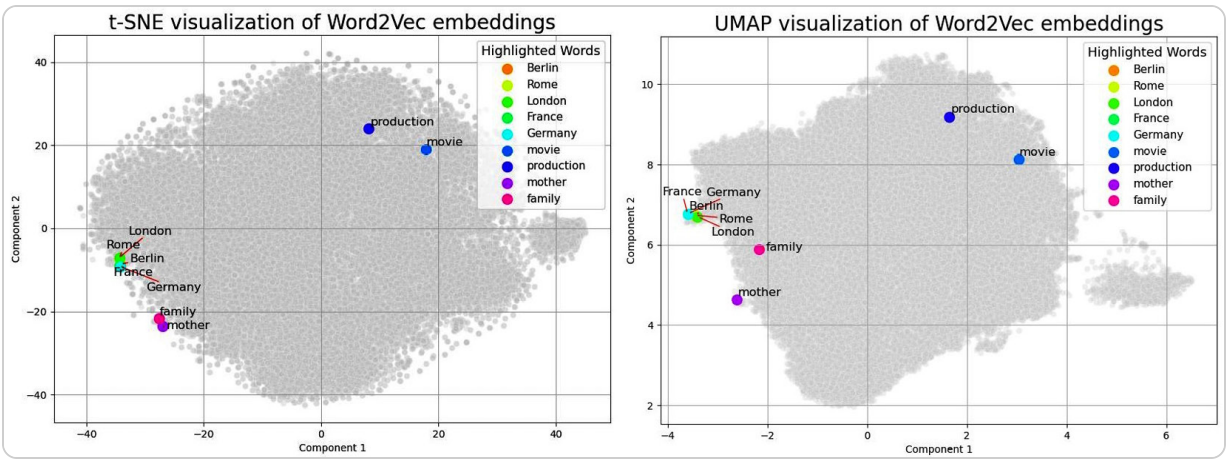

Figure 1.7 – 2D projection of word2vec embedding highlighting some examples

UMAP has emerged because it produces visualizations that better preserve semantic meaning and relationships between examples and also better represent local and global structures. This makes for better clusters, and UMAP can also be used in preprocessing steps before a classification task on vectors.

A notion of similarity for text

Once we have obtained vector representations, we need a method to calculate the similarity between them. This is crucial in many applications—for instance, to find words in an embedding space that are most similar to a given word, we compute the similarity between its vector and those of other words. Similarly, given a query sentence, we can retrieve the most relevant documents by comparing its vector with document embeddings and selecting those with the highest similarity.

Most similarity measures are based on the **dot product**. This is because the dot product is high when the two vectors have values in the same dimension. In contrast, vectors that have zero alternately will have a dot product of zero, thus orthogonal or dissimilar. This is why the dot product was used as a similarity measure for word co-occurrence matrices or with vectors derived from document TF matrices:

$$\boldsymbol{dotproduct}: \boldsymbol{a} \bullet \boldsymbol{b} = \sum_{\boldsymbol{i}=1}^{N} a_i \times b_i = a_1 \times b_1 + a_2 \times b_2 + \cdots a_n \times b_n$$

$$magnitude = |\boldsymbol{a}| = \sqrt[2]{\sum_{\boldsymbol{i}=1}^{N} a_i^2}$$

The dot product has several problems, though:

- It tends to favor vectors with long dimensions
- It favors vectors with high values (which, in general, are those of very frequent and, therefore, useless words)
- The value of the dot product has no limits

Therefore, alternatives have been sought, such as a normalized version of the dot product. The normalized dot product is equivalent to the cosine of the angle between the two vectors, hence **cosine similarity**:

$$cos\Theta = \frac{\boldsymbol{a} \bullet \boldsymbol{b}}{|\boldsymbol{a}||\boldsymbol{b}|} = \frac{\sum_{i=1}^{N} a_i \times b_i}{\sqrt[2]{\sum_{i=1}^{N} a_i^2}\sqrt[2]{\sum_{i=1}^{N} b_i^2}}$$

Cosine similarity has some interesting properties:

- It is between -1 and 1. Opposite or totally dissimilar vectors will have a value of -1, 0 for orthogonal vectors (or totally dissimilar for scattered vectors), and 1 for perfectly similar vectors. Since it measures the angle between two vectors, the interpretation is easier and is within a specific range, so it allows one intuitively to understand the similarity or dissimilarity.
- It is fast and cheap to compute.
- It is less sensitive to word frequency and, thus, more robust to outliers.
- It is scale-invariant, meaning that it is not influenced by the magnitude of the vectors.
- Being normalized, it can also be used with high-dimensional data.

For 2D vectors, we can plot to observe these properties:

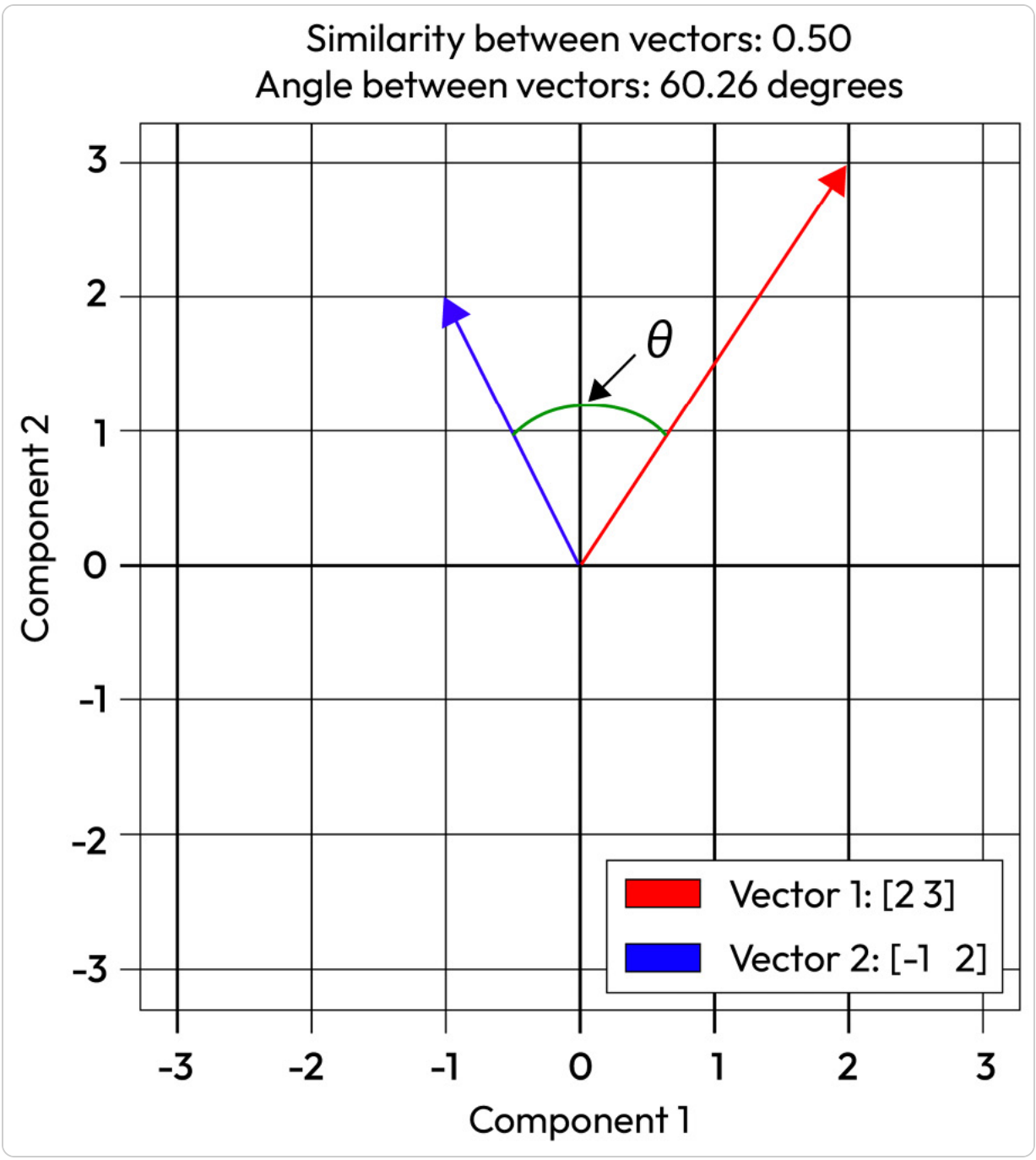

Figure 1.8 – Example of cosine similarity between two vectors

In the next section, we can define the properties of our trained embedding using this notion of similarity.

Properties of embeddings

Embeddings are a surprisingly flexible method and manage to encode different syntactic and semantic properties that can both be visualized and exploited for different operations. Once we have a notion of similarity, we can search for the words that are most similar to a word *w*. Note that similarity is defined as appearing in the same context window; the model cannot differentiate synonyms and antonyms.

In addition, the model is also capable of representing grammatical relations such as superlatives or verb forms.

Another interesting relationship we can study is analogies. The parallelogram model is a system for representing analogies in a cognitive space. The classic example is *king:queen::man:?* (which in a formula would be *a:b::a*:?*). Given that we have vectors, we can turn this into an *a-a*+b* operation.

We can test this in Python using the embedding model we have trained:

- We can check the most similar words
- We can test the analogy
- We can then test the capacity to identify synonyms and antonyms

The code for this process is as follows:

```
word_1 = "good"
syn = "great"
ant = "bad"
most_sim =model.wv.most_similar("good")
print("Top 3 most similar words to {} are :{}".format(
    word_1, most_sim[:3]))
synonyms_dist = model.wv.distance(word_1, syn)
antonyms_dist = model.wv.distance(word_1, ant)
print("Synonyms {}, {} have cosine distance: {}".format(
    word_1, syn, synonyms_dist))
print("Antonyms {}, {} have cosine distance: {}".format(
    word_1, ant, antonyms_dist))
a = 'king'
a_star = 'man'
b = 'woman'
b_star= model.wv.most_similar(positive=[a, b], negative=[a_star])
print("{} is to {} as {} is to: {} ".format(
    a, a_star, b, b_star[0][0]))
```

> ***IMPORTANT NOTE***
>
> *This is done with the embedding we have trained before. Notice how the model is not handling antonyms well.*

The method is not exactly perfect, so sometimes the right answer is not the first result, but it could be among the first three inputs. Also, this system works for entities that are frequent within the text (city names, common words) but much less with rarer entities.

Embeddings can also be used as a tool to study how the meaning of a word changes over time, especially if you have text corpora that span several decades. This is demonstrated in the following diagram, which shows how the meanings of the words *gay*, *broadcast*, and *awful* have changed:

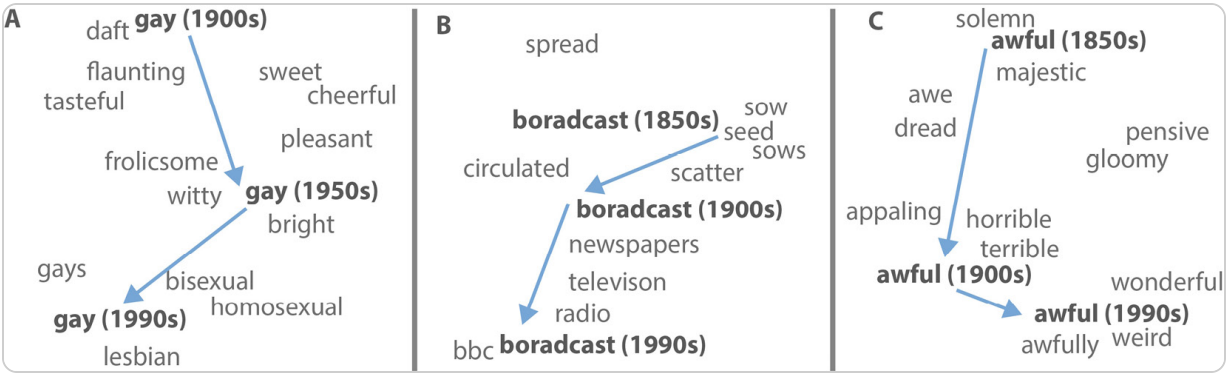

Figure 1.9 – 2D visualization of how the semantic meaning of a word changes over the years; these projections are obtained using text from different decades and embeddings (https://arxiv.org/abs/1605.09096)

Finally, a word can still have multiple meanings! For example, common words such as "good" have more than one meaning depending on the context. One may wonder whether a vector for a word in an embedding represents only one meaning or whether it represents the set of meanings of a word. Fortunately, embedding vectors represent a weighted sum of the various meanings of a word (linear superposition). The weights of each meaning are proportional to the frequency of that meaning in the text. Although these meanings reside in the same vector, when we add or subtract during the calculation of analogies, we are working with these components. For example, "apple" is both a fruit and the name of a company; if we conduct the operation *apple:red::banana:?*, we are subtracting only a very specific semantic component from the apple vector (the component that is similar to red). This flexibility can be useful when we want to disambiguate meanings. Also, since the vector space is sparse, by exploiting sparse coding, we can separate the various meanings:

tie					spring				
trousers	season	scoreline	wires	operatic	beginning	dampers	flower	creek	humid
blouse	teams	goalless	cables	soprano	until	brakes	flowers	brook	winters
waistcoat	winning	equaliser	wiring	mezzo	months	suspension	flowering	river	summers
skirt	league	clinching	electrical	contralto	earlier	absorbers	fragrant	fork	ppen
sleeved	finished	scoreless	wire	baritone	year	wheels	lilies	piney	warm
pants	championship	replay	cable	coloratura	last	damper	flowered	elk	temperatures

Figure 1.10 – Table showing how a vector in word2vec is encoding for different meanings at the same time (https://aclanthology.org/Q18-1034/)

These vectors are now providing contextual and semantic meaning for each word in the text. We can use this rich source of information for tasks such as text classification. What we need now are models that can handle the sequential nature of text, which we will learn about in the next section.

RNNs, LSTMs, GRUs, and CNNs for text

So far, we have discussed how to represent text in a way that is digestible for the model; in this section, we will discuss how to analyze the text once a representation has been obtained. Traditionally, once we obtained a representation of the text, it was fed to models such as naïve Bayes or even algorithms such as logistic regression. The success of neural networks has made these machine learning algorithms outdated. In this section, we will discuss deep learning models that can be used for various tasks.

RNNs

The problem with classical neural networks is that they have no memory. This is especially problematic for time series and text inputs. In a sequence of words *t*, the word *w* at time *t* depends on the *w* at time *t-1*. In fact, in a sentence, the last word is often dependent on several words in the sentence. Therefore, we want an NN model that maintains a memory of previous inputs. An **RNN** maintains an internal state that maintains this memory; that is, it stores information about previous inputs, and the outputs it produces are affected by previous inputs. These networks perform the same operation on all elements of the sequence (hence recurrent) and maintain the memory of this operation:

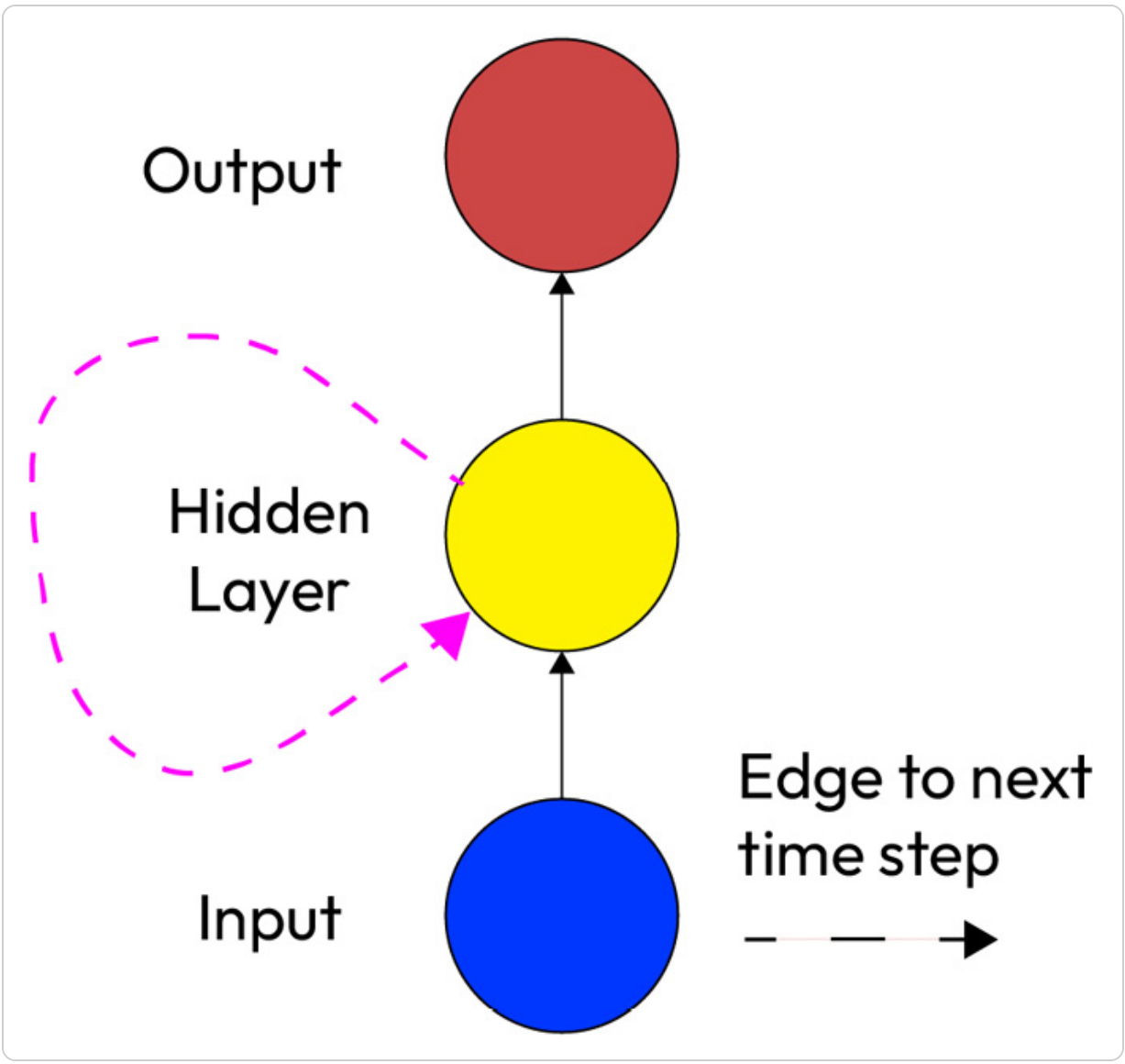

Figure 1.11 – Simple example of an RNN (https://arxiv.org/pdf/1506.00019)

A classical neural network (**feedforward neural network**) considers inputs to be independent, and one layer of a neural network performs the following operation for a vector representing the element at time *t*:

$$y(t)=\sigma(Wxt+b)$$

However, in a simple RNN, the following operations are conducted:

$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{U}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}\boldsymbol{x}^{t}$$

$$ht=\tanh(at)$$

$$ot=c+Vht$$

$$y_t = \sigma(o_t)$$

These operations may seem complicated, but in fact, we are simply maintaining a hidden state that considers the previous iterations. The first equation is a normal feedforward layer modified, in which we multiply the previously hidden state *h* by a set of weights *U*. This matrix *U* allows us to control how the neural network uses the previous context to bind input and past inputs (how the past influences the output for input at time *t*). In the second equation, we create a new hidden state that will then be used for subsequent computations but also for the next input. In the third equation, we are creating the output; we use a bias vector and a matrix to compute the output. In the last equation, it is simply passed as a nonlinearity function.

These RNNs can be seen as unrolled entities throughout time, in which we can represent the network and its computations throughout the sequence:

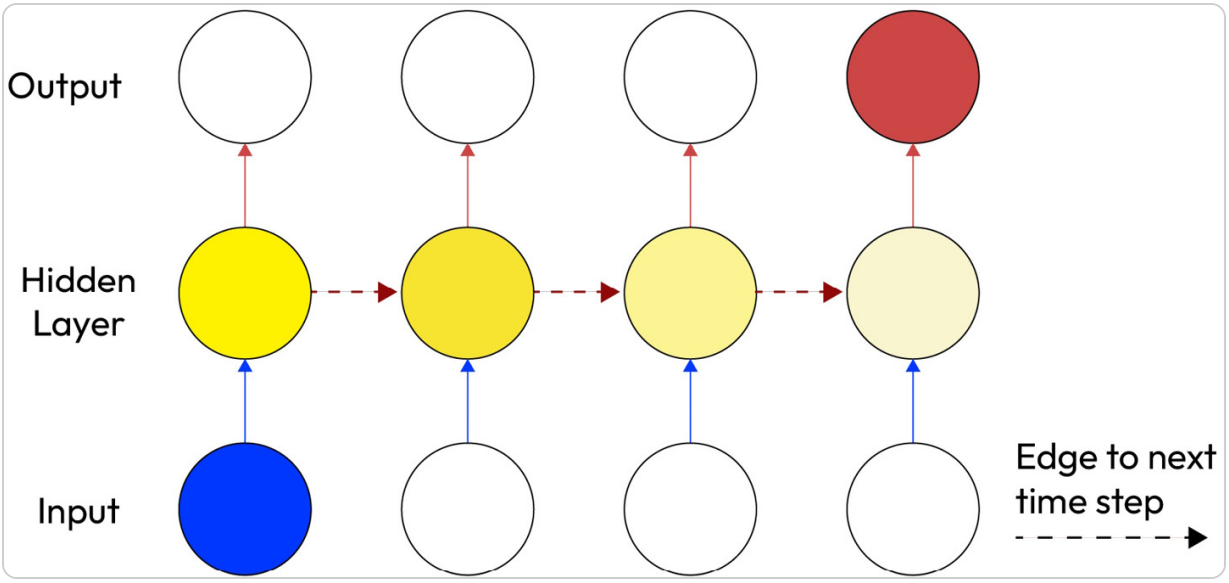

Figure 1.12 – Simple example of an RNN unrolled through the sequence (https://arxiv.org/pdf/1506.00019)

We can test in Python with a PyTorch RNN to see how it is transforming the data:

```
array = np.random.random((10, 5, 3))
# Convert the numpy array to a PyTorch tensor
data_tensor = torch.tensor(array, dtype=torch.float32)
RNN = nn.RNN(input_size=3, hidden_size=10,
             num_layers=1, batch_first=True)
output, hidden = RNN(data_tensor)
output.shape
```

> ***IMPORTANT NOTE***

> *Notice how the model is transforming the data; we can also access the hidden state.*

We can see several interesting things:

- The RNN is not limited by the size of the input; it is a cyclic operation that is conducted over the entire sequence. RNNs basically process one word at a time. This cyclicality also means backpropagation is conducted for each time step. Although this model works well for series analysis, its sequential nature does not allow it to be parallelized.
- Theoretically, this model could be trained with infinite sequences of words; theoretically, after a few time steps, it begins to forget the initial inputs.
- Training can become inefficient due to the vanishing gradient problem, where gradients must propagate from the final cell back to the initial one. During this process, they can shrink exponentially and approach zero, making it difficult for the model to learn long-range dependencies. Conversely, the exploding gradient problem can also occur, where gradients grow uncontrollably large, leading to unstable training.

