Alma Mater Studiorum · Università di Bologna

SCHOOL OF SCIENCE
Laurea Magistrale in Matematica
Curriculum Advanced Mathematics for Applications

MATRIX FACTORIZATION TECHNIQUES FOR LARGE LANGUAGE MODELS

Supervising Professor:
Chiar.mo Prof.
DAVIDE PALITTA

Co-Supervisor:
Chiar.ma Prof.ssa
MARGHERITA PORCELLI

Co-Supervisor:
Dott.ssa
DOMITILLA BRANDONI

Presented by:
SIMONE PANDINI

V Session
Academic Year 2023/2024

Abstract

In the last years the development of Large Language Models (LLMs) has revolutionized the field of natural language processing (NLP), enabling significant advancements in several contexts, such as text translation, code generation and question answering. To address all these tasks, LLMs have become increasingly complex and resource-intensive, since they require an extensive training on a huge amount of data, mostly in English, that need to be preprocessed. Since the most famous LLMs are meant to be general purpose, a fine-tuning procedure is usually needed to tailor the models on specific domains. However, updating all parameters, would be computationally expensive and would require significant memory resources. This brought to the exploration of parameter-efficient fine-tuning (PEFT) methods that allow to modify only a small subset of parameters while keeping the majority fixed. This approach not only reduces the computational effort but also minimizes the risk of catastrophic forgetting, particularly when working with limited task-specific data. Additionally, compared to training from scratch, fine-tuning may be achieved with fewer labeled instances and less computing resources by utilizing the knowledge already present in these huge models. This method improves the effectiveness of implementing LLMs in practical applications while also democratizing access to cutting-edge AI capabilities. All the most important PEFT methods rely on different types of matrix factorization, such as low-rank, sparse or Singular Value Decomposition in order to decompose the trainable fine-tuning matrix. These factorizations contain fewer parameters than full fine-tuning, allowing to significantly decrease training time and computational resources without affecting the overall model's performance.

Contents

Introduction **v**

1 Tokenization and Embedding **1**
1.1 Challenges . . . 2
1.2 Rule-based techniques . . . 2
1.3 Trained techniques . . . 4
1.3.1 Byte-Pair Encoding . . . 5
1.3.2 WordPiece . . . 6
1.3.3 Unigram . . . 7
1.3.4 SentencePiece . . . 10
1.3.5 Training of a tokenizer . . . 11
1.4 Embedding . . . 12
1.4.1 Word2Vec . . . 14
1.4.1.1 Skip-Gram . . . 16
1.4.1.2 CBOW . . . 18
1.4.1.3 Algorithm Optimization . . . 20

2 Transformers **21**
2.1 Positional Encoding . . . 21
2.2 Self-attention layer . . . 22
2.3 Transformer layer . . . 28
2.3.1 Residual connection . . . 30
2.3.2 LayerNorm and BatchNorm . . . 31
2.4 Applications . . . 33

3 Language Models **35**
3.1 Encoder model: BERT . . . 36
3.1.1 Pre-training . . . 36

3.1.2 Fine-tuning . . . 37
3.2 Decoder model: GPT-3 . . . 39
3.2.1 Masked self-attention . . . 40
3.2.2 Training of the decoder . . . 41
3.2.3 Challenges and limitations of GPT-3 . . . 43
3.3 Encoder-decoder model: machine translation . . . 44
3.4 Developing an LLM . . . 49
3.5 Evaluation metrics . . . 53
3.5.1 BLEU . . . 53
3.5.2 ROUGE . . . 57
3.5.3 Perplexity . . . 59
3.5.4 GLUE Benchmark . . . 60
3.5.5 Evaluation of LLaMA and GPT-3 . . . 62

4 PEFT **65**
4.1 LoRA . . . 66
4.1.1 LoRA+ . . . 71
4.1.2 LoRA-drop . . . 73
4.2 SMT . . . 75
4.3 SVFT . . . 79
4.4 LLM in HPC . . . 82
4.5 Experiments on Leonardo . . . 84