RNNs are not the only form of deep learning models that are relevant to this topic.

LSTMs

In theory, RNNs should be able to process long sequences and remember the initial input. However, in reality, the information inside the hidden state is local rather than global, and for a time t, it considers only the previous time steps and not the entire sequence. The main problem with such a simple model is that the hidden state must simultaneously fulfill two roles: provide information relevant to the output at time t and store memory for future decisions.

An **LSTM** is an extension of RNNs, designed with the idea that the model can forget information that is not important and keep only the important context.

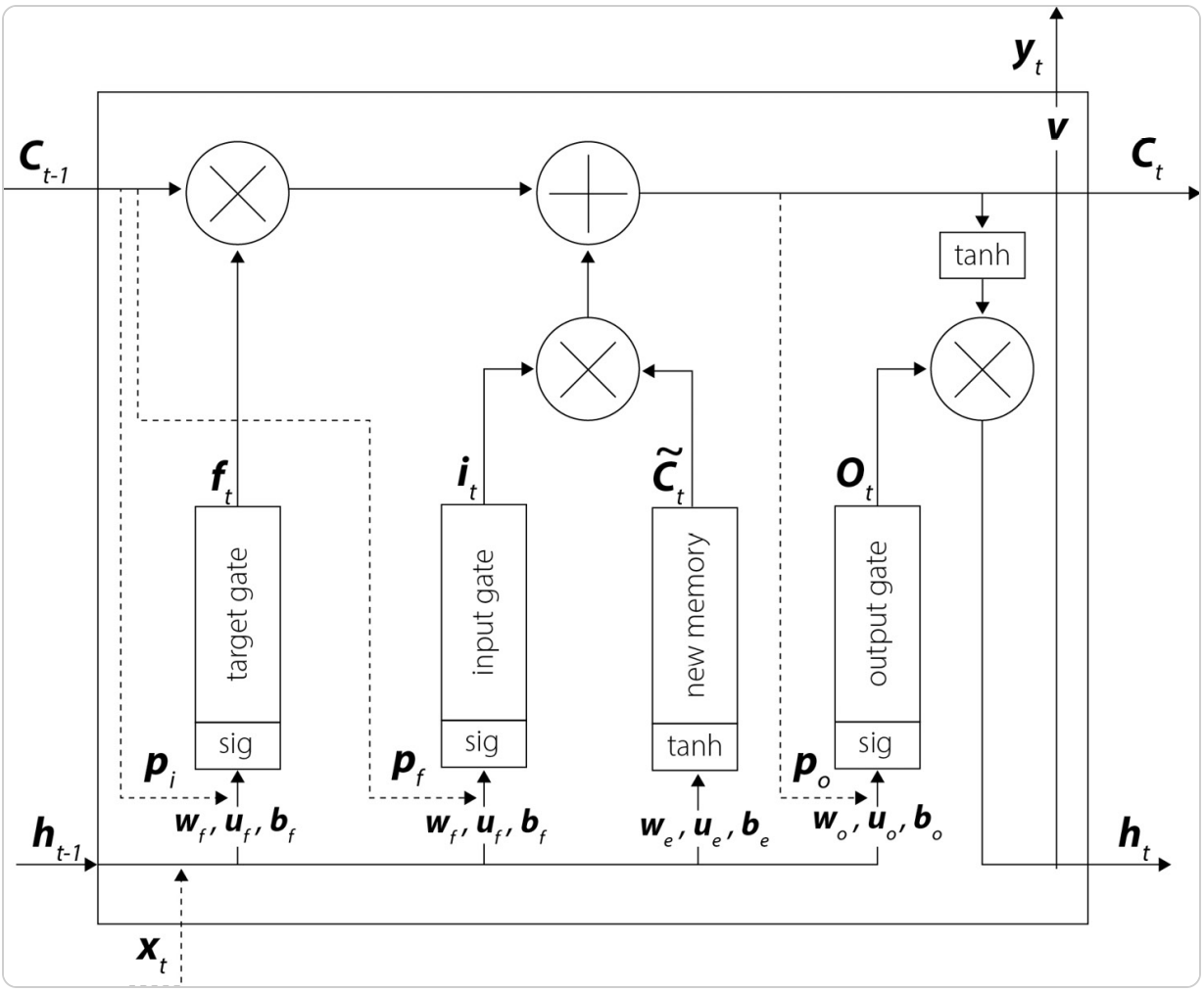

Figure 1.13 – Internal structure of an LSTM cell (https://arxiv.org/pdf/2304.11461)

An LSTM has internal mechanisms to control the information (gates) within the layer; additionally, it has a dedicated context layer. So, we have two hidden states in which the first, *h*, serves for information at time *t* (short memory) and the other, *c*, for information at long term. The gates can be open (1) or closed (0); this is achieved by a feed-forward layer with sigmoid activation to squeeze values between zero and one. After that, we use the **Hadamard product** (or pointwise multiplication) for the gating mechanism of a layer. This multiplication acts as a binary gate, allowing information to pass if the value is close to 1 or blocking it when the value is close to 0. These gates allow a dynamic system in which during a time step, we decide how much information we preserve and how much we forget.

The first gate is called the **forget gate** because it is used to forget information that is no longer needed from the context and, therefore, will no longer be needed in the next time step. So, we will use the output of the forget gate to multiply the context. At this time, we extract both information from the input and the previous time step's hidden state. Each gate has a set of gate-specific *U* weights:

$$\boldsymbol{f}^{(t)} = \sigma(\boldsymbol{b_f} + \boldsymbol{U_f}\boldsymbol{h}^{(t-1)} + \boldsymbol{W_f}\boldsymbol{x}^t\Big)$$

kt=ct-1⊙ft

The next step is to extract information from the input and decide which of that information will be added to the context. This is controlled by an **input gate** *i* that controls how much information will then be added. The context is then obtained by the sum of what we add and what we forget:

$$\boldsymbol{g}^{(t)} = tanh(\boldsymbol{b_g} + \boldsymbol{U_g}\boldsymbol{h}^{(t-1)} + \boldsymbol{W_g}\boldsymbol{x}^t\Big)$$

$$\boldsymbol{i}^{(t)} = \sigma(\boldsymbol{b_i} + \boldsymbol{U_i}\boldsymbol{h}^{(t-1)} + \boldsymbol{W_i}\boldsymbol{x}^t\Big)$$

jt=g(t)⊙it

ct=j(t)+kt

The final step is the calculation of the output; this is achieved with a final gate. The output or final layer decision is also used to update the hidden state:

$$\boldsymbol{o}^{(t)} = \sigma(\boldsymbol{b_o} + \boldsymbol{U_o}\boldsymbol{h}^{(t-1)} + \boldsymbol{W_o}\boldsymbol{x}^t\Big)$$

kt=ot⊙tanh(ct

These gates are independent of each other so that efficient implementations of the LSTM can parallelize them. We can test in Python with a PyTorch RNN to see how it is transforming the data:

```
data_tensor = torch.tensor(np.random.random((10, 5, 3)),
                           dtype=torch.float32)
LSTM =nn.LSTM(input_size=3, hidden_size=10,
              num_layers=1, batch_first=True)
output, (hidden, cell) = LSTM(data_tensor)
output.shape
```

> ***IMPORTANT NOTE***
>
> *Notice how the model is transforming the data; we can access the hidden state as well as the cell state.*

We can also note some other interesting properties:

- Computation augmentation is internal to layers, which means we can easily substitute LSTMs for RNNs.
- The LSTM manages to preserve information for a long time because it retains only the relevant part of the information and forgets what is not needed.
- Standard practice is to initialize an LSTM with vector 1 (preserves everything), after which it learns by itself what to forget and what to add.

- An LSTM, as opposed to an RNN, can remember up to 100 time steps (the RNN after 7 time steps starts forgetting). The plus operation makes vanishing or exploding gradients less likely.

Let's look at another model option that is computationally lighter but still has this notion of context vector.

GRUs

GRUs are another variant of RNNs to solve the vanishing gradient problem, thus making them more effective in remembering information. They are very similar to LSTMs since they have internal gates, but they are much simpler and lighter. Despite having fewer parameters, GRUs can converge faster than LSTMs and still achieve comparable performance. GRUs exploit some of the elements that have made LSTMs so effective: the plus operation, the Hadamard product, the presence of a context, and the control of information within the layer:

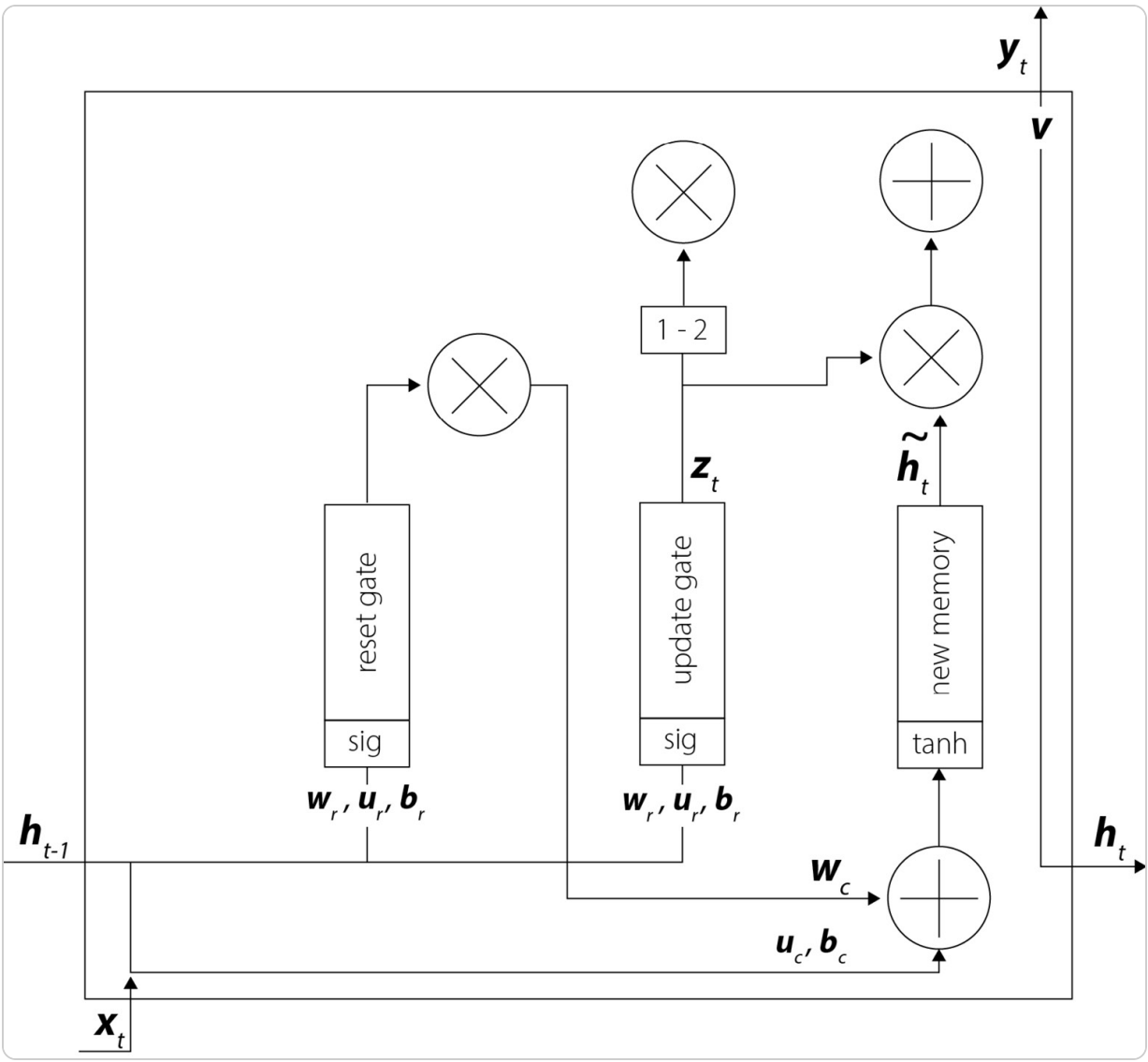

Figure 1.14 – Internal structure of a GRU cell (https://arxiv.org/pdf/2304.11461)

In a GRU, the forget gate is called the **update gate**, but it has the same purpose: important information is retained (values near 1) and unimportant information is rewritten during the update (values near 0). In a GRU, the input gate is called the **reset gate** and is not independent as in an LSTM, but connected to the update gate.

The first step is the update gate *z*, which is practically the same as the forget gate in an LSTM. At the same time, we calculate the reset gate *r*:

$$\boldsymbol{z}^{(t)} = \sigma(\boldsymbol{b}_{\boldsymbol{z}} + \boldsymbol{U}_{\boldsymbol{z}}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{\boldsymbol{z}}\boldsymbol{x}^{t})$$

$$\boldsymbol{r}^{(t)} = \sigma(\boldsymbol{b}_{\boldsymbol{r}} + \boldsymbol{U}_{\boldsymbol{r}}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{\boldsymbol{r}}\boldsymbol{x}^{t})$$

The next step is to update the hidden state; this depends on the reset gate. In this way, we decide what new information is put into the hidden state and what relevant information from the past is saved. This is called the **current memory gate**:

$$\overline{\boldsymbol{h}}^{(t)} = \tanh\left(\boldsymbol{W}_{\boldsymbol{h}}\boldsymbol{x}^{t} + \boldsymbol{r}^{(t)} \odot \boldsymbol{U}_{\boldsymbol{z}}\boldsymbol{h}^{(t-1)}\right)$$

At this point, we have the final update of the hidden state in which we also use the update gate:

$$\boldsymbol{h}^{(t)} = \boldsymbol{z}^{(t)} \odot \boldsymbol{h}^{(t-1)} + \left(1 - \boldsymbol{z}^{(t)}\right) \odot \overline{\boldsymbol{h}}^{(t)}$$

We can test in Python with a PyTorch RNN to see how it is transforming the data:

```
data_tensor = torch.tensor(np.random.random((10, 5, 3)),
                           dtype=torch.float32)
GRU =nn.GRU(input_size=3, hidden_size=10,
            num_layers=1, batch_first=True)
output, hidden = GRU(data_tensor)
output.shape
```

> ***IMPORTANT NOTE***
>
> *Notice how the model is transforming the data; we can also access the hidden state.*

We can see some interesting elements here as well:

- GRU networks are similar to LSTM networks, but they have the advantage of fewer parameters and are computationally more efficient. This means, though, they are more prone to overfitting.
- They can handle long sequences of data without forgetting previous inputs. For many textual tasks (but also speech recognition and music generation) they perform quite well, though they are less efficient than LSTMs when it comes to modeling long-term dependencies or complex patterns.

Next, we'll look at CNNs.

CNNs for text

CNNs are designed to find patterns in images (or other 2D matrixes) by running a filter (a matrix or kernel) along them. The convolution is conducted pixel by pixel, and the filter values are multiplied by the pixels in the image and then summed. During training, a weight is learned for each of the filter entries. For each filter, we get a different scan of the image that can be visualized; this is called a feature map.

Convolutional networks have been successful in **computer vision** because of their ability to extract local information and recognize complex patterns. For this reason, convolutional networks have been proposed for sequences. In this case, 1-dimensional convolutional networks are exploited, but the idea

is the same. In fact, on a sequence, 1D convolution is used to extract a feature map (instead of being a 2-dimensional filter or matrix, we have a uni-dimensional filter that can be seen as the context window of word2vec):

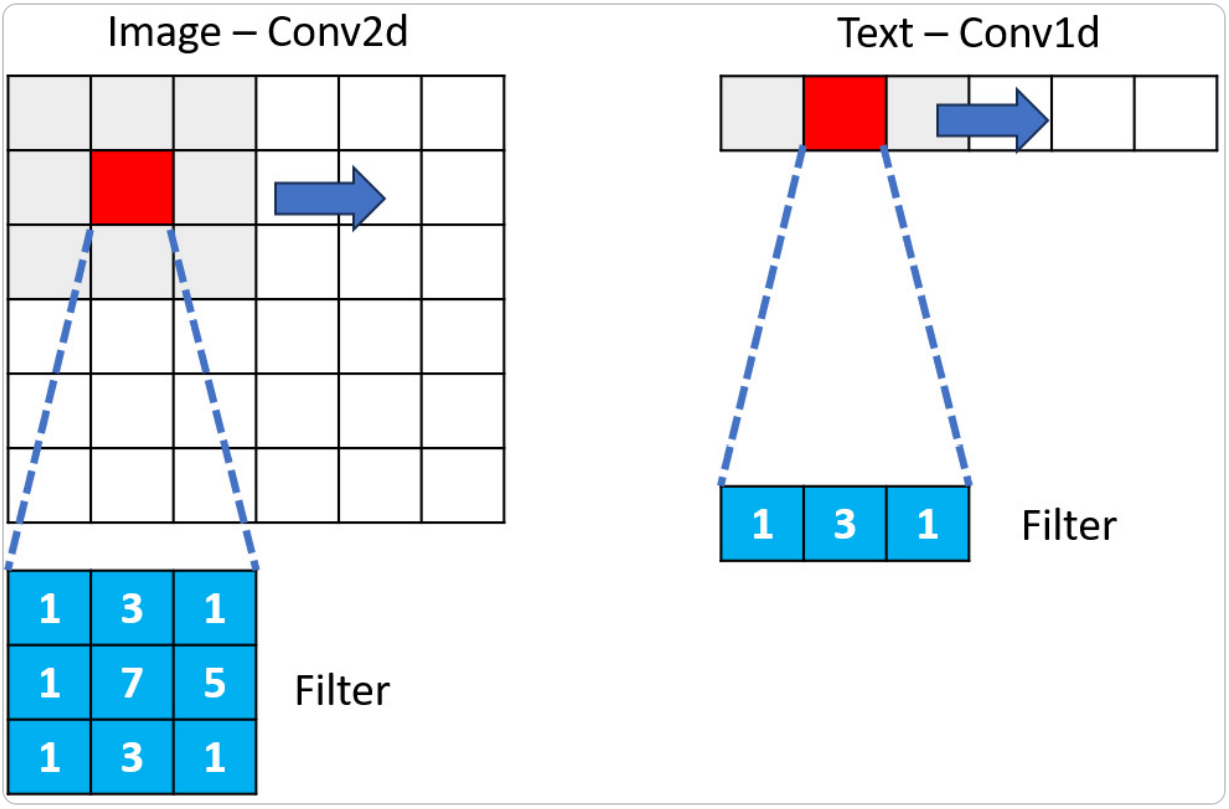

Figure 1.15 – In 1D convolution, we are sliding a 1D filter over the sequence

In the preceding figure, we scroll a uni-dimensional filter over the sequence; the process is very fast, and the filter can have an arbitrary size (three to seven words or even more). The model tries to learn patterns among the various words found within this kernel. It can also be used on vectors previously obtained from an embedding, and we can also use multiple kernels (so as to learn different patterns for each sequence). As with image CNNs, we can add operations such as max pooling to extract the most important features.

We can test in Python with a PyTorch RNN to see how it is transforming the data:

```
data_tensor = torch.tensor(np.random.random((10, 5, 3)),
                           dtype=torch.float32)
Conv1d = nn.Conv1d(in_channels=5, out_channels=16,
                   kernel_size=3, stride=1, padding=1)
output = Conv1d(data_tensor)
output.shape
```

> ***IMPORTANT NOTE***
>
> *Notice how the model is transforming the data and how this is different from what we have seen before.*

Now that we have a method to transform text into numerical representation (while preserving the contextual information) and models that can handle this representation, we can combine them to obtain an end-to-end system.

Performing sentiment analysis with embedding and deep learning

In this section, we will train a model for conducting sentiment analysis on movie reviews. The model we will train will be able to classify reviews as positive or negative. To build and train the model, we will exploit the elements we have encountered so far. In brief, we're doing the following:

- We are preprocessing the dataset, transforming in numerical vectors, and harmonizing the vectors
- We are defining a neural network with an embedding and training it

The dataset consists of 50,000 positive and negative reviews. We can see that it contains a heterogeneous length for reviews and that on average, there are 230 words:

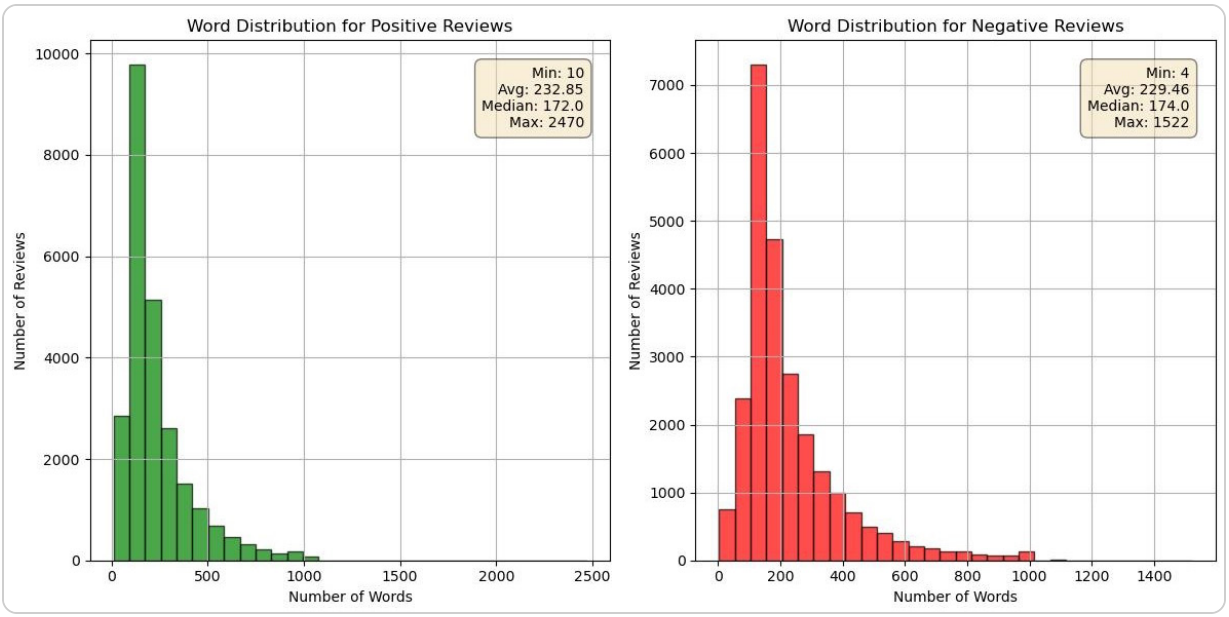

Figure 1.16 – Graphs showing the distribution of the length of the review in the text; the left plot is for positive reviews, while the right plot is for negative reviews

In addition, the most prevalent words are, obviously, "*movie*" and "*film*":

Figure 1.17 – Word cloud for the most frequent words in positive (left plot) and negative (right plot) reviews

The text is messy and must be cleaned before the model can be trained. The first step is binary encoding of the label ("positive" equals 0, "negative" equals 1). After that, we divide the features and the labels (for a dataset in **supervised learning**, `x` are the features and `y` are the labels). Next, we create three balanced datasets for training, validation, and testing:

```
df['sentiment_encoded'] = np.where(
    df['sentiment']=='positive',0,1)
X,y = df['review'].values, df['sentiment_encoded'].values
x_train,x_test,y_train,y_test = train_test_split(
    X,y,stratify=y, test_size=.2)
x_train,x_val,y_train,y_val = train_test_split(
    x_train,y_train,stratify=y_train, test_size=.1)
y_train, y_val, y_test = np.array(y_train), np.array(y_val), \
    np.array(y_test)
```

A few steps are necessary before proceeding with the training:

- A **preprocessing** step in which we remove excessive spaces, special characters, and punctuation.
- A **tokenization** step in which we convert the various reviews into tokens. In this step, we also remove stopwords and single-character words. We extract for each review only the 1,000 most popular words (this step is only to reduce computation time during training).
- Transformation of the the words into indices (**vectorization**) according to our vocabulary to make the model work with numerical values.
- Since the reviews have different lengths, we apply padding to harmonize the length of the review to a fixed number (we need this for the training).

These preprocessing steps depend on the dataset. The code is in the GitHub repository. Note, however, that the tokenization and preprocessing choices alter the properties of the reviews – in this case, the

summary statistics.

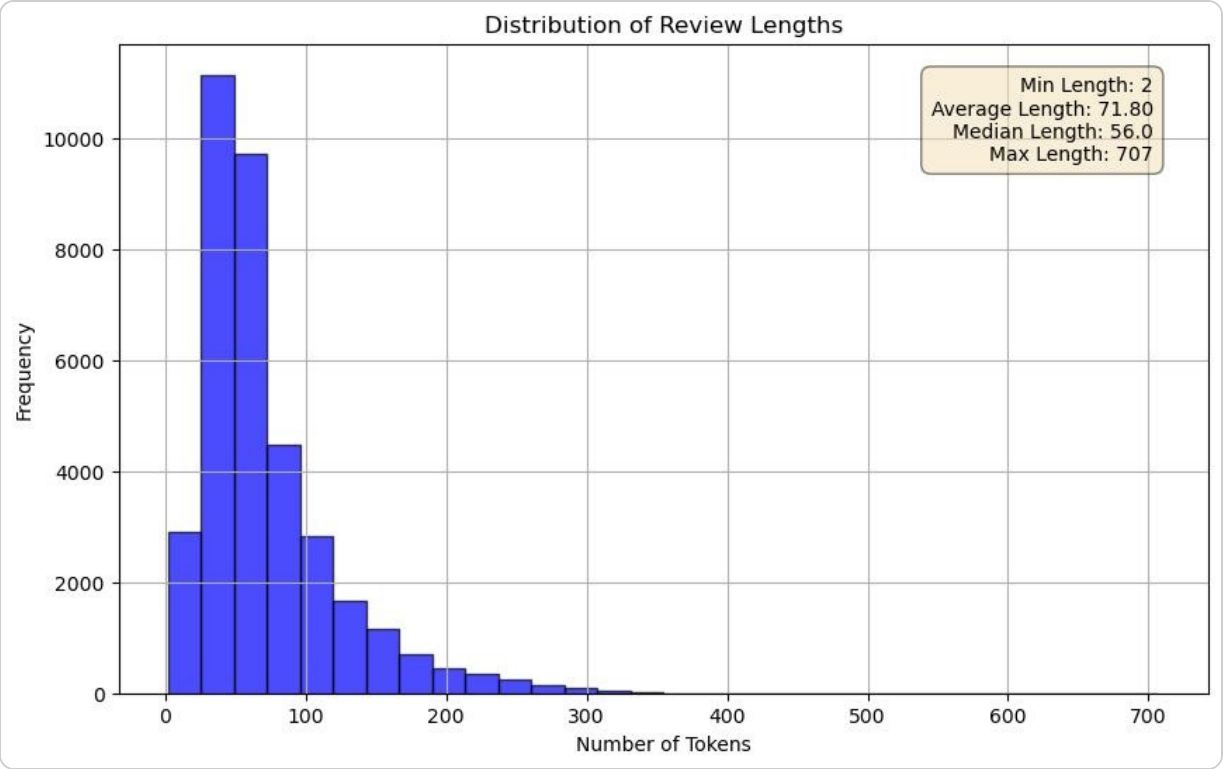

Figure 1.18 – Graph showing the distribution of review length after tokenization

We are defining the model with its hyperparameters. In this case, we are training a neural network to predict sentiment data composed of embeddings and GRUs. To make the training more stable, we add regularization (dropout). The linear layer is to map these features that we extracted to a single representation. We use this representation to calculate the probability that the review is positive or negative:

```
# Hyperparameters
no_layers = 3
vocab_size = len(vocab) + 1  # extra 1 for padding
embedding_dim = 300
output_dim = 1
hidden_dim = 256
# Initialize the model
model = SentimentRNN(no_layers, vocab_size, hidden_dim,
                     embedding_dim, drop_prob=0.5)
```

Note that in this case, we use binary cross-entropy loss because we have only two categories (positive and negative). Also, we use **`Adam`** as an optimizer, but one can test others. In this case, we conduct batch training since we have thousands of reviews:

```
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
epoch_tr_loss, epoch_vl_loss = [], []
```

```
epoch_tr_acc, epoch_vl_acc = [], []
for epoch in range(epochs):
    train_losses = []
    train_acc = 0.0
    model.train()
    h = model.init_hidden(50)
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        h = h.data
        model.zero_grad()
        output, h = model(inputs, h)
        loss = criterion(output.squeeze(), labels.float())
        loss.backward()
        train_losses.append(loss.item())
        accuracy = acc(output, labels)
        train_acc += accuracy
        optimizer.step()
```

The following graph displays the accuracy and loss for the training and validation sets:

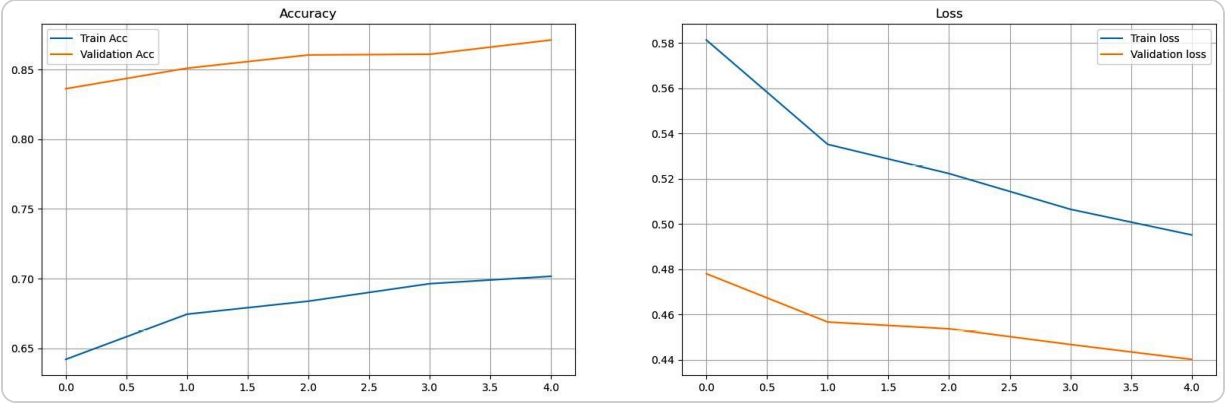

Figure 1.19 – Training curves for training and validation set, for accuracy and loss

The model achieves good accuracy, as we can see from the following confusion matrix:

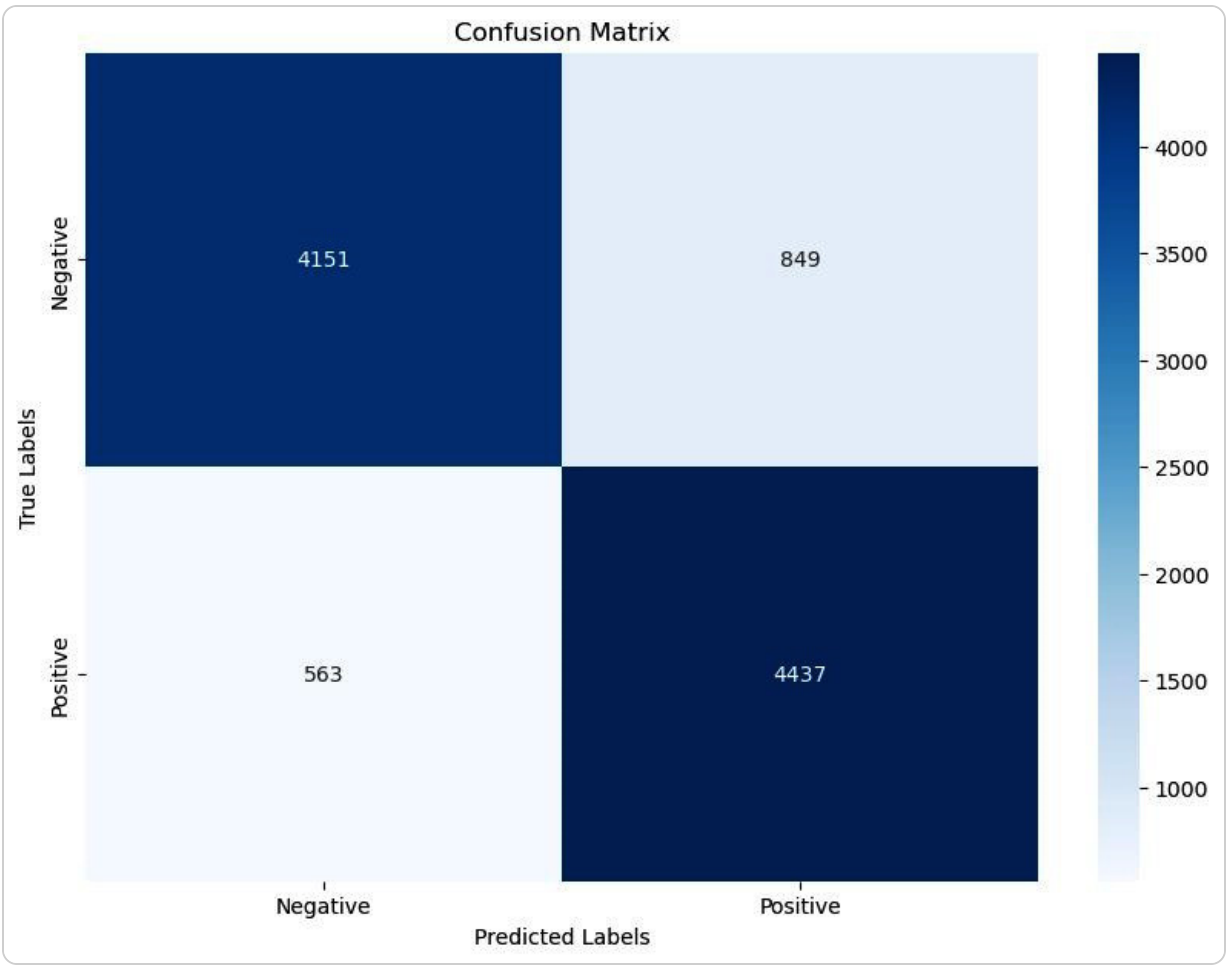

Figure 1.20 – Confusion matrix for the test set

In addition, if we look at the projection of reviews before and after the training, we can see that the model has learned how to separate positive and negative reviews:

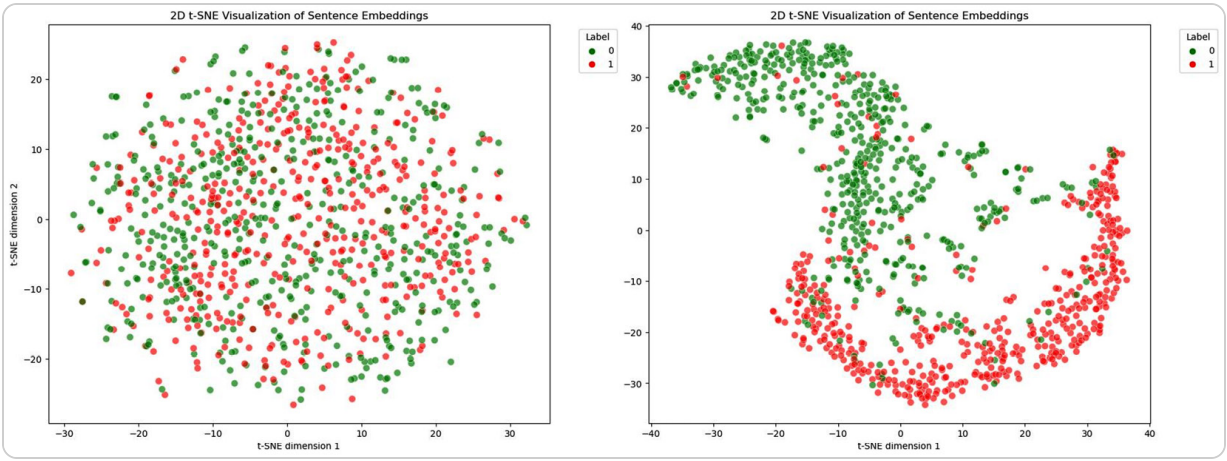

Figure 1.21 – Embedding projection obtained from the model before (left plot) and after (right plot) training

We have now trained a model that can take a review in plain text and classify it as positive or negative. We did that by combining the elements we saw previously in the chapter. The same approach can be followed with any other dataset; that is the power of deep learning.

Summary

In this chapter, we saw how to transform text to an increasingly complex vector representation. This numerical representation of text allowed us to be able to use machine learning models. We saw how to preserve the contextual information (word embedding) of a text and how this can then be used for later analysis (for example, searching synonyms or clustering words). In addition, we saw how neural networks (RNNs, LSTM, GRUs) can be used to analyze text and perform tasks (for example, sentiment analysis).

In the next chapter, we will see how to solve some of the remaining unsolved challenges and see how this will lead to the natural evolution of the models seen here.

2
The Transformer: The Model Behind the Modern AI Revolution

In this chapter, we will discuss the limitations of the models we saw in the previous chapter, and how a new paradigm (first attention mechanisms and then the transformer) emerged to solve these limitations. This will enable us to understand how these models are trained and why they are so powerful. We will discuss why this paradigm has been successful and why it has made it possible to solve tasks in **natural language processing** (**NLP**) that were previously impossible. We will then see the capabilities of these models in practical application.

This chapter will clarify why contemporary LLMs are inherently based on the transformer architecture.

In this chapter, we'll be covering the following topics:

- Exploring attention and self-attention
- Introducing the transformer model
- Training a transformer
- Exploring masked language modeling
- Visualizing internal mechanisms
- Applying a transformer

Technical requirements

Most of this code can be run on a CPU, but some parts (fine-tuning and knowledge distillation) are preferable to be run on a GPU (one hour of training on a CPU versus less than five minutes on a GPU).

The code is written in PyTorch and uses standard libraries for the most part (PyTorch, Hugging Face Transformers, and so on), though some snippets come from Ecco, a specific library. The code can be found on GitHub: https://github.com/PacktPublishing/Modern-AI-Agents/tree/main/chr2

Exploring attention and self-attention

In the 1950s, with the beginning of the computer revolution, governments began to become interested in the idea of machine translation, especially for military applications. These attempts failed miserably, for three main reasons: machine translation is more complex than it seems, there was not enough

computational power, and there was not enough data. Governments concluded that it was a technically impossible challenge in the 1960s.

By the 1990s, two of the three limitations were beginning to be overcome: the internet finally allowed for abundant text, and the advent of GPUs finally allowed for computational power. The third requirement still had to be met: a model that could harness the newfound computational power to handle the complexity of natural language.

Machine translation captured the interest of researchers because it is a practical problem for which it is easy to evaluate the result (we can easily understand whether a translation is good or not). Moreover, we have an abundance of text in one language and a counterpart in another. So, researchers tried to adapt the previous models to the tasks (RNN, LSTM, and so on). The most commonly used system was the **seq2seq model**, where you have an **encoder** and a **decoder**. The encoder transforms the sequence into a new succinct representation that should preserve the relevant information (a sort of good summary). The decoder receives as input this context vector and uses this to transform (translate) this input into the output.

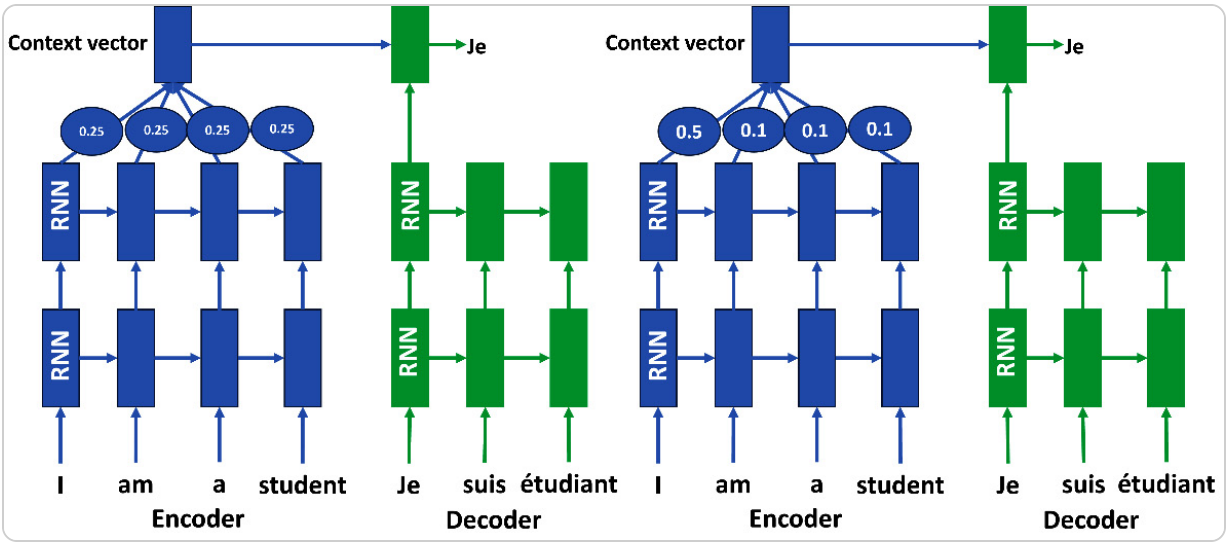

Figure 2.1 – A seq2seq model with an encoder and a decoder. In one case, we take the average of the hidden states (left); in the other case, we use attention to identify which hidden state is more relevant for the translation (right)

RNNs and derived models have some problems:

- **Alignment**: The length of input and output can be different (for example, to translate English to French "*she doesn't like potatoes*" into "*elle n'aime pas les pommes de terre*").
- **Vanishing and exploding gradients**: Problems that arise during training so that multiple layers cannot be managed effectively.
- **Non-parallelizability**: Training is computationally expensive and not parallelizable. RNNs forget after a few steps.

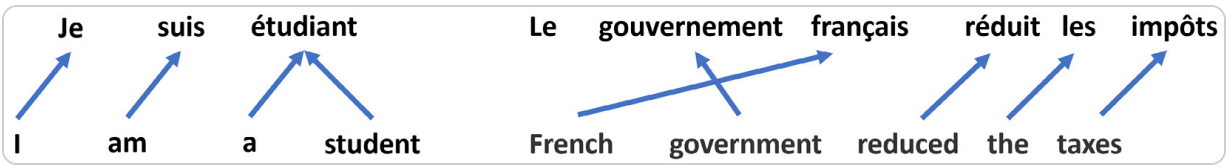

Figure 2.2 – Example of issues with alignment: one to many (left) and spurious word (right)

Attention mechanisms were initially described to solve the alignment problem, as well as to learn the relationships between the various parts of a text and the corresponding parts of the translated text.