Conclusions **91**

A High Performance Computing **93**
A.1 Structure and workflow of a supercomputer . . . 93
A.2 Parallelism . . . 96
A.3 Supercomputing in Cineca . . . 98

Introduction

In the last years the development of Large Language Models (LLMs) has revolutionized the field of natural language processing (NLP), enabling significant advancements in several contexts, such as text translation, code generation and question answering. To address all these tasks, LLMs have become increasingly complex and resource-intensive, since they require an extensive training on a huge amount of data, mostly in English, that need to be preprocessed. For example, recent LLMs as LLaMA-3 [1], PaLM [2] and GPT-4 [3] have 70 billion, 540 billion and 1.8 trillion parameters, respectively, and are trained on a different variety of data, as books, web pages, scientific articles, public news and code sources.
Since the most famous LLMs are meant to be general purpose, a fine-tuning procedure is usually needed to tailor the models on specific domains. However, updating all parameters, would be computationally expensive and would require significant memory resources. For instance, fine-tuning a model like GPT-3 [4] requires computing nearly 175 billion parameters, making it impractical for many applications. This brought to the exploration of parameter-efficient fine-tuning (PEFT) methods that allow to modify only a small subset of parameters while keeping the majority fixed. This approach not only reduces the computational effort but also minimizes the risk of catastrophic forgetting, particularly when working with limited task-specific data. Additionally, compared to training from scratch, fine-tuning may be achieved with fewer labeled instances and less computing resources by utilizing the knowledge already present in these huge models. This method improves the effectiveness of implementing LLMs in practical applications while also democratizing access to cutting-edge AI capabilities.
All the most important PEFT methods rely on different types of matrix factorization, such as low-rank (Low-Rank Adaptation or LoRA [5]), sparse (Sparse Matrix Tuning or SMT [6]) or Singular Value Decomposition (Singular Value Fine Tuning or SVFT [7]) in order to decompose the trainable fine-tuning matrix. These factorizations contain fewer parameters than full fine-tuning, allowing to significantly

decrease training time and computational resources without affecting the overall model's performance. For instance, employing Low Rank Adaptation (LoRA) to LlaMA-2-7B can reduce the number of parameters involved in backpropagation from 7B to 160M.

This thesis is structured into four chapters as follows. The first chapter deals with two techniques, namely tokenization and embedding of a training corpus text. The first technique allows to split any word into smaller, meaningful and representative subwords, known as tokens, in order to build a token vocabulary. Since computers deal with numbers instead of words, the second technique converts any token into a vector in a high-dimensional embedding space. Embedding's goal is to build a map such that tokens belonging to a similar semantic field are represented through vectors with similar directions. The second chapter contains an overview of the existing literature on transformer architecture. In particular, it focuses on the importance of multi-head attention mechanisms, which allow to capture short and long range dependencies between tokens in a sequence, enabling the model to understand complex patterns inside human language. The third chapter is about Language Models. It first provides three different applications of the transformer architecture for solving different tasks, as generating or translating text, and remarking the importance of using different attention mechanisms depending on the task. Then, it describes the procedure of LLM training for next-token generation task and several evaluation metrics. Finally, the fourth chapter contains an overview of different PEFT techniques. It focuses on the effect of applying several matrix factorization techniques, as low-rank, sparse and SVD, for fine-tuning an LLM. Moreover, this chapter discusses the advantages from a computational point of view of applying PEFT techniques instead than full fine-tuning, and the achieved performances in terms of different evaluation metrics. Several experiments were performed on Leonardo supercomputer and the results reported in this thesis, underling the fundamental role of High Performance Computing (HPC) for training and fine-tuning LLMs.
This thesis is the result of the internship at the HPC department of Cineca [8], a non profit Consortium, made up of 118 universities, agencies and public institutions. Cineca is the largest Italian computing centre and one of the most important worldwide, providing support to the scientific community and companies through HPC. Since 2022, Cineca hosts Leonardo, the 9th most powerful supercomputer in the world, according to the November 2024 TOP500 list [9].