The idea is that instead of passing the hidden state of RNNs, we pass contextual information that focuses only on the important parts of the sequence. During decoding (translation) for each token, we want to retrieve the corresponding and specific information in the other language. Attention determines which tokens in the input are important at that moment.

The first step is the alignment between the hidden state of the encoder (h) and the previous decoder output (s). The score function can be different: dot product or cosine similarity is most commonly used, but it can also be more complex functions such as the feedforward neural network layer. This step allows us to understand how relevant hidden state encoders are to the translation at that time. This step is conducted for all encoder steps.

$$(1) e_{i,j} = score(s_{i-1}, h_j)$$

Right now though, we have a scalar representing the similarity between two vectors (h and s). All these scores are passed into the softmax function that squeezes everything between 0 and 1. This step also serves to assign relative importance to each hidden state.

$$(2) a_{i,j} = softmax(e_{i,j}) = \frac{\exp(e_{i,j})}{\sum_{k=1}^{t} \exp(e_{i,k})}$$

Finally, we conduct a weighted sum of the various hidden states multiplied by the attention score. So, we have a fixed-length context vector capable of giving us information about the entire set of hidden states. In simple words, during translation, we have a context vector that is dynamically updated and tells us how much attention we should give to each part of the input sequence.

$$(3) c_t = \sum_{j=1}^{T} a_{i,j} \bullet h_j$$

As you can see from the original article, the model pays different attention to the various words in the input during translation.

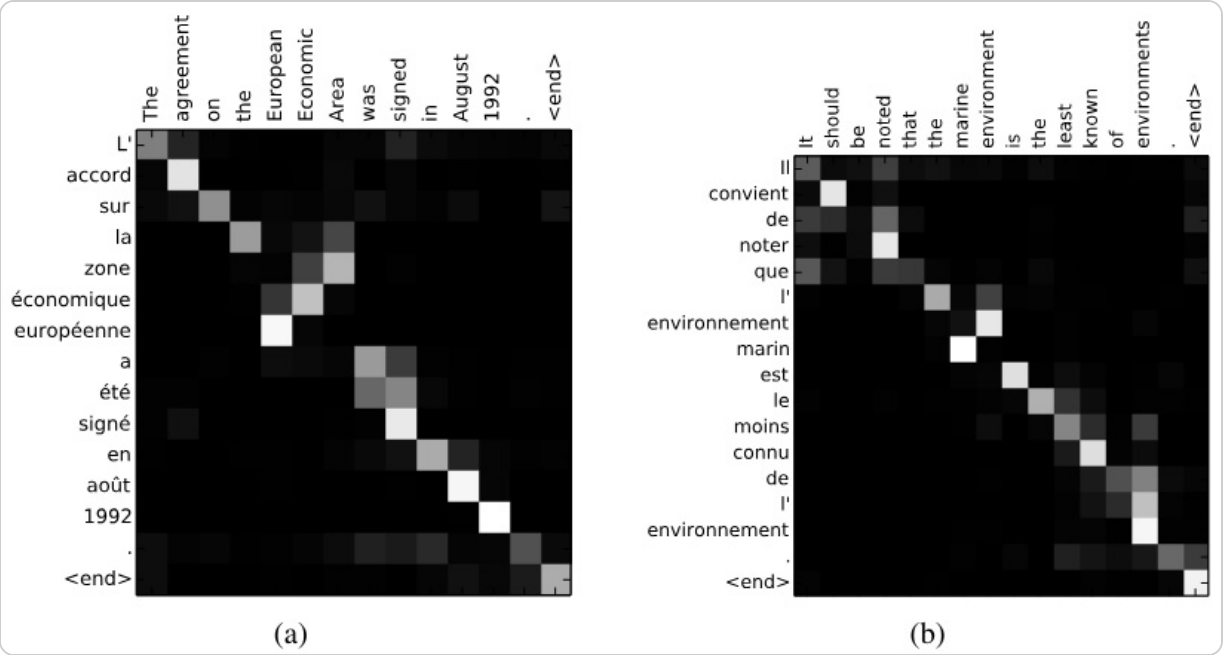

Figure 2.3 – Example of alignment between sentences after training the model with attention. Each pixel shows the attention weight between the source word and the target word. (https://arxiv.org/pdf/1409.0473)

In addition to solving alignment, the attention mechanism has other advantages:

- It reduces the vanishing gradient problem because it provides a shortcut to early states.
- It eliminates the bottleneck problem; the encoder can directly go to the source in the translation.
- It also provides interpretability because we know which words are used for alignment.
- It definitely improves the performance of the model.

Its success has given rise to several variants where the scoring function is different. One variant in particular, called **self-attention**, has the particular advantage that it extracts information directly from the input without necessarily needing to compare it with something else.

The insight behind self-attention is that if we want to look for a book for an essay on the French Revolution in a library (query), we don't need to read all the books to find a book on the history of France (value), we just need to read the coasts of the books (key). Self-attention, in other words, is a method that allows us to search within context to find the representation we need.

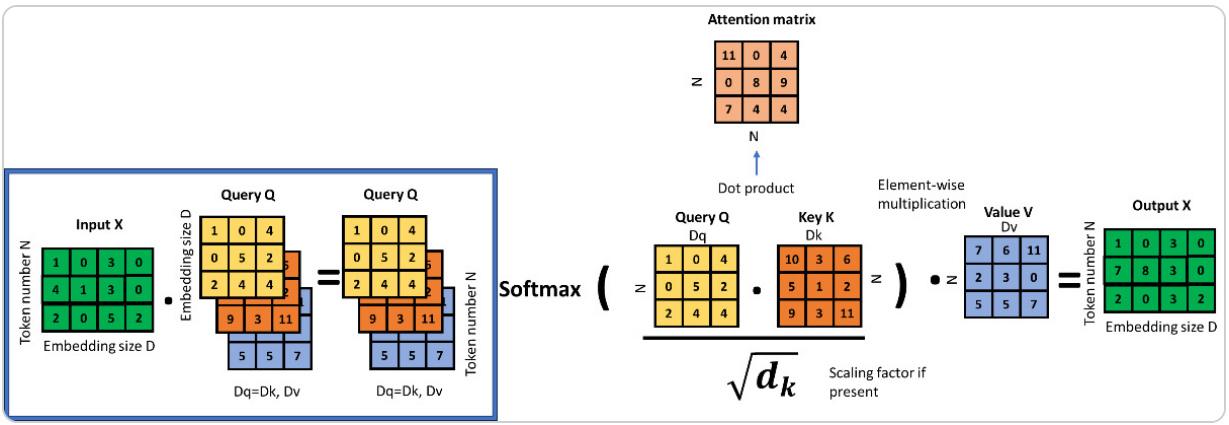

Figure 2.4 – Self-attention mechanism. Matrix dimensions are included; numbers are arbitrary

Transacting this for a model, given an input we want to conduct a series of comparisons between the various components of the sequence (such as tokens) to obtain an output sequence (which we can then use for various models or tasks). The self-attention equation is as follows:

$$Attention(Q, K, V) = softmax\left(\frac{Q \bullet k^T}{\sqrt{d_k}}\right) \bullet V$$

You can immediately see that it is derived from the original attention formula. We have the dot product to conduct comparisons, and we then exploit the `softmax` function to calculate the relative importance and normalize the values between 0 and 1. *D* is the size of the sequence; in other words, self-attention is also normalized as a function of the length of our sequence.

The next step is using `softmax`. Here's a little refresher on the function (how you calculate and how it is implemented more efficiently in Python):

$$y = \frac{e^{x_i}}{\sum_{i=1}^{i=n} e^{x_i}}$$

$$y = \begin{matrix} y_1 \\ y_2 \\ y_3 \end{matrix} = softmax(x) = f \begin{matrix} x_1 \\ x_2 \\ x_3 \end{matrix} = \begin{matrix} \frac{e^{x_1}}{e^{x_1}+e^{x_2}+e^{x_3}} \\ \frac{e^{x_2}}{e^{x_1}+e^{x_2}+e^{x_3}} \\ \frac{e^{x_3}}{e^{x_1}+e^{x_2}+e^{x_3}} \end{matrix}$$

$$python: y = \begin{matrix} y_1 \\ y_2 \\ y_3 \end{matrix} = softmax(x) = f \begin{matrix} x_1 \\ x_2 \\ x_3 \end{matrix} = \frac{[x_1 \quad x_2 \quad x_3]}{e^{x_1} + e^{x_2} + e^{x_3}} = \frac{e^x}{sum(e^x)}$$

As we saw in the previous chapter, the dot product can become quite wide as the length of the vectors increases. This can lead to inputs that are too large in the `softmax` function (this shifts the probability mass in `softmax` to a few elements and thus leads to small gradients). In the original article, they solved this by normalizing by the square root of *D*.

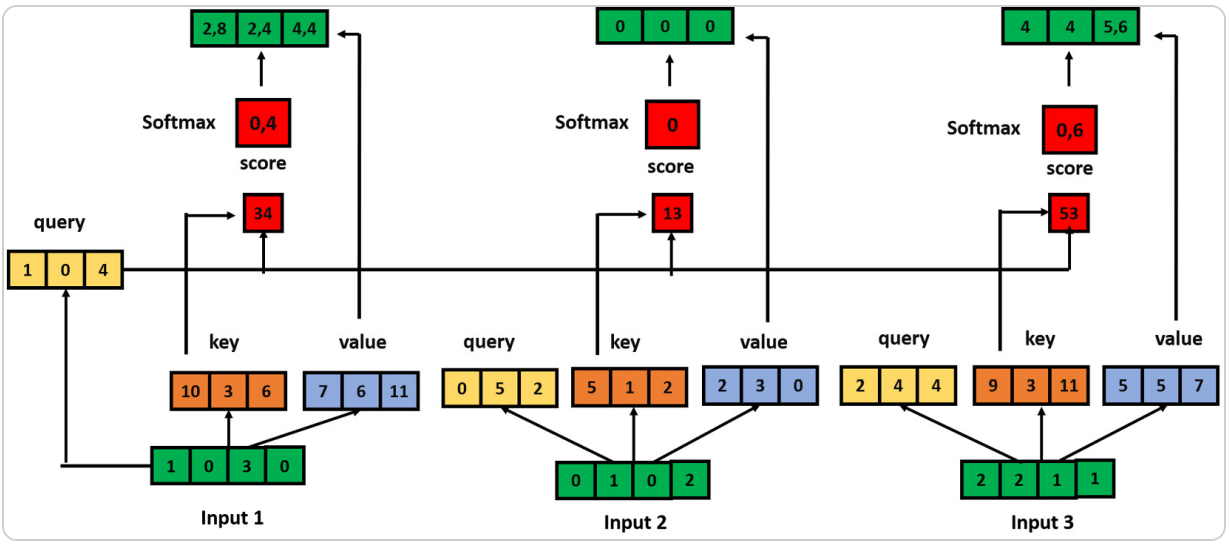

Figure 2.5 – Self-attention unrolled

The real difference is that we use three matrices of weights **Query** (**Q**), **Key** (**K**), and **Value** (**V**) that are initially randomly initialized. *Q* is the current focus of attention, while *K* informs the model about previous inputs, and *V* serves to extract the final input information. So, the first step is the multiplication of these three matrices with our input *X* (an array of vectors, of which each represents a token).

$$\boldsymbol{Q} = \boldsymbol{X} \bullet \boldsymbol{W}^{Q}, \boldsymbol{K} = \boldsymbol{X} \bullet \boldsymbol{W}^{V}, \boldsymbol{V} = \boldsymbol{X} \bullet \boldsymbol{W}^{V}$$

The beauty of this system is that we can use it to extract more than one representation from the same input (after all, we can have multiple questions in a textbook). Therefore, since the operations are parallelizable, we can have multi-head attention. **Multi-head self-attention** enables the model to simultaneously capture multiple types of relationships within the input sequence. This is crucial because a single word in a sentence can be contextually related to several other words. During training, the *K* and *Q* matrices in each head specialize in modeling different kinds of relationships. Each attention head produces an output based on its specific perspective, resulting in *n* outputs for *n* heads. These outputs are then concatenated and passed through a final linear projection layer to restore the dimensionality backt to the original input size.

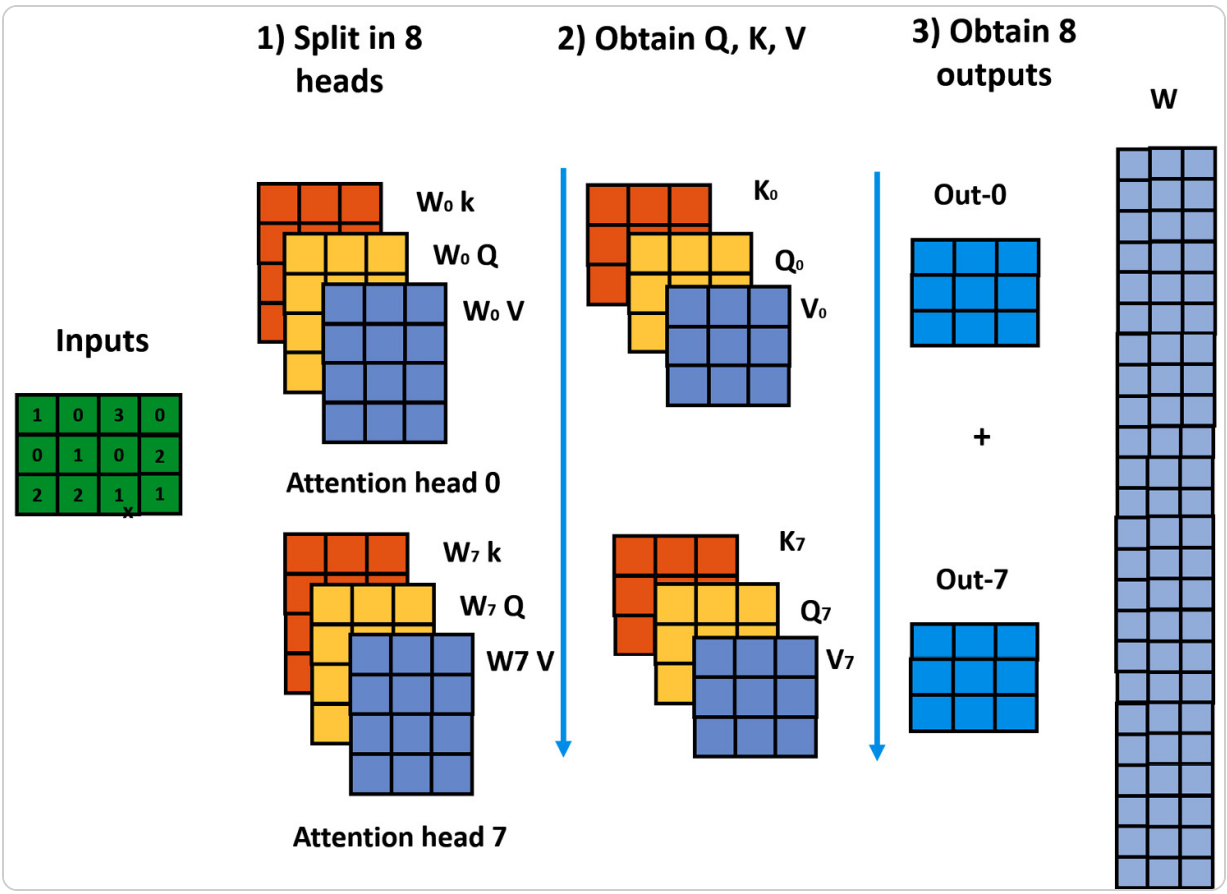

Figure 2.6 – Multi-head self-attention

Self-attention has several advantages:

- We can extract different representations for each input.
- We can conduct all these computations in parallel and thus with a GPU. Each head can be computed independently.
- We can use it in models that do not necessarily consist of an encoder and a decoder.
- We do not have to wait for different time steps to see the relationship between distant word pairs (as in RNN).
- However, it has a quadratic cost in function of the number of tokens *N*, and it has no inherent notion of order.

Self-attention is computationally expensive. It can be shown that, considering a sequence T and sequence length d, the computation cost and space is quadratic:

$$time = \mathcal{O}\left(T^2 + d\right) space = \mathcal{O}\left(T^2 + Td\right)$$

They identified the dot product as the culprit. This computational cost is one of the problems of scalability (taking into account that multi-head attention is calculated in each block). For this reason, many variations of self-attention have been proposed to reduce the computational cost.

Despite the computational cost, self-attention has shown its capability, especially when several layers are stacked on top of each other. In the next section, we will discuss how this makes the model extremely powerful despite its computational cost.

Introducing the transformer model

Despite this decisive advance though, several problems remain in machine translation:

- The model fails to capture the meaning of the sentence and is still error-prone
- In addition, we have problems with words that are not in the initial vocabulary
- Errors in pronouns and other grammatical forms
- The model fails to maintain context for long texts
- It is not adaptable if the domain in the training set and test data is different (for example, if it is trained on literary texts and the test set is finance texts)
- RNNs are not parallelizable, and you have to compute sequentially

Considering these points, Google researchers in 2016 came up with the idea of eliminating RNNs altogether rather than improving them. According to the authors of the *Attention is All You Need* seminal article; all you need is a model that is based on multi-head self-attention. Before going into detail, the transformer consists entirely of stacked layers of multi-head self-attention. In this way, the model learns a hierarchical and increasingly sophisticated representation of the text.

The first step in the process is the transformation of text into numerical vectors (tokenization). After that, we have an embedding step to obtain vectors for each token. A special feature of the transformer is the introduction of a function to record the position of each token in the sequence (self-attention is not position-aware). This process is called **positional encoding**. The authors in the article use sin and cos alternately with position. This allows the model to know the relative position of each token.

$$PE_{pos,2i}=\sin(pos/10002i/d)$$

$$PE_{pos,2i+1}=\cos(pos/10002i/d)$$

In the first step, the embedding vectors are summed with the result of these functions. This is because self-attention is not aware of word order, but word order in a period is important. Thus, the order is directly encoded in the vectors it awaits. Note, though, that there are no learnable parameters in this function and that for long sequences, it will have to be modified (we will discuss this in the next chapter).

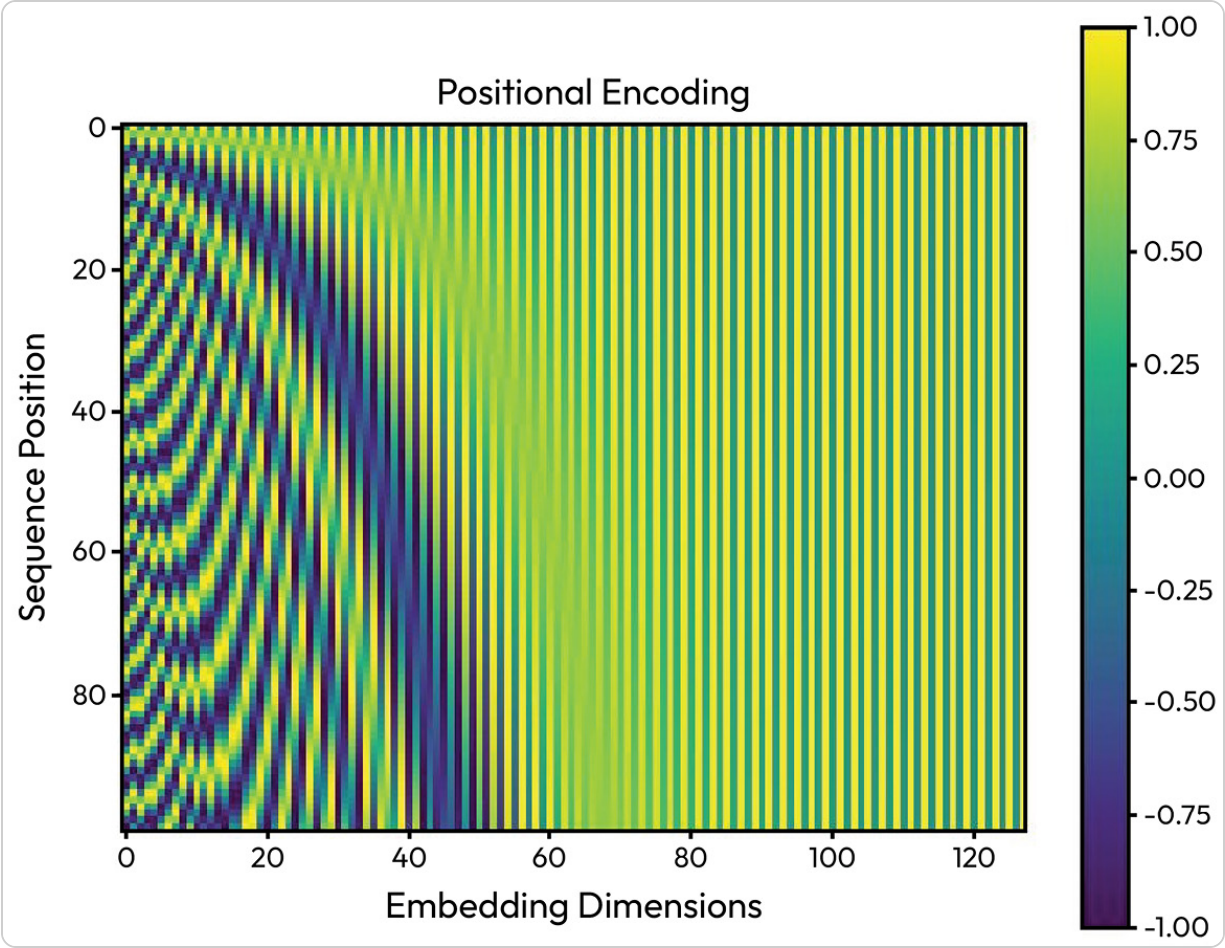

Figure 2.7 – Positional encoding

After that, we have a series of transformer blocks in sequence. The **transformer block** consists of four elements: multi-head self-attention, feedforward layer, residual connections, and layer normalization.

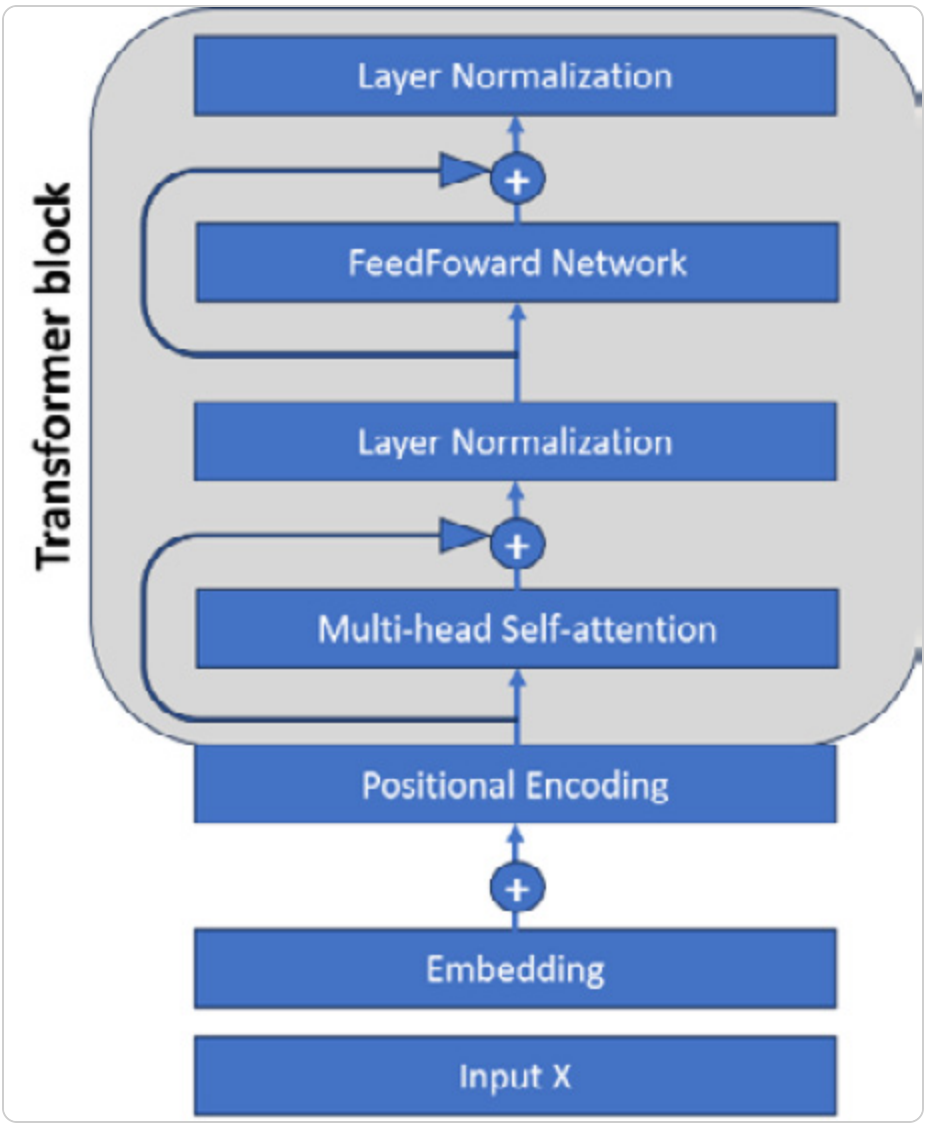

Figure 2.8 – Flow diagram of the transformer block

The feedforward layer consists of two linear layers. This layer is used to obtain a linear projection of the multi-head self-attention. The weights are identifiable for each position and are separated. It can be seen as two linear transformations with one ReLU activation in between.

$$FFNx=max(0,xW1+b1)W2+b2$$

This adds a step of non-linearity to self-attention. The FFN layer is chosen because it is an easily parallelized operation.

Residual connections are connections that pass information between two layers without going through the intermediate layer transformation. Initially developed in convolutional networks, they allow a shortcut between layers and help the gradient pass down to the lower layers. In the transformer, blocks are present for both the attention layer and feedforward, where the input is

summed with the output. Residual connections also have the advantage of making the loss surface smoother (this helps the model find a better minimum and not get stuck in a local loss). This powerful effect can be seen clearly in *Figure 2.9*:

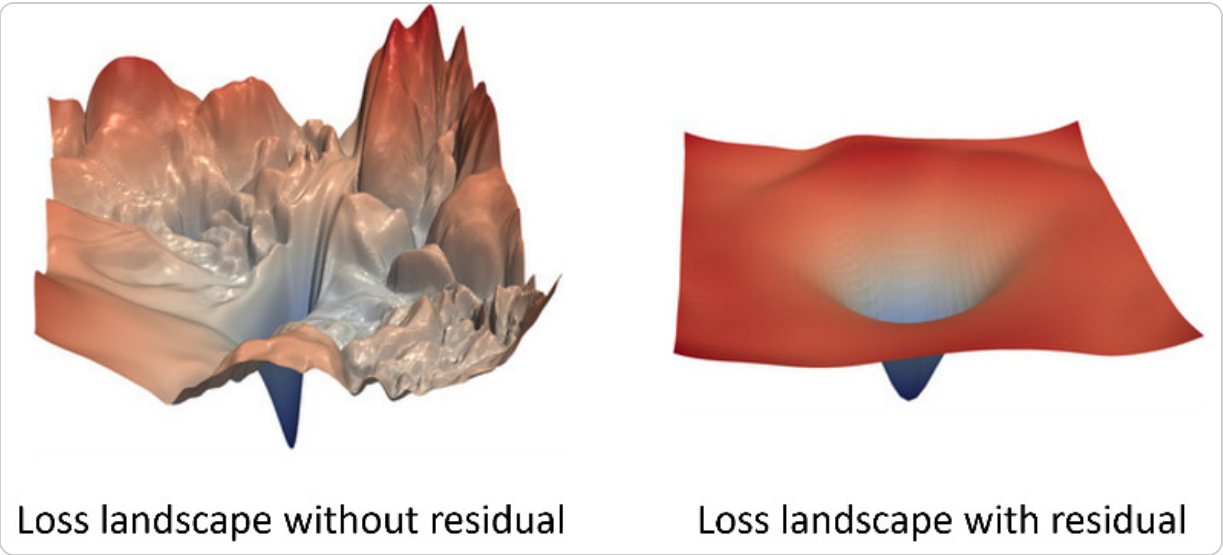

Figure 2.9 – Effect of the residual connections on the loss

> ***NOTE***
>
> *Figure 2.9 is originally from Visualizing the Loss Landscape of Neural Nets by Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein (https://github.com/tomgoldstein/loss-landscape/tree/master).*

The residual connection makes the loss surface smoother, which allows the model to be trained more efficiently and quickly.

Layer normalization is a form of normalization that helps training because it keeps the hidden layer values in a certain range (it is an alternative to batch normalization). Having taken a single vector, it is normalized in a process that takes advantage of the mean and standard deviation. Having calculated the mean and standard deviation, the vector is scaled:

$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i \sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2}$$

$$\widehat{x} = \frac{(x - \mu)}{\sigma}$$

In the final transformation, we exploit two parameters that are learned during training.

$$LayerNormalization = \gamma\widehat{x} + \beta$$

There is a lot of variability during the training, and this can hurt the learning of the training. To reduce uninformative variability, we add this normalization step, thus normalizing the gradient as well.

At this point, we can assemble everything into a single block. Consider that after embedding, we have as input *X* a matrix of dimension *n x d* (with *n* being the number of tokens, and *d* the dimensions of the embedding). This input *X* goes into a transformer block and comes out with the same dimensions. This process is repeated for all transformer blocks:

$$H=\text{LayerNorm}X+\text{MultiHeadSelfAttention}X$$

$$H=\text{LayerNorm}H+\text{FFN}H$$

Some notes on this process are as follows:

- In some architectures, *LayerNorm* can be after the *FFN* block instead of before (whether it is better or not is still debated).
- Modern models have up to 96 transformer blocks in series, but the structure is virtually identical. The idea is that the model learns an increasingly complex representation of the language.
- Starting with the embedding of an input, self-attention allows this representation to be enriched by incorporating an increasingly complex context. In addition, the model also has information about the location of each token.
- Absolute positional encoding has the defect of overrepresenting words at the beginning of the sequence. Today, there are variants that consider the relative position.

Once we have "the bricks," we can assemble them into a functional structure. In the original description, the model was structured for machine translation and composed of two parts: an encoder (which takes the text to be translated) and a decoder (which will produce the translation).

The original transformer is composed of different blocks of transformer blocks and structures in an encoder and decoder, as you can see in *Figure 2.10*.

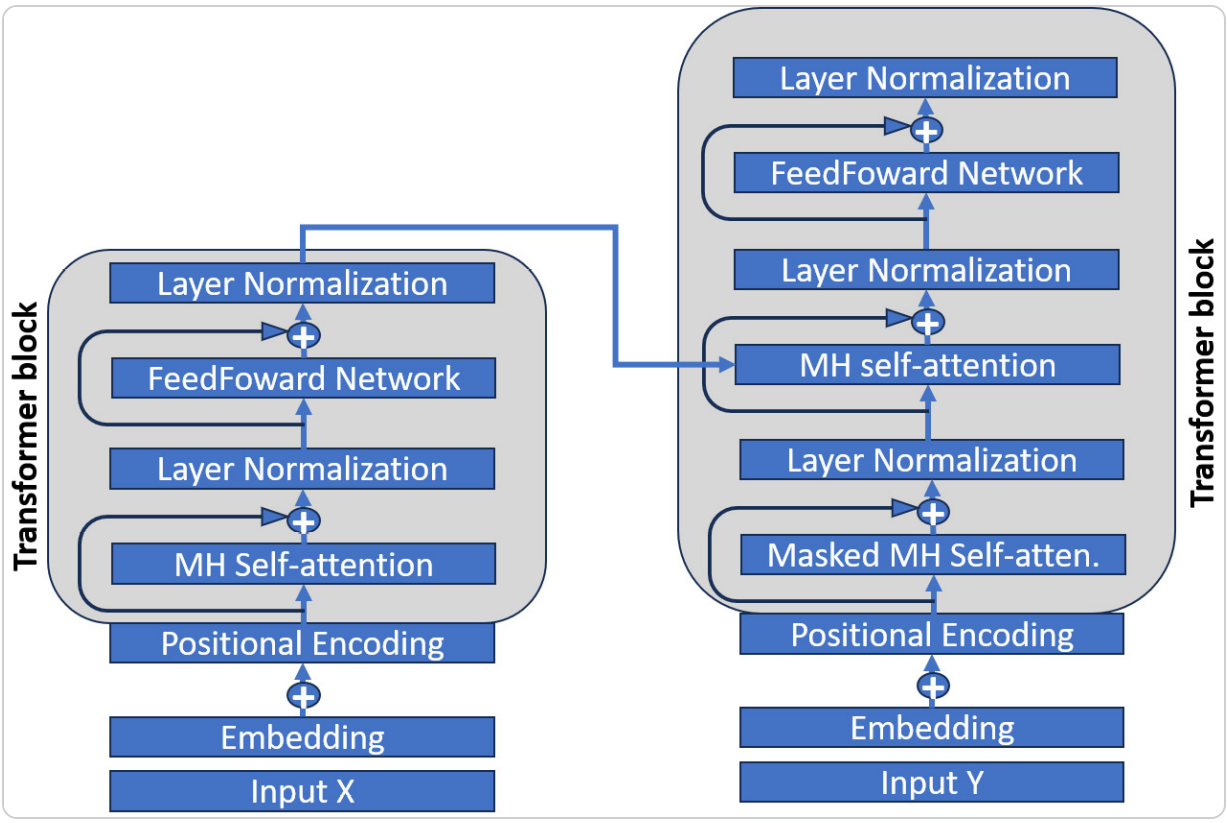

Figure 2.10 – Encoder-decoder structure

The decoder, like the encoder, is composed of an embedding, a positional encoder, and a series of transformer blocks. One note is that in the decoder, instead of self-attention, we have **cross-attention**. Cross-attention is exactly the same, only we take both elements from the encoder and the decoder (because we want to condition the generation of the decoder based on the encoder input). In this case, the queries come from the encoder and the rest from the decoder. As you can see from *Figure 2.11*, the decoder sequence can be of different sizes, but the result is the same:

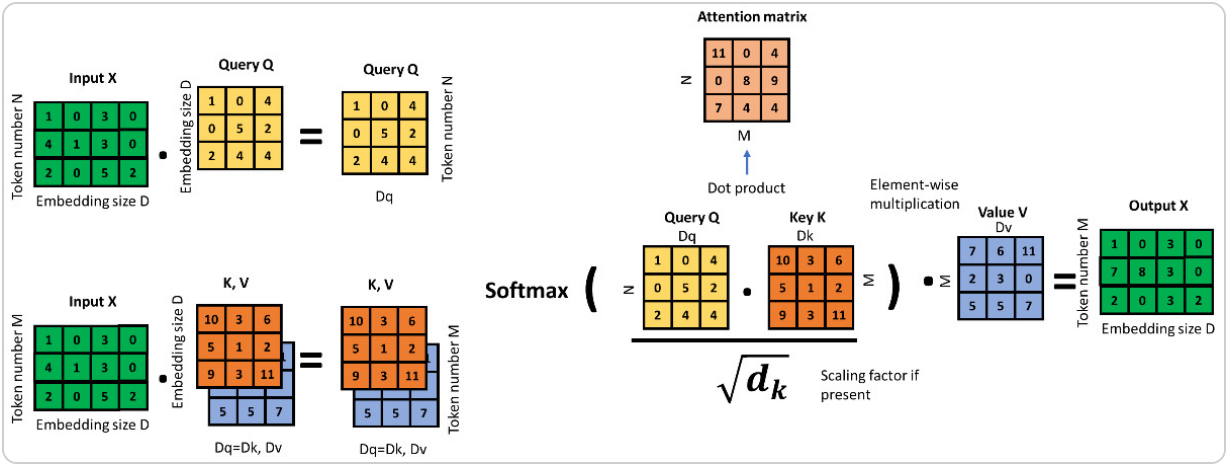

Figure 2.11 – Cross-attention

Input *N* comes from the encoder, while input *M* is from the decoder. In the figure, cross-attention is mixing information from the encoder and decoder, allowing the decoder to learn from the encoder.

Another note on the structure: in the decoder, the first self-attention has an additional mask to prevent the model from seeing the future.

This is especially true in the case of QT. In fact, if one wants to predict the next word and the model already knows what it is, we have data leakage. To compensate for this, we add a mask in which the upper-triangular portion is replaced with negative infinity: - ∞.

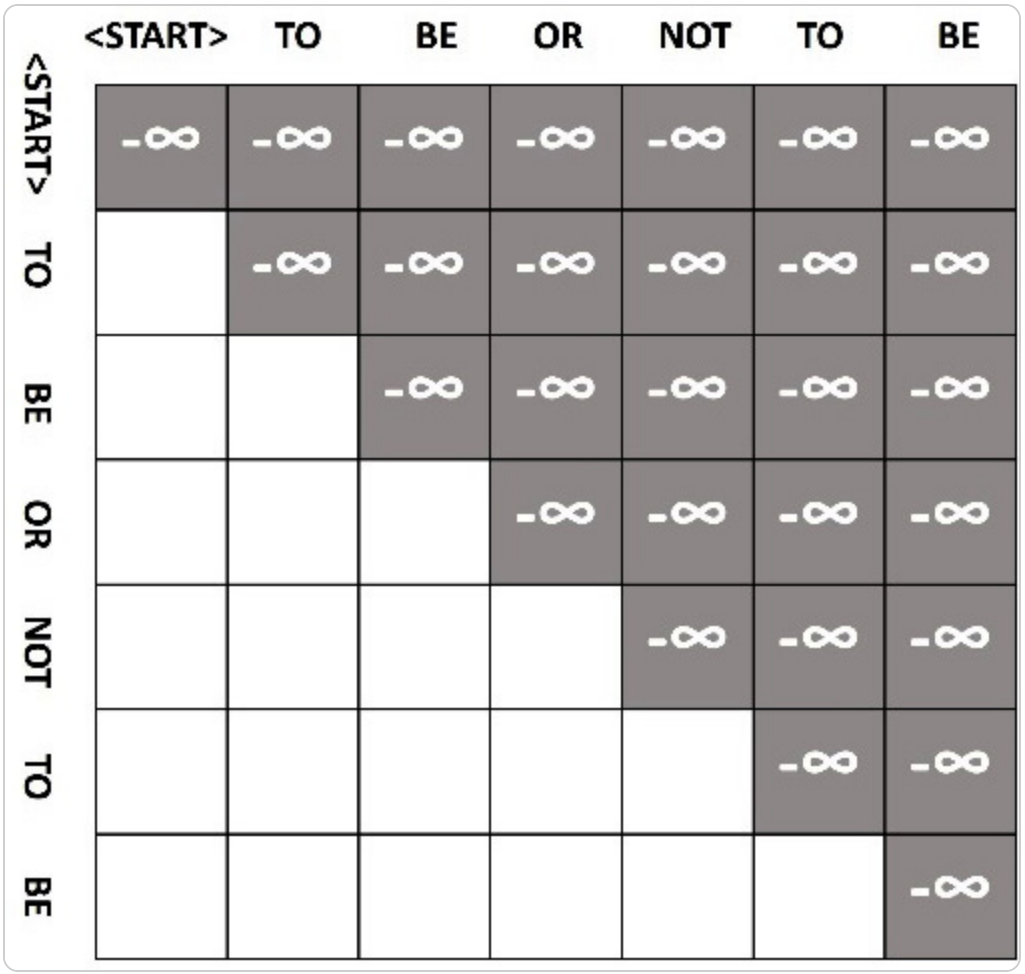

Figure 2.12 – Masked attention

The first transformer consisted of an encoder and decoder, but today there are also models that are either encoder-only or decoder-only. Today, for generative AI, they are practically all decoder-only. We have our model; now, how can you train a system that seems so complex? In the next section, we will see how to succeed at training.

Training a transformer

How do you train such a complex model? The answer to this question is simpler than you might think. The fact that the model can learn through multi-head self-attention complex and diverse relationships allows the model to be able to be flexible and able to learn complex patterns. It would be too expensive to build examples (or find them) to teach these complex relationships to the model. So, we want a system that allows the model to learn these relationships on its own. The advantage is that if we have a large amount of text available, the model can learn without the need for us to curate the training corpus. Thanks to the advent of the internet, we have the availability of huge corpora that allow models to see text examples of different topics, languages, styles, and more.

Although the original model was a `seq2seq` model, later transformers (such as LLMs) were trained as language models, especially in a **self-supervised manner**. In language modeling, we consider a sequence of word *s*, and the probability of the next word in the sequence *x* is P(w|h). This probability depends on the words up to that point. By the chain rule of the probability, we can decompose this probability:

Pwh=Pwnw1:n-1=∏i=1nPwiw1:i-1

This allows us to calculate the conditional probability of a word from a sequence of previous words. The idea is that when we have enough text we can take a sequence such as **"to be or not to"** as input and have the model estimate the probability for the next word to be **"be,"** $P(be|tobeornotto)$. Then after the transformer block sequence, we have a layer that conducts a linear projection and a **softmax layer** that generates the output. The previous sequence is called context; the context length of the first transformers was 512 tokens. The model generates an output, which is a probability vector of dimension *V* (the model vocabulary), also called a **logit vector**. The projection layer is called an **unembedder** (it does reverse mapping) because we have to go from a dimension *N* tokens x *D* embedding to 1 x *V*. Since the input and output of each transformer block are the same, we could theoretically eliminate blocks and attach an unembedder and softmax to any intermediate block. This allows us to better interpret the function of each block and its internal representation.

Once we have this probability vector, we can use self-supervision for training. We take a corpus of text (unannotated) and train the model to minimize the difference between the probability of the true word in the sequence and the predicted probability. To do this, we use **cross-entropy loss** (the difference between the predicted and true probability distribution). The predicted probability distribution is the logit vector, while the true one is a one-hot encoder vector where it is 1 for the next word in the sequence and 0 elsewhere.

LCE=-∑w∈Vytwlogy^t[w]

In practice, it is simplified during training simply between the probability of the actual predicted word and 1. The process is iterative for each word in the word sequence (and is called teacher forcing). The final loss is the average over the entire sequence.