Chapter 1

Tokenization and Embedding

Tokenization refers to the process of splitting an input text into smaller chunks, such as characters, words or subwords, called *tokens*. This process has a lot of modern applications since it helps machines to accomplish one of the most complex tasks: understanding human language by breaking it down into easier pieces. The main goal of tokenization is to represent human language in a rather small but meaningful vocabulary using statistical techniques or predefined rules. In this process several challenges need to be addressed, such as the optimal vocabulary size or the handling of punctuation and special characters. Depending on the natural language processing (NLP) task there are different tokenization strategies to segment a text: some methods are elementary, since they are based on predefined rules and break down the text into small units, such as characters or words, or into bigger ones, such as whole sentences or paragraphs. This type of tokenization is called *rule-based*. More complex strategies involve statistical tools, learning the frequencies and patterns of characters in the text, thus we refer to them as *trained* tokenization techniques. Since a machine is not able to deal with text inputs, the tokens need to be converted into a numerical representation through an embedding, whose main goal is to capture semantic and syntactic relationships between tokens in a high dimensional space.

In this chapter, we describe the main challenges of tokenization and we give a thorough description of several tokenization techniques, both rule-based and trained, analyzing the similarities and differences. Then, we introduce some architectures for word embedding and perform different experiments; finally, we briefly review few optimization strategies for the described algorithms.

1.1 Challenges

Human language is wide and full of nuances and ambiguities, which a tokenizer should deal with. Some of these challenges are the following [10]:

- **Ambiguity and Polysemy**: words may share the same root or have multiple meanings or contexts, therefore a tokenizer may struggle to determine the correct meaning, which can possibly lead to misinterpretations [11].

- **Handling special characters**: an incorrect tokenization of punctuation marks or other special characters can lead to syntactic and semantic errors, especially when these are part of a token [12].

- **Language specificity**: a tokenizer may encounter some problems when dealing with languages without clear spaces between words [13].

- **Vocabulary size**: a vocabulary may contain a huge number of tokens, due to the vastness of words. For instance, singular and plural nouns need to be encoded in different tokens, which may increase significantly the vocabulary size.

1.2 Rule-based techniques

We briefly discuss some rule-based techniques depending on their different approaches [14]:

- **Character Tokenization** breaks down the text into individual characters, which means that any character is a token. This tokenization strategy is beneficial in certain tasks, like text generation to predict the next character in a sequence, therefore it may be applied for modelling character-level languages. An English character-based vocabulary is very slim since it contains only 256 tokens; moreover, character tokenization does not face the problem of Out-Of-Vocabulary (OOV) words, since the vocabulary contains all characters used in that language. However, characters do not hold as much information individually as a word would do. Ideogram-based languages have lots of information in a single character, encouraging a character-based approach rather than a word one [15].

- **Word Tokenization** divides the text into individual words, using whitespaces or punctuation marks as delimiters. Many NLP tasks share this approach, in which words are treated as basic units of language. Word-based vocabularies can have huge dimension, since in the English language there are approximately 170.000 words. In spite of character tokenization, word tokenization faces the challenge of Out-Of-Vocabulary words, which are words that appear in the new text to tokenize but not in the vocabulary. For instance, if we train a vocabulary on English text data and we want to tokenize an Italian word, then this is probably an OOV word; the same happens when dealing with location or people's names [16].

- **Sentence Tokenization** segments the text into individual sentences, which are delimited by dot marks.

- **Paragraph Tokenization** breaks down the text into individual paragraphs, which are delimited by blank lines.

- **Regular Expressions Tokenization** extracts token provided a regex pattern. A regex or regular expression is a sequence of characters specifying a matching pattern in a text: for instance, the string "\d" extracts digits or "\w" word characters [17]. Regex tokenization is also suitable for more complex tokenization rules, such as extracting an e-mail address from a text, provided a proper regex pattern [18].