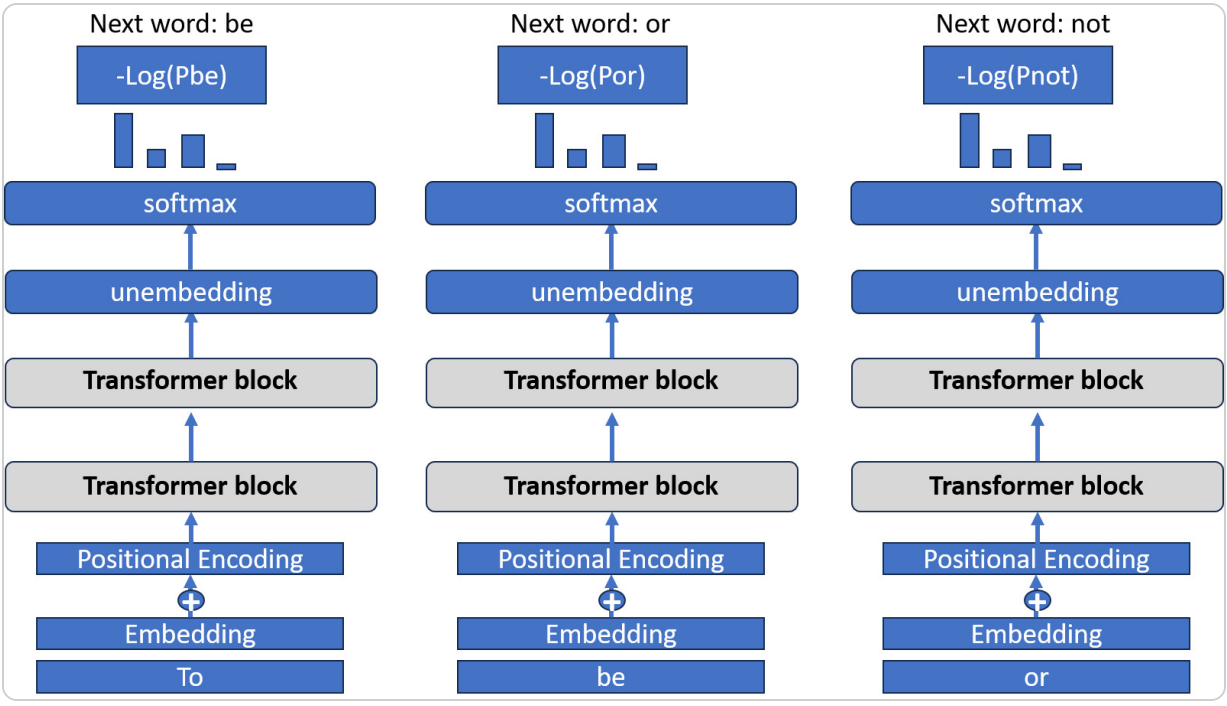

Figure 2.13 – Training of the transformer; the loss is the average of the loss of all the time steps

Since all calculations can be done in parallel in the transformer, we do not have to calculate word by word, but we fed the model with the whole sequence.

Once we have obtained a probability vector, we can choose the probability most (**greedy decoding**). Greedy decoding is formally defined as choosing the token with the highest probability at each time step:

$$wt=\text{argmax}w\in VP(w|w<t)$$

In fact, it is no longer used because the result is too predictable, generic, and repetitive. So, more sophisticated and less deterministic sampling methods are used. This sampling process is called decoding (or autoregressive generation or causal language modeling, since it is derived from previous word choice). This system, in the simplest version, is based either on generating the text of at most a predetermined sequence length, or as long as an end-of-sentence token (`<EOS>`) is selected.

We need to find a way to be able to select tokens while balancing both quality and diversity. A model that always chooses the same words will certainly have higher quality but will also be repetitive. There are different methods of doing the sampling:

- **Random sampling**: The model chooses the next token randomly. The sentences are strange because the model chooses rare or singular words.
- **Top-k sampling**: At each step, we sort the probabilities and choose the top *k* most likely words. We renormalize the probability and choose one at random.
- **Top-p sampling**: This is an alternative in which we keep only a percentage of the most likely words.
- **Temperature sampling**: Before `softmax`, we divide by a temperature parameter (between 0 and 1). The closer *t* is to 0, the closer the probability of the most likely words is to 1 (close to greedy sampling). In some cases, we can also have *t* greater than 1 when we want a less greedy approach.

So far, we have considered the fixed vocabulary and assumed that each token was a word. In general, once the model is trained, there might be some words that the model does not know to which a special token, `<UNK>`, is assigned. In transformers and LLMs afterward, a way was sought to solve the unknown word problem. For example, in the training set, we might have words such as *big*, *bigger*, and *small* but not *smaller*. *Smaller* would not be known by the model and would result in `<UNK>`. Depending on the training set, the model might have incomplete or outdated knowledge. In English, as in other languages, there are definite morphemes and grammatical rules, and we would like the tokenizer to be aware. To avoid too many `<UNK>` one solution is to think in terms of sub-words (tokens).

One of the most widely used is **Byte-Pair Encoding** (**BPE**). The process starts with a list of individual characters. The algorithm then scans the entire corpus and begins to merge the symbols that are most frequently found together. For example, we have **E** and **R**, and after the first scan, we add a new **ER** symbol to the vocabulary. The process continues iteratively to merge and create new symbols (longer and longer character strings). Typically, the algorithm stops when it has created *N* tokens (with *N* being a predetermined number at the beginning). In addition, there is a special end-of-word symbol to differentiate whether the token is inside or at the end of a word. Once the algorithm arrives at creating a vocabulary, we can segment the corpus with the tokenizer and for each subword, we assign an index corresponding to the index in the vocabulary.

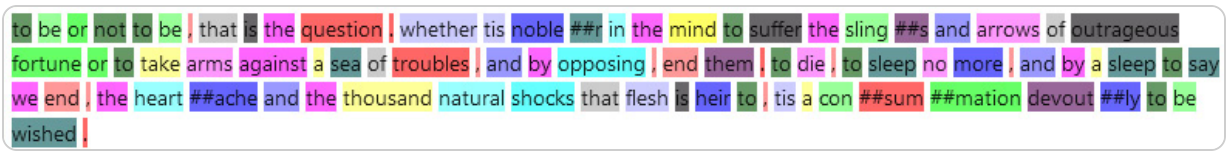

Figure 2.14 – Example of the results of tokenization

This approach generally causes common words to be present in the model vocabulary while rare words are split into subwords. In addition, the model also learns suffixes and prefixes, and considers the difference between *app* and the *app#* subword, representing a complete word and a subword (*app#* as a subword of *application*).

Exploring masked language modeling

Although the transformer was revolutionary, the popularization of the transformer in the scientific community is also due to the **Bidirectional Encoder Representations from Transformers** (**BERT**) model. This is because BERT was a revolutionary variant of the transformer that showed the capabilities of this type of model. BERT was revolutionary because it was already prospectively designed specifically for future applications (such as question answering, summarization, and machine translation). In fact, the original transformer analyzes the left-to-right sequence, so when the model encounters an entity, it cannot relate it to what is on the right of the entity. In these applications, it is important to have context from both directions.

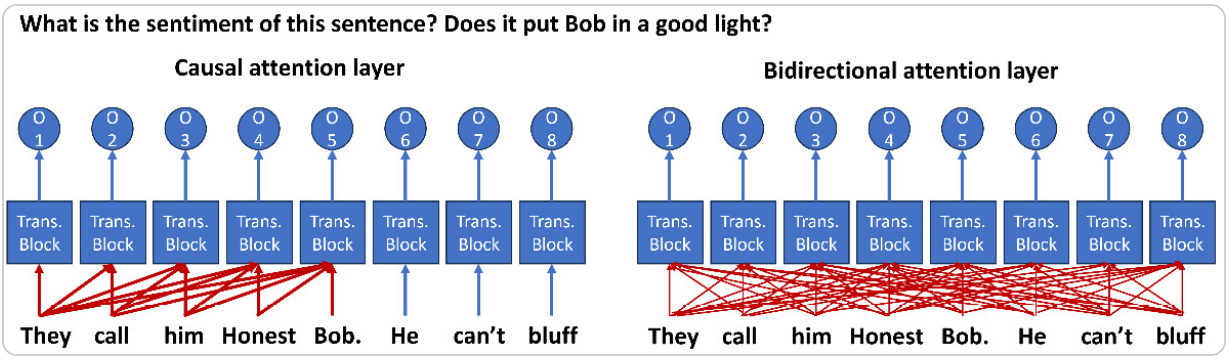

Figure 2.15 – Difference between a causal and bidirectional language model

Bidirectional encoders resolve this limitation by allowing the model to find relationships over the entire sequence. Obviously, we can no longer use a language model to train it (it will be too easy to identify the next word in the sequence when you already know the answer) but we have to find a way to be able to train a bidirectional model. For clarification, the model reads the entire sequence at once and, in this case, consists of the encoder only.

To try to minimize changes to the structure we use what is called the **Masked Language Model** (**MLM**) **objective**. Instead of predicting the next word in the sequence, we mask some tokens and the model has to use the rest of the sequence to predict them. Given the entire context (the model now has access to the entire sequence), BERT must predict the token that has been masked with a special token (usually called `<MASK>`). In the original training, they masked 15 % of the tokens randomly. Notice that, in this case, we do not mask the future because we want the model to be aware of the whole context. Also, to better separate the different sentences, we have a special token, `[CLS]`, that signals the beginning of an input, and `[SEP]` to separate sentences in the input (for example, if we have a question and an answer). Otherwise, the structure is the same: we have an embedder, a position encoder, different transformer blocks, a linear projection, and a softmax. The loss is calculated in the same way; instead of using the next token, we use the masked token. The original article introduced two versions

of BERT: BERT-BASE (12 layers, hidden size with d=768, 12 attention heads, and 110M total parameters) and BERT-LARGE (24 layers, hidden size with d=1024, 24 attention heads, and 340M total parameters).

MLM is a flexible approach because the idea is to corrupt the input and ask the model to rebuild. We can mask, but we can also reorder or conduct other transformations. The disadvantage of this method is that only 15 percent of the tokens are actually used to learn, so the model is highly inefficient.

The training is also highly flexible. For example, the model can be extended to **next sentence prediction**, where the task is to predict whether two pairs of sentences are related (paraphrase, coherence, and entailment). In this case, BERT was trained with training pairs of sentences positively related and unrelated sentences (we exploit the `[SEP]` token between sentences). The last layer is a softmax for sentence classification; we consider the loss over the categories. This shows how the system is flexible and can be adapted to different tasks.

One final clarification. Until 2024, it was always assumed that these models were not capable of generating text. In 2024, two studies showed that by adapting the model, you can generate text even with a BERT-like model. For example, in this study, they show that one can generate text by exploiting a sequence of [MASK] tokens.

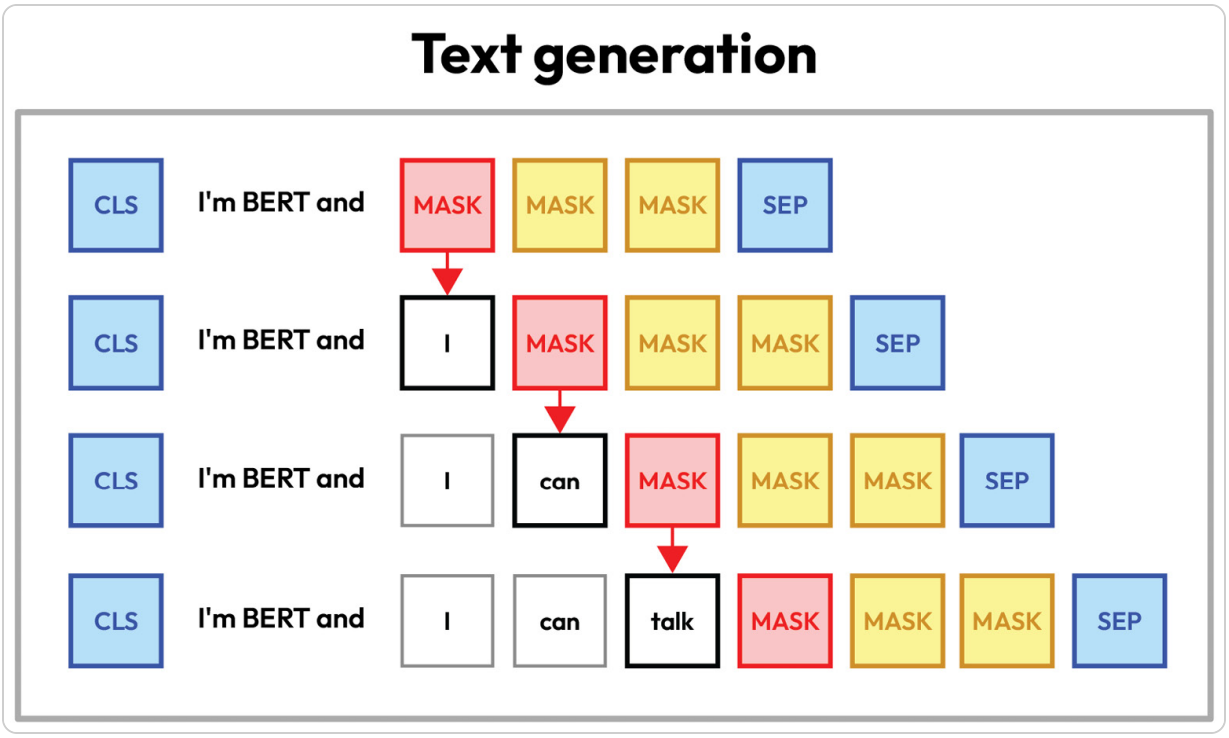

Figure 2.16 – Text generation with MLM (https://arxiv.org/pdf/2406.04823)

Now that we have seen the two main types of training for a transformer, we can better explore what happens inside these models.

Visualizing internal mechanisms

We have seen the inner workings of the transformer, how it can be trained, and the main types of models. The beauty of attention is that we can visualize these relationships, and in this section, we will see how to do that. We can then visualize the relationships within the BERT attention head. As mentioned, in each layer, there are several attention heads and each of them learns a different representation of the input data. The color intensity indicates a greater weight in the attention weights (darker colors indicate weights that are close to 1).

We can do this using the BERTviz package:

```
head_view(attention, tokens, sentence_b_start)
```

> ***IMPORTANT NOTE***
>
> *The visualization is interactive. The code is in the repository. Try running it using different phrases and exploring different relationships between different words in the phrases. The visualization allows you to explore the different layers in the model by taking advantage of the drop-down model. Hovering over the various words allows you to see the individual weights of the various heads.*

This is the corresponding visualization:

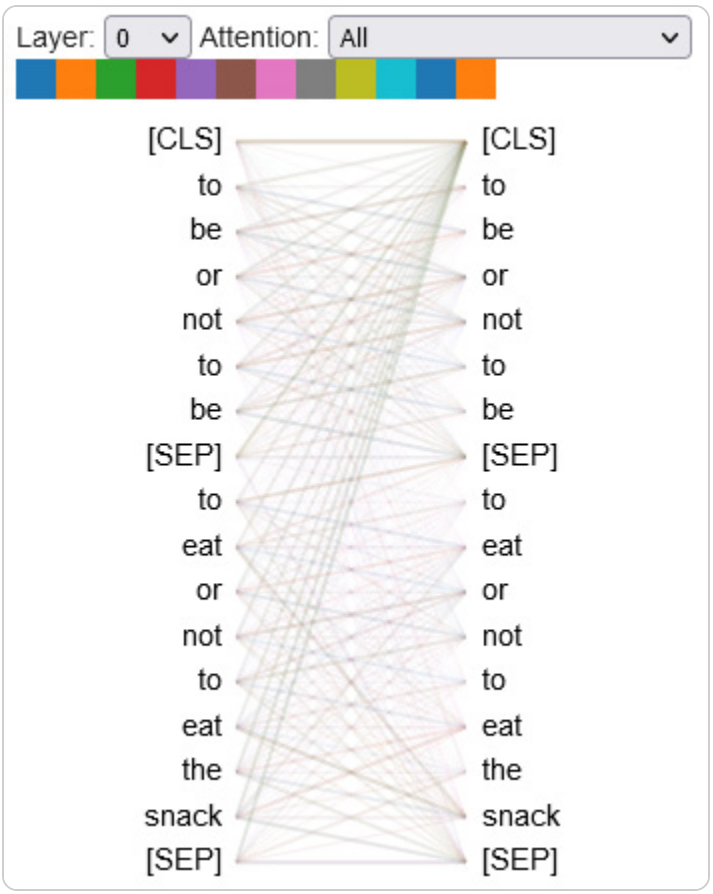

Figure 2.17 – Visualization of attention between all words in the input

We can also view the various heads of the model at the same time. This allows us to see how the various heads model different relationships. This model has 12 heads for 12 layers, so the model has 144 attention heads and can therefore see more than 100 representations for the same sentences (this explains the capacity of a model). Moreover, these representations are not completely independent; information learned from earlier layers can be used by later layers:

```
model_view(attention, tokens, sentence_b_start)
```

> ***IMPORTANT NOTE***
>
> *The visualization is interactive. The code is in the repository. Try running it using different phrases and exploring different relationships. Here, we have the ensemble representation of the various attention heads. Observe how each head has a different function and how it models a different representation of the same inputs.*

This is the corresponding visualization:

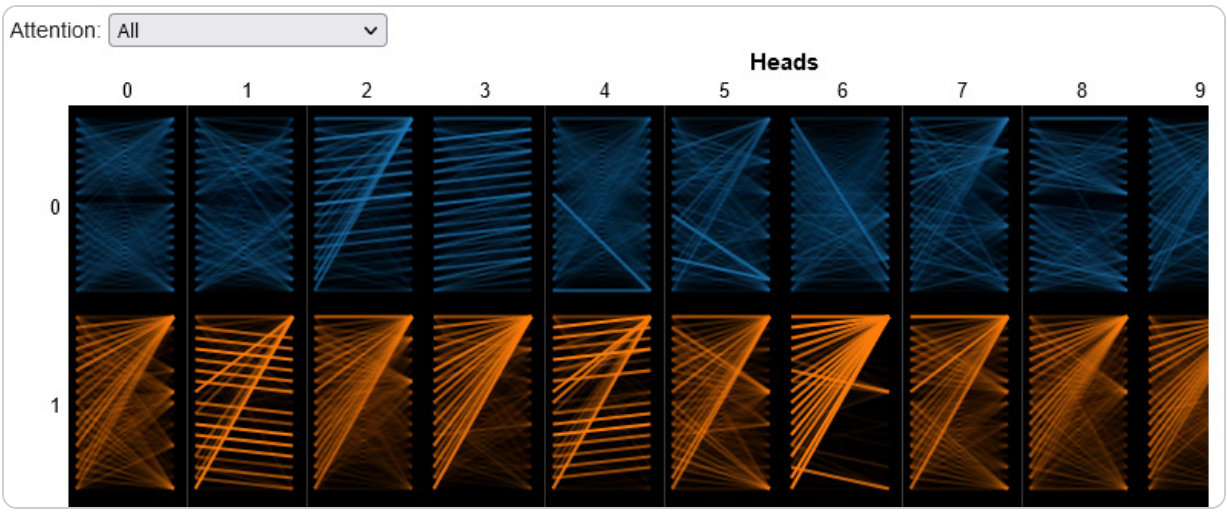

Figure 2.18 – Model view of the first two layers

Another model that has been fundamental to the current development of today's models is **Generative Pre-Trained Transformer 2** (**GPT-2**). GPT-2 is a causal (unidirectional) transformer pre-trained using language modeling on a very large corpus of ~40 GB of text data. GPT-2 was specifically trailed to predict the next token and to generate text with an input (it generates a token at a time; this token is then added to the input sequence to generate the next in an autoregressive process). In addition, this is perhaps the first model that has been trained with a massive amount of text. In addition, this model consists only of the decoder. GPT-2 is a family of models ranging from 12 layers of GPT-2 small to 48 layers of GPT-2 XL. Each layer consists of masked self-attention and a feed-forward neural network.

GPT-2 is generative and trained as a language model so we can give it an input judgment and observe the probability for the next token. For example, using "To be or not to" as input, the token with the highest probability is "be."

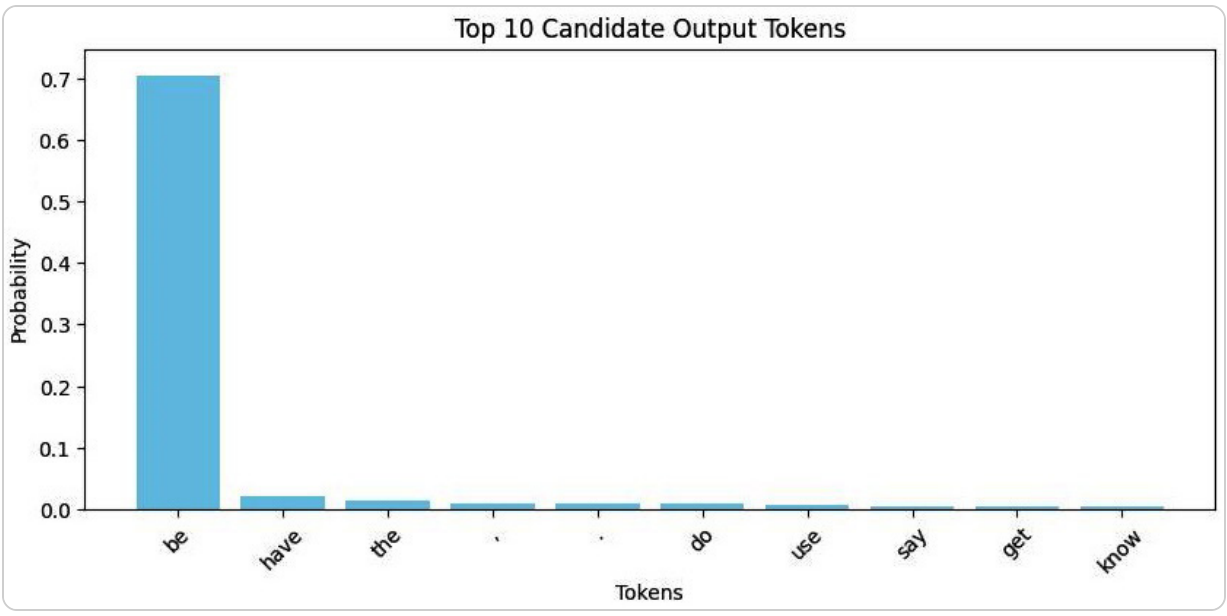

Figure 2.19 – Probabilities associated with the next token for the GPT-2 model when probed with the "To be or not to" input sequence

Sometimes, it may be necessary to understand which tokens are most important to the model to generate the next token. **Gradient X input** is a technique originally developed for convolutional networks; at a given time step, we take the output probabilities for each token, select the tokens with the highest probability, and compute the gradient with respect to the input up to the input tokens. This gives us the importance of each token to generate the next token in the sequence (the rationale is that small changes in the input tokens with the highest importance carry the largest changes in the output). In the figure, we can see the most important tokens for the next token in the sequence:

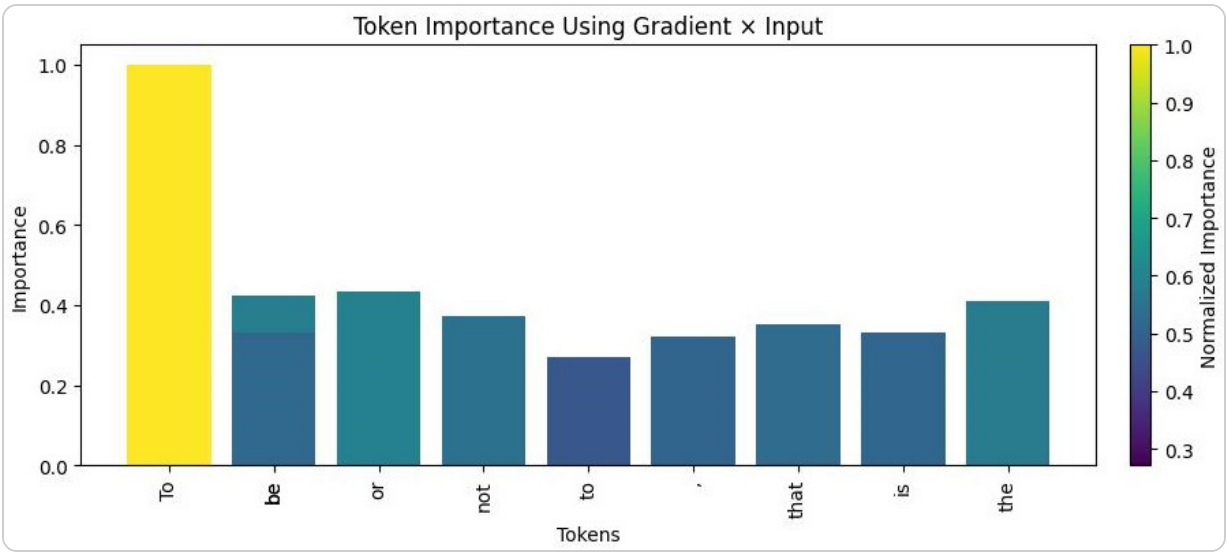

Figure 2.20 – Gradient X input for the next token in the sequence

As mentioned before, there is not only self-attention but also feedforward neural network, which plays an important role (it provides a significant portion of the parameters in the transformer block, about 66%). Therefore, several works have focused on examining the firings of neurons in layers (this technique was also originally developed for computer vision).

We can follow this activation after each layer, and for each of the tokens, we can monitor what their rank (by probability) is after each layer. As we can see, the model understands from the first layers which token is the most likely to continue a sequence:

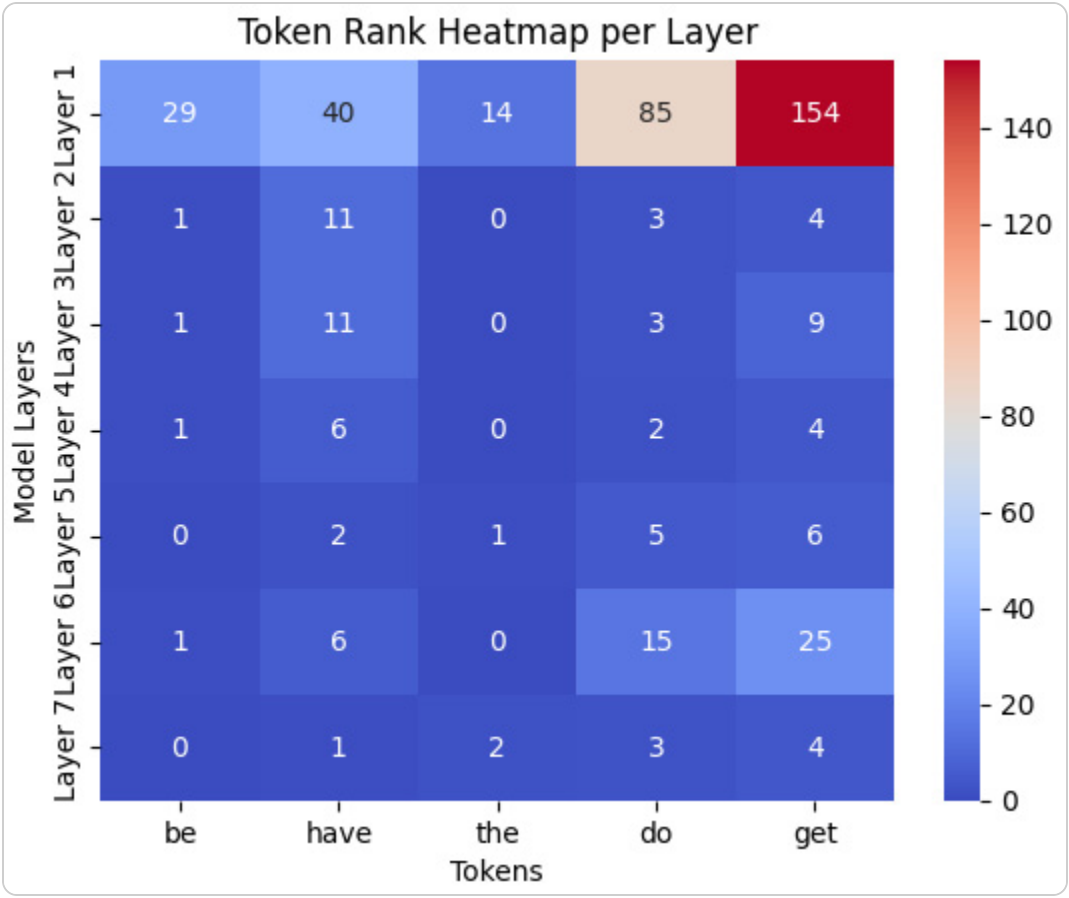

Figure 2.21 – Heatmap of the rank for the top five most likely tokens after each layer

Since there are a considerable number of neurons, it is complex to be able to observe them directly. Therefore, one way to investigate these activations is to first reduce dimensionality. To avoid negative activations, it is preferred to use **Non-Negative Matrix Factorization** (**NMF**) instead of **Principal Component Analysis** (**PCA**). The process first captures the activation of neurons in the FFNN layers of the model and is then decomposed into some factors (user-chosen parameters). Next, we can interactively observe the factors with the highest activation when a token has been generated. What we see in the graph is the factor excitation for each of the generated tokens:

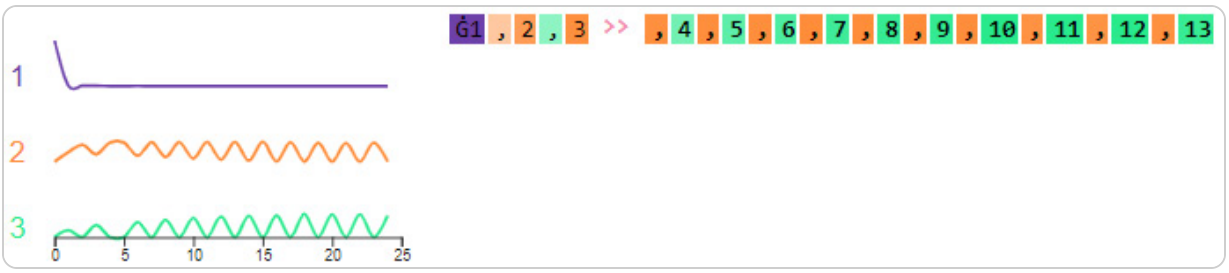

Figure 2.22 – NMF for the activations of the model in generating a sequence

We can also conduct this analysis for a single layer. This allows us to analyze interesting behaviors within the neurons of a layer (in the image layer 0 of the model). In this case, there are certain factors that focus on specific portions of the text (beginning, middle, and end). As we mentioned earlier, the model keeps track of word order in a sequence due to positional encoding, and this is reflected in activation. Other neurons, however, are activated by grammatical structures (such as conjunctions, articles, and so on). This indicates to us a specialization of what individual neurons in a pattern track and is one of the strength components of the transformer. By increasing the number of facts, we can increase the resolution and better understand what grammatical and semantic structures the pattern encodes in its activations. Moving forward in the structure of the model, we can see that layers learn a different representation.

Figure 2.23 – NMF for the activations of the model in generating a sequence

We have seen how to build a transformer and how it works. Now that we know the anatomy of a transformer, it is time to see it at work.

Applying a transformer

The power of a transformer lies in its ability to be able to learn from an enormous amount of text. During this phase of training (called **pre-training**), the model learns general rules about the structure of a language. This general representation can then be exploited for a myriad of applications. One of the most important concepts in deep learning is **transfer learning**, in which we exploit the ability of a

model trained on a large amount of data for a task different from the one it was originally trained for. A special case of transfer learning is **fine-tuning**. Fine-tuning allows us to adapt the general knowledge of a model to a particular case. One way to do this is to add a set of parameters to a model (at the top of it) and then train these parameters by gradient descent for a specific task.

The transformer has been trained with large amounts of text and has learned semantic rules that are useful in understanding a text. We want to exploit this knowledge for a specific application such as sentiment classification. Instead of training a model from scratch, we can adapt a pre-trained transformer to classify our sentences. In this case, we do not want to destroy the internal representation of the model but preserve it. That is why, during fine-tuning, most of the layers are frozen (there is no update on the weights). Instead, we just train those one or two layers that we add to the top of the model. The idea is to preserve the representation and then learn how to use it for our specific task. Those two added layers learn precisely how to use the internal representation of the model. To give a simple example, let's imagine we want to learn how to write scientific papers. To do that, we don't have to learn how to write in English again, just to adapt our knowledge to this new task.

In BERT, as we mentioned, we add a particular token to the beginning of each sequence: a `[CLS]` token. During training or even inference in a bidirectional transformer, this token waits for all others in the sequence (if you remember, all tokens are connected). This means that the final vector (the one in the last layer) is contextualized for each element in the sequence. We can then exploit this vector for a classification task. If we have three classes (for example, positive, neutral, and negative) we can take the vector for a sequence and use softmax to classify.

$$y=\text{softmax}W_h\text{CLS}$$

The model was not originally trained for sequence classification, so we'd like to introduce a learnable matrix to enable class separation. This matrix represents a linear transformation and can alternatively be implemented using one or more linear layers. We then apply a cross-entropy loss function to optimize these weights. This setup follows the standard supervised learning paradigm, where labeled data is used to adapt the transformer to a specific task.

In this process, we have so far assumed that the remainder of the transformer's weights remain frozen. However, as observed in convolutional neural networks, even minimal fine-tuning of model parameters can enhance performance. Such updates are typically carried out with a very low learning rate.

We can adapt a pre-trained transformer for new tasks through supervised fine-tuning.

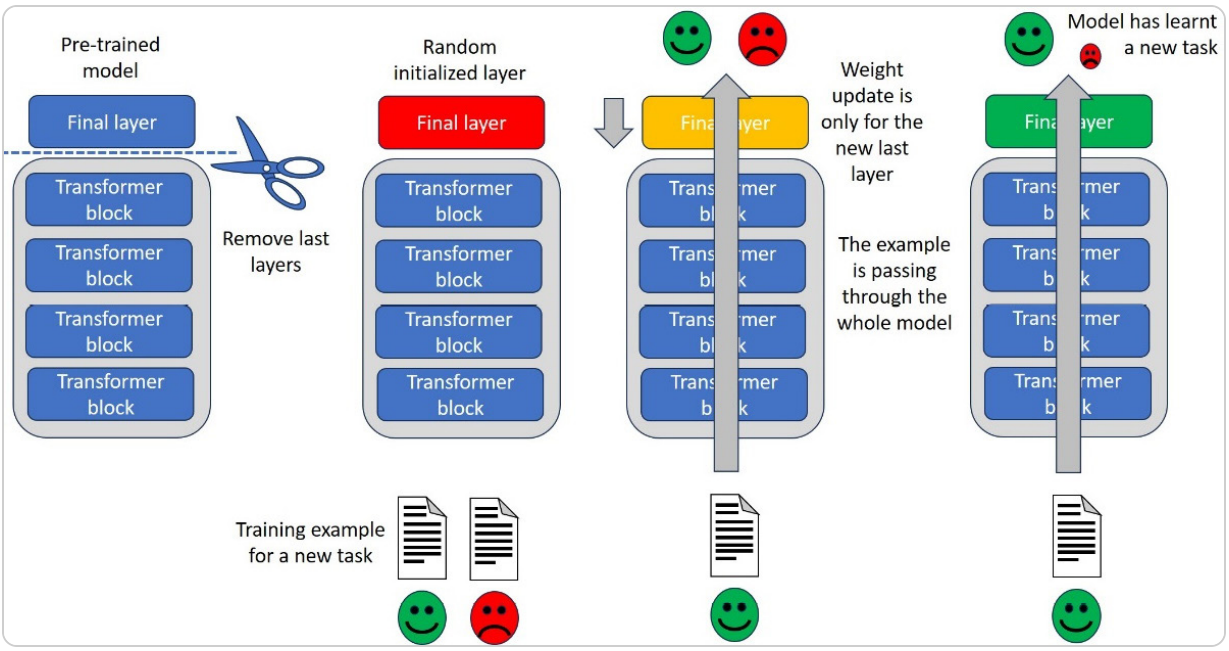

Figure 2.24 – Fine-tuning a transformer

In the first step, we are removing the final layer (this is specific to the original task). In the second step, we add a random initialized layer and gather training examples for the new task. During the fine-tuning, we are presenting the model with new examples (in this case, positive and negative reviews). While keeping the model frozen (each example is processed by the whole model in the forward pass), we update the weight only in the new layer (through backpropagation). The model has now learned the new task.

Conducting finetuning with Hugging Face is a straightforward process. We can use a model such as distill-BERT (a distilled version of BERT) with a few lines of code and the dataset we used in the previous chapter. We need to prepare the dataset and tokenize it (so that it can be used with a transformer). Hugging Face then allows with a simple wrapper that we can train the model. The arguments for training are stored in **`TrainingArguments`**:

```
training_args = TrainingArguments(
    output_dir='./results',
    num_train_epochs=3,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=16
    warmup_steps=500,
    weight_decay=0.01,
    logging_dir='./logs',
    logging_steps=10,
    evaluation_strategy="epoch"
)
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
```

```
    eval_dataset=val_dataset
)
trainer.train()
```

> ***IMPORTANT NOTE***
>
> *Notice that the process is very similar to training a neural network. In fact, the transformer is a deep learning model; for the training, we are using similar hyperparameters.*

In this case, we used only a small fraction of the reviews. The beauty of fine-tuning is that we need only a few examples to have a similar (if not better) performance than a model trained from scratch.

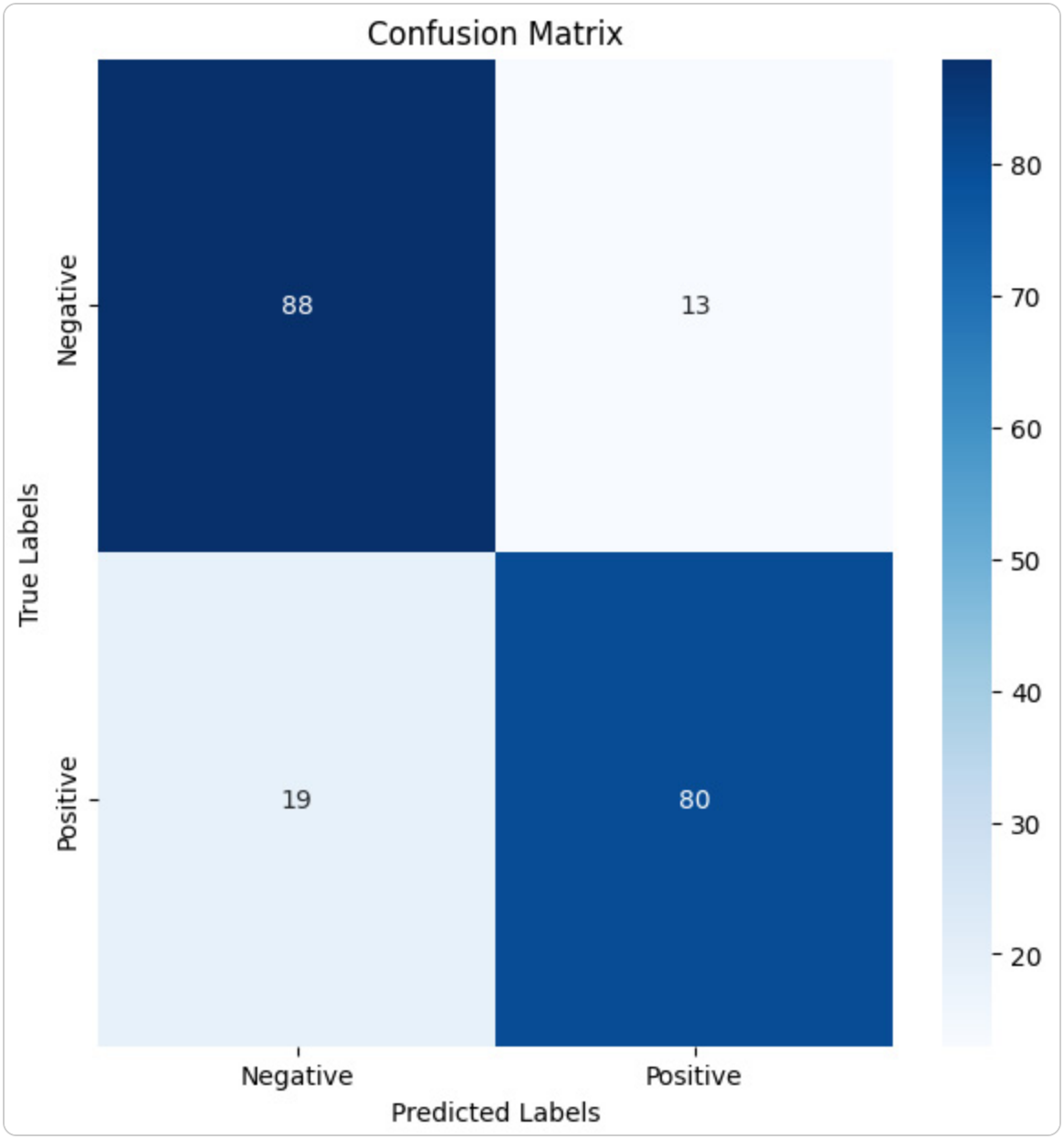

Figure 2.25 – Confusion matrix after fine-tuning

BERT's training was done on 64 TPUs (special hardware for tensor operations) for four days; this is beyond the reach of most users. In contrast, fine-tuning is possible either on a single GPU or on a CPU. As a result, BERT achieved state-of-the-art performance upon its release across a wide array of tasks, including paraphrase detection, question answering, and sentiment analysis. Hence, several variants such as **RoBERTa** and **SpanBERT** (in this case, we mask an entire span instead of a single token with better results) or adapted to specific domains such as **SciBERT** were born. However, encoders are not optimal for generative tasks (because of mask training) while decoders are.

To conduct machine translation, the original transformer consisted of an encoder and a decoder. A model such as GPT-2 only has a decoder. We can conduct fine-tuning in the same way as seen before, we just need to construct the dataset in an optimal way for a model that is constituted by the decoder alone. For example, we can take a dataset in which we have English and French sentences and build a dataset for finetuning as follows: `<sentence in English>` followed by a special `<to-fr>` token and then the `<sentence in French>`. The same approach can be used to teach summarization to a model, where we insert a special token meaning summarization. The model is fine-tuned by conducting the next token prediction (language modeling).

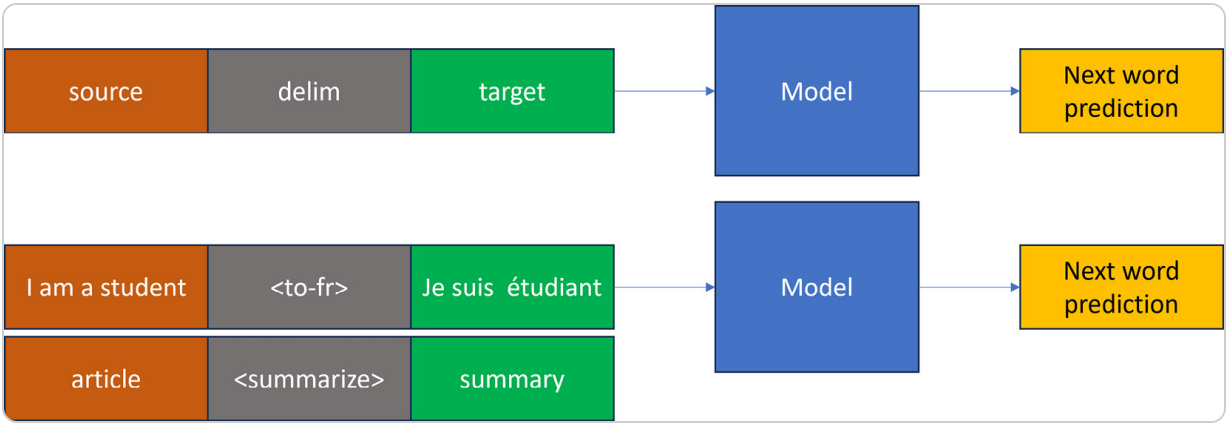

Figure 2.26 – Fine-tuning of a decoder-only model

Another way to exploit the learned knowledge of a model is to use **knowledge distillation**. In the previous section, we used distillGPT-2 which is a distilled version of GPT-2. A distilled model captures knowledge from a much larger model without losing significant performance but is much more manageable. Models that are trained with a large amount of text learn a huge body of knowledge. All this knowledge and skill is often redundant when we need a model for some specific task. We are interested in having a model that is very capable for a task, but without wanting to deal with a model of billions of parameters. In addition, sometimes we do not have enough examples for a model to learn the task from scratch. In this case, we can extract knowledge from the larger model.