Example 1. We apply the previous described tokenization techniques to the following text sequence [19]:

```
test_text = "Tokenization is an important NLP task. It helps breaking
down text into smaller units."
```

- **Character Tokenization**:

```
tokenized_text = ['T', 'o', 'k', 'e', 'n', 'i', 'z', 'a', 't',
'i', 'o', 'n', ' ', 'i', 's', ' ', 'a', 'n', ' ', 'i', 'm', 'p',
'o', 'r', 't', 'a', 'n', 't', ' ', 'N', 'L', 'P', ' ', 't', 'a',
's', 'k', '.', ' ', 'I', 't', ' ', 'h', 'e', 'l', 'p', 's', ' ',
'b', 'r', 'e', 'a', 'k', 'i', 'n', 'g', ' ', 'd', 'o', 'w', 'n',
' ', 't', 'e', 'x', 't', ' ', 'i', 'n', 't', 'o', ' ', 's', 'm',
'a', 'l', 'l', 'e', 'r', ' ', 'u', 'n', 'i', 't', 's', '.']
```

- **Word Tokenization**:

```
tokenized_text = ['Tokenization', 'is', 'an', 'important',
'NLP', 'task', '.', 'It', 'helps', 'breaking', 'down', 'text',
'into', 'smaller', 'units', '.']
```

- **Sentence Tokenization**:

  ```
  tokenized_text = ['Tokenization is an important NLP task.', 'It
  helps break down text into smaller units.']
  ```

- **Paragraph Tokenization**:

  ```
  tokenized_text = ['Tokenization is an important NLP task. It
  helps breaking down text into smaller units.']
  ```

- **Regular Expressions Tokenization**: it depends on the chosen regex pattern, for instance, we select only the capitalized words with the pattern "[A-Z]\w+".

  ```
  tokenized_text = ['Tokenization', 'NLP', 'It']
  ```

We remark that the input test sequence contains two sentences but only one paragraph, therefore the sentence and paragraph tokenization techniques give different results.

1.3 Trained techniques

Subword tokenization lies in between word-based and character-based tokenization: on one hand, we deal with large vocabularies and a great amount of out-of-vocabulary words, while on the other hand, we deal with long sequences and less meaningful tokens [20]. Subword techniques rely on the idea that frequently used words should not be decomposed into smaller subword, while rare words can be split into meaningful subwords [21]. The following example shows the core idea behind subword tokenization.

Example 2. Consider the word *tokenization*, which can be split into the root *token* and the suffix *ization*. The suffix gives the root a slightly different meaning: the algorithm should then be able to recognize that the words *tokens*, *tokenizer* and *tokenizing* share all the same root with similar meanings [15].

There exists different ways to implement subword-based tokenizers: BERT tokenizer uses the prefix ## before the tokens, which are part of a word but not at its beginning, and is based on the WordPiece algorithm [22]. Instead, GPT-2 is

based on Byte-Pair Encoding algorithm and uses a special symbol "Ġ" to handle whitespaces before words [23]. In the previous example, using BERT the tokens are therefore *to*, *##ken* and *##ization*, while using GPT-2 the tokens are *Ġtoken* and *ization*. Other tokenizers may use different symbols or instead use that symbol at the beginning of the word. In the following sections, we analyze some algorithms for subword tokenization, such as Byte-Pair Encoding, WordPiece, Unigram and SentencePiece. All these algorithms share two common parts: a token *learner* which takes a raw training corpus text and creates a vocabulary, and a token *segmenter* which takes a raw test sentence and applies tokenization according to the created vocabulary [24]. For this reason, subword tokenization techniques are also known as *trained* methods, since the vocabulary is induced from a raw text dataset.