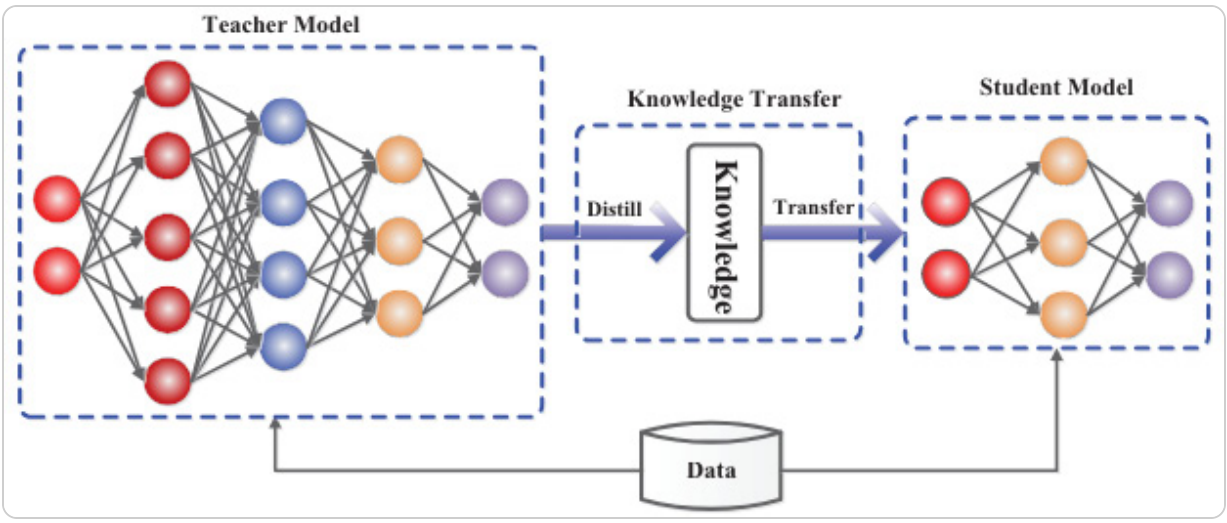

Figure 2.27 – Generic teacher-student framework for knowledge distillation (https://arxiv.org/pdf/2006.05525)

Knowledge distillation can be seen as a form of compression, in which we try to transfer knowledge from a trained "teacher" model with many parameters to a "student" model with fewer parameters. The student model tries to mimic the teacher model and achieve the same performances as the teacher model in a task. In such a framework, we have three components: the models, knowledge, and algorithm. The algorithm can exploit either the teacher's logits or intermediate activations. In the case of the logits, the student tries to mimic the predictions of the teacher model, so we try to minimize the difference between the logits produced by the teacher and the student. To do this, we use a distillation loss that allows us to train the student model.

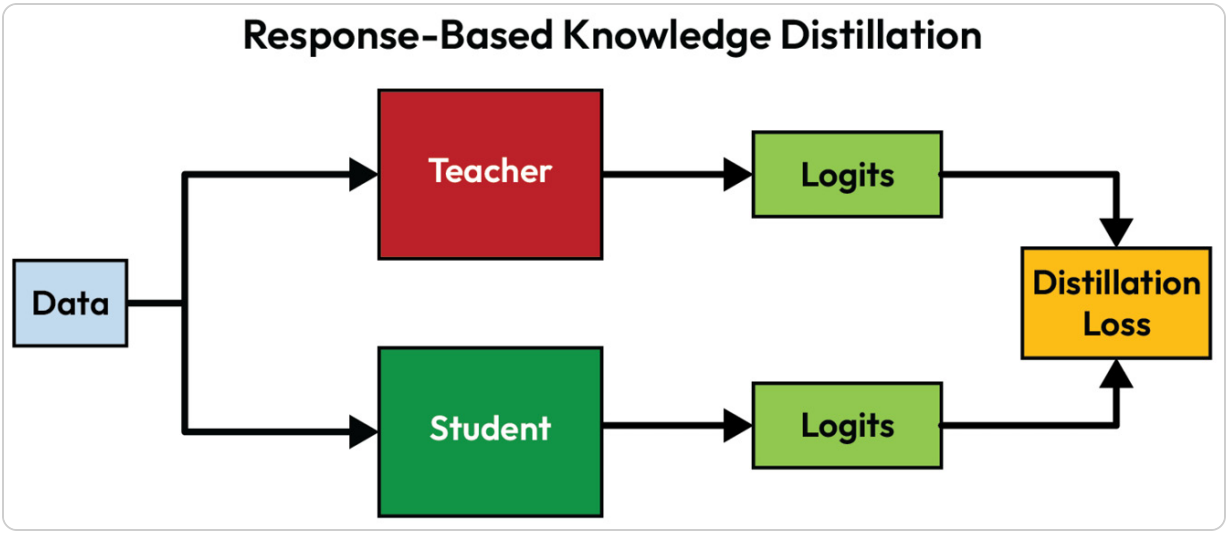

Figure 2.28 – Teacher-student framework for knowledge distillation training (https://arxiv.org/pdf/2006.05525)

For knowledge distillation, the steps are also similar. The first step is data preprocessing. For each model, you must remember to choose the model-specific tokenizer (although the one from GPT-2 is the most widely used many models have different tokenizers). We must then conduct fine-tuning of a model on our task (there is no model that is specific to classify reviews). This model will be our teacher. The next step is to train a student model. We can also use a pre-trained model that is smaller than the teacher (this allows us to be able to use a few examples to train it).

One important difference is that we now have a specific loss for knowledge distillation. This distillation loss calculates the loss between the teacher's logits and the student's logits. This function typically uses the Kullback-Leibler divergence loss to calculate the difference between the two probability distributions (Kullback-Leibler divergence is really a measure of the difference between two probability distributions). We can define it as follows:

```
def distillation_loss(outputs_student, outputs_teacher,
                      temperature=2.0):
    log_prob_student = F.log_softmax(
        outputs_student / temperature, dim=-1)
    prob_teacher = F.softmax(
        outputs_teacher / temperature, dim=-1)
    loss = KLDivLoss(reduction='batchmean')(
        log_prob_student, prob_teacher)
    return loss
```

At this point, we just have to have a way to train our system. In this case, the teacher will be used only in inference while the student model will be trained. We will use the teacher's logits to calculate the loss:

```
def train_epoch(model, dataloader, optimizer, device,
                teacher_model, temperature=2.0):
    model.train()
    total_loss = 0
    for batch in tqdm(dataloader, desc="Training"):
        inputs = {k: v.to(device)
                  for k, v in batch.items()
                  if k in ['input_ids', 'attention_mask']}
        with torch.no_grad():
            outputs_teacher = teacher_model(**inputs).logits
        outputs_student = model(**inputs).logits
        loss = distillation_loss(
            outputs_student, outputs_teacher, temperature)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    return total_loss / len(dataloader)
```

As can be seen in the following figure, the performance of the student model is similar to the teacher model:

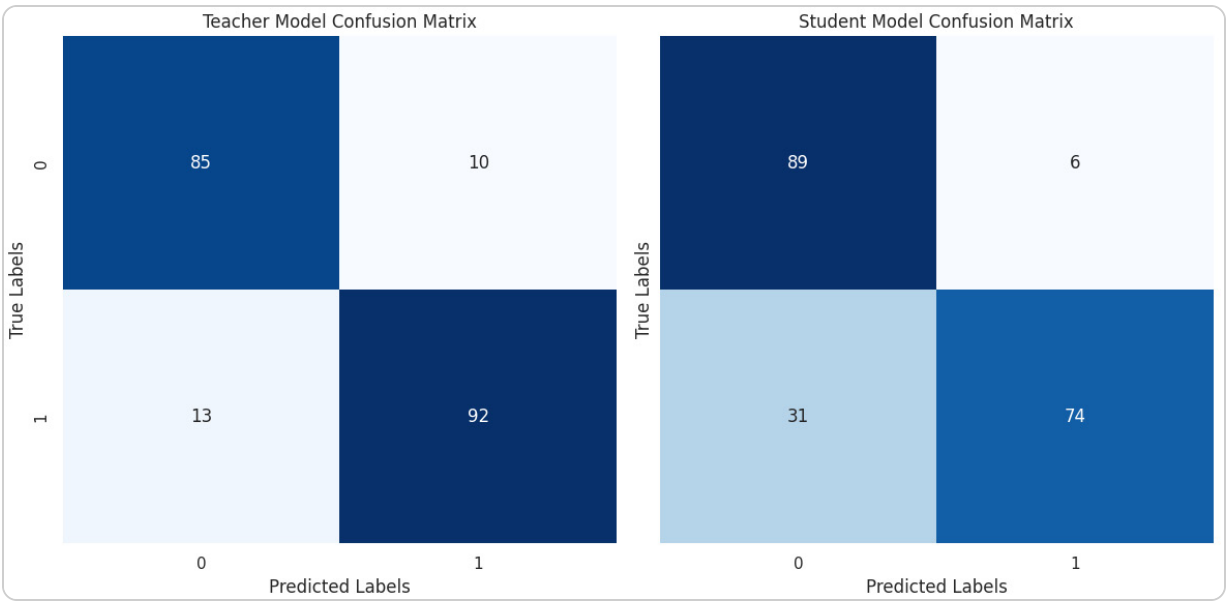

Figure 2.29 – Confusion matrix for the teacher and student model

Fine-tuning and knowledge distillation allow us to be able to use a transformer for any supervised task. Fine-tuning allows us to work with datasets that are small (and where there are often too few examples to train a model from scratch). Knowledge distillation, on the other hand, allows us to get a smaller model (but performs as well as a much larger one) when the computational cost is the limit. By taking advantage of these techniques, we can tackle any task.

Summary

In this chapter, we discussed the transformer, the model that revolutionized NLP and artificial intelligence. Today, all models that have commercial applications are derivatives of the transformer, as we learned in this chapter. Understanding how it works on a mechanistic level, and how the various parts (self-attention, embedding, tokenization, and so on) work together, allows us to understand the limitations of modern models. We saw how it works internally in a visual way, thus exploring the motive of modern artificial intelligence from multiple perspectives. Finally, we saw how we can adapt a transformer to our needs using techniques that leverage prior knowledge of the model. Now we can repurpose this process with virtually any dataset and any task.

Learning how to train a transformer will allow us to understand what happens when we take this process to scale. An LLM is a transformer with more parameters and that has been trained with more text. This leads to emergent properties that have made it so successful, but both its merits and shortcomings lie in the elements we have seen.

In *Chapter 3*, we will see precisely how to obtain an LLM from a transformer. What we have learned in this chapter will allow us to see how this step comes naturally.

3
Exploring LLMs as a Powerful AI Engine

In the previous chapter, we saw the structure of a transformer, how it is trained, and what makes it so powerful. The transformer is the seed of this revolution in **natural language processing** (**NLP**), and today's **large language models** (**LLMs**) are all based on transformers trained at scale. In this chapter, we will see what happens when we train huge transformers (more than 100 billion parameters) with giant datasets. We will focus on how to enable this training at scale, how to fine-tune similar modern ones, how to get more manageable models, and how to extend them to multimodal data. At the same time, we will also see what the limitations of these models are and what techniques are used to try to overcome these limitations.

In this chapter, we'll be covering the following topics:

- Discovering the evolution of LLMs
- Instruction tuning, fine-tuning, and alignment
- Exploring smaller and more efficient LLMs
- Exploring multimodal models
- Understanding hallucinations and ethical and legal issues
- Prompt engineering

Technical requirements

Most of this code can be run on a CPU, but it is preferable to run it on a GPU. The code is written in PyTorch and uses standard libraries for the most part (PyTorch, Hugging Face Transformers, and so on). The code can be found on GitHub: https://github.com/PacktPublishing/Modern-AI-Agents/tree/main/chr3.

Discovering the evolution of LLMs

An LLM is a transformer (although different architectures are beginning to emerge today). In general, an LLM is defined as a model that has more than 10 billion parameters. Although this number may seem arbitrary, some properties emerge with scale. These models are designed to understand and generate human language, and over time, they have acquired the ability to generate code and more. To achieve this beyond parameter size, they are trained with a huge amount of data. Today's LLMs are almost all trained on **next-word prediction** (**autoregressive language modeling**).

Parameter growth has been motivated in the transformer field by different aspects:

- **Learnability**: According to the scaling law, more parameters should lead to greater capabilities and a greater understanding of nuances and complexities in the data
- **Expressiveness**: The model can express more complex functions, thus increasing the ability to generalize and reducing the risk of overfitting
- **Memory**: A larger number of parameters allows for internalizing more knowledge (information, entities, differences in topics)

In the next subsections, we will discuss in detail all these elements to explain what is happening in the transition from the transformer to the LLM.

The scaling law

It may seem surprising that such large models are trained with such a simple task as **language modeling**. Many practical **natural language** tasks can be represented as next-word prediction. This flexibility allows us to use LLMs in different contexts. For example, sentiment analysis can be cast as a next-word prediction. The sentence "*The sentiment of the sentence: 'I like Pizza' is*" can be used as input for an LLM, and we can extract the probability for the next token being *positive* or *negative*. We can then assign the sentiment depending on which of the two has the higher probability. Notice how this probability is a function of context:

P(positive| *The sentiment of the sentence: 'I like Pizza' is*)

P(negative| *The sentiment of the sentence: 'I like Pizza' is*)

Similarly, we can use the same approach for other tasks. **Question answering** (**QA**) with an LLM can be thought of as generating the probability of the right answer given the question. In text summarization, we want to generate given the original context:

QA: P(answer| question)

Text summarization: P(summary|original article)

In the following diagram, we can see that using language modeling, we can solve almost any task. For example, here, the answer is the most probable token given the previous sequence (the question):

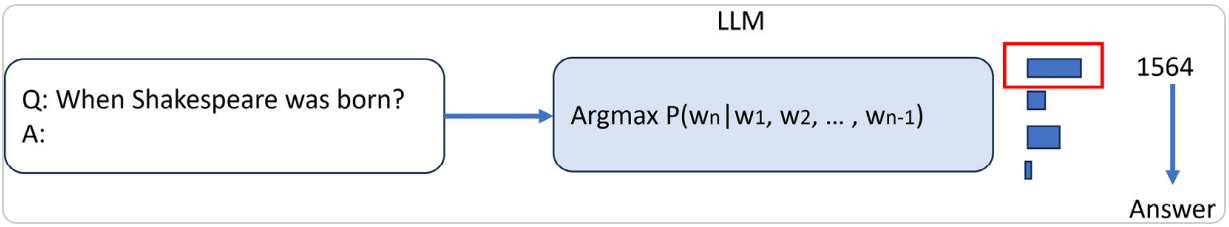

Figure 3.1 – Rephrasing of any task as LM

What we need is a dataset large enough for the model to both learn knowledge and use that knowledge for tasks. For this, specific datasets are assembled for training an LLM. These datasets typically consist of billions of words obtained from various sources (internet, books, articles, GitHub, different languages, and so on). For example, **GPT-3** was trained with Common Crawl (web crawl data, 410 billion tokens), Books1 and Books2 (book corpora, 12 billion and 55 billion tokens, respectively), and Wikipedia (3 billion tokens). Such diversity provides specific knowledge but also examples of tasks.

In parallel with the growth of training datasets (today, we are talking about more than a trillion tokens), the number of parameters has grown. The number of parameters in a transformer depends on three factors:

- **Embedding layer**: The number of parameters on the size of the vector and the vocabulary (which, especially for multi-language models, can be very large). Attention mechanisms are the heaviest component and hold the most parameters.
- **Self-attention mechanism**: This component includes multiple weight matrices that can grow in size with context length. Also, there can be multiple heads per single self-attention.
- **Depth**: Transformers are composed of multiple transformer blocks, and increasing the number of these blocks directly adds more parameters to the model.

GPT-3 and other studies have shown that the performance of LLMs depends mainly on three factors: model size (number of parameters), data size (the size of the training dataset), and computing size (amount of computing). So, in theory, to increase the performance of our model, we should enlarge the model (add layers or attention heads), increase the size of the pre-training dataset, and train it for more epochs. These factors have been related by OpenAI with the so-called **scaling law**. From a model with a number of parameters N, a dataset D, and computing amount C, if two parameters are constant, the loss L is the following:

$$L(N) = \left(\frac{N_c}{N}\right)^{\alpha_N} L(D) = \left(\frac{D_c}{D}\right)^{\alpha_D} L(C) = \left(\frac{C_c}{C}\right)^{\alpha_C}$$

This is represented visually in the following diagram:

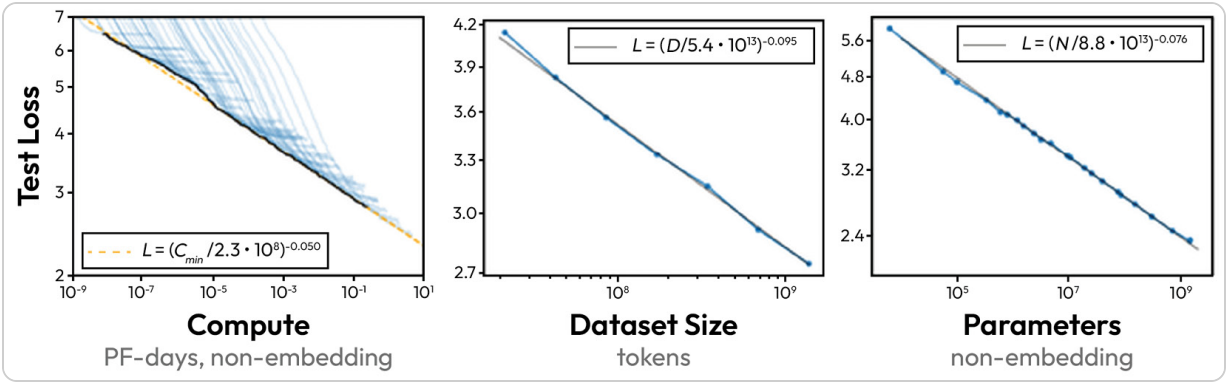

Figure 3.2 – Language modeling performance improves smoothly with the increase of model size, dataset size, and amount of computing (https://arxiv.org/pdf/2001.08361)

The loss is, in this case, the cross-entropy loss. In successive studies, OpenAI has shown that this loss can be decomposed into **irreducible loss** (which cannot be eliminated because it is related to data entropy) and reducible loss. This scaling law, in other words, allows us to calculate the desired performance of the model before training it. We can decide whether to invest more in enlarging the model or the dataset to reduce the loss (improve performance). However, these constants are dependent on the architecture and other training choices:

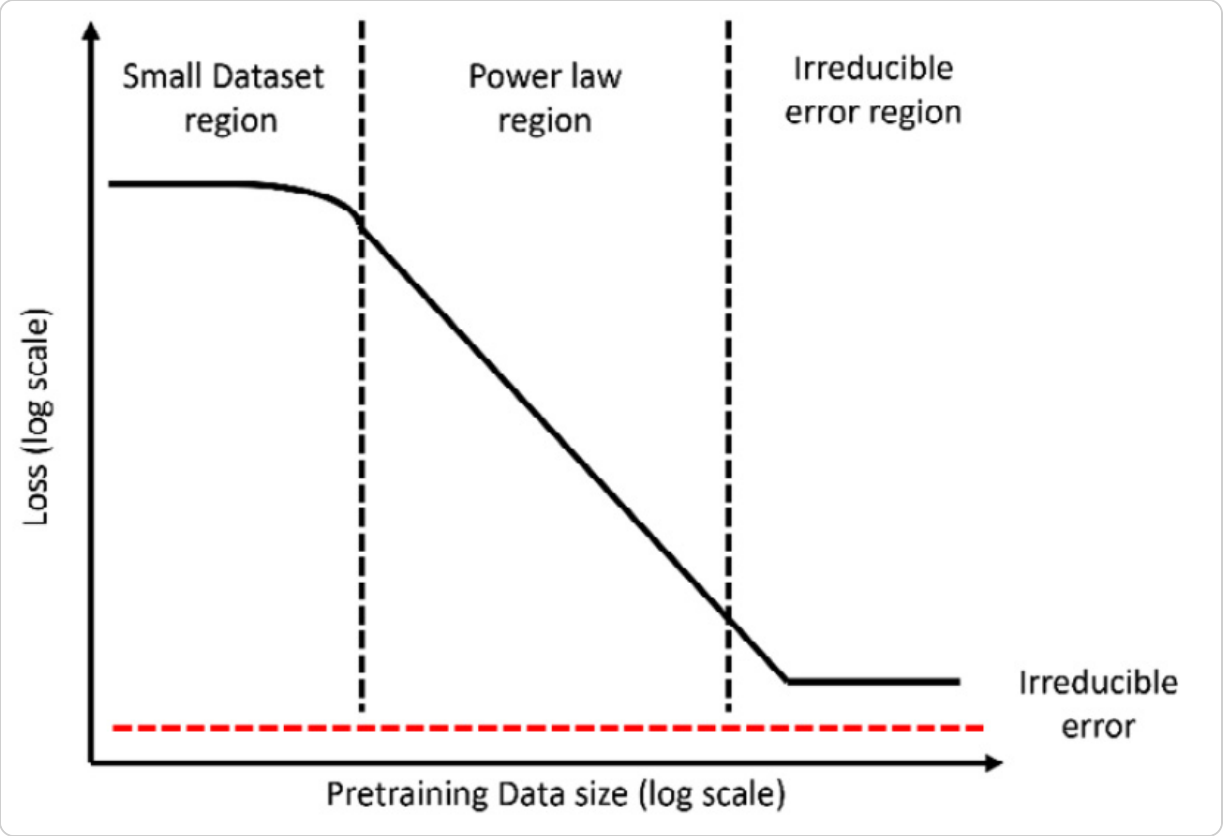

Figure 3.3 – Scaling law for an LLM

Although this scaling law has been taken for granted, the reality is more nuanced than it seems. According to DeepMind's Chinchilla (https://arxiv.org/abs/2203.15556), performance depends much more on the number of tokens than OpenAI believes. So, LLMs would currently be underfitted because they are trained with fewer tokens than expected. Meta's Llama also states that not just any tokens will do, but they must be of quality. So, not all tokens count the same, and according to other authors, using tokens produced by other models is just a more sophisticated form of distillation. In other words, to train a model at its best, you need a large amount of tokens, and they should be