1.3.1 Byte-Pair Encoding

Byte-Pair Encoding (BPE) was introduced in 1994 as a simple and iterative data compression technique, which replaces the most frequent pair of bytes in a sequence with a single byte [25]. For NLP tasks, instead of considering bytes, BPE merges characters of a word. Intuitively, the initial tokens are characters or punctuation with a special token acting as whitespace. The algorithm proceeds iteratively by joining together the most frequent characters so that tokens emerge [26]. We see now in details how it works.

First of all, a normalization step involves a general cleanup of the corpus text, such as removing needless whitespaces, lowercasing and accents, and a pre-tokenization step with a word-based tokenizer splits the corpus text into word strings on whitespaces and punctuation and adds the symbol "_" at the end of each word to prevent merging of pairs of two different words [27]. All the unique characters of all words are stored in a vocabulary, and for each of them it counts its occurrence. BPE iteratively counts all symbol pairs and replaces each occurrence of the most frequent pair ("A", "B") with a new symbol ("AB"), which will be the first merge rule. Each merge operation produces a new merge rule and a new symbol, which is added to the vocabulary. The algorithm ends when the final size k of the vocabulary is reached. Thus, the only hyperparameter of the algorithm is k. The symbols in the final vocabulary are the tokens [28].

To tokenize a new test text, firstly, the token segmenter applies normalization

and pre-tokenization steps. Then, the merge rules learned during the BPE procedure are applied in order. According to the description above, the first merge rule is ("A", "B") → ("AB") and has to be applied at first. Any token not in the vocabulary is replaced by an unknown token "[*UNK*]" [29].

Example 3 (BPE)**.** Consider the string *aaabdaaabac*. The pair *aa* occurs most often and is replaced by *A=aa*. The initial string is now *AabdAabac*. The pair *ab* occurs most often, and again is replaced by *B=ab*. The string is *ABdABac*. Finally, the encoding *C=AB* is applied and the string is compressed into *CdCac*. To retrieve the initial string is sufficient to apply the previous replacement in reverse order [30].

A pseudocode of the BPE algorithm is shown in Algorithm 1.

Algorithm 1: Byte-Pair Encoding [31]

Input: Corpus text C
Hyperparameter: Number of merges k
Output: Vocabulary $\mathcal{V}$

$\mathcal{V}^{(1)} \leftarrow$ all unique characters in C
for $i = 1, \ldots, k$ **do**
  $t_{NEW} \leftarrow$ most frequent pair of adjacent tokens in C
  $t_L, t_R \leftarrow$ left and right element of the pair t_{NEW}
  $\mathcal{V}^{(i+1)} \leftarrow \mathcal{V}^{(i)} + t_{NEW}$
  Replace each occurrence of (t_L, t_R) in C with t_{NEW}
end for
$\mathcal{V} \leftarrow \mathcal{V}^{(k+1)}$
return $\mathcal{V}$

1.3.2 WordPiece

WordPiece is another subword tokenization algorithm with a similar approach of BPE and was outlined in 2012 for building a successful voice search system applied to Japanese and Korean at Google [32]. It gained popularity through the model BERT. The principal difference between BPE and WordPiece is in the way in which symbols are added to the vocabulary: BPE counts the frequency, WordPiece maximizes the likelihood of the data once added to the vocabulary [33].

Instead of using the symbol "_" at the end of each word, WordPiece adds the prefix "##" before each letter that does not start a word: the initial vocabulary contains all characters present at the beginning of a word and inside a word preceded with the prefix "##" [34]. Now, it calculates the score of each pair ("A", "B"), defined as follows:

$$\text{Score} = \frac{\text{Frequency of the pair}}{\text{Frequency of first element} \times \text{Frequency of second element}}$$

and selects the pair which maximizes the score. BPE maximizes only the frequency of the pair, while WordPiece considers also the frequencies of the two elements which form the pair: in this way, given two pairs with same frequency, WordPiece chooses the one whose elements are less frequent. Finally, the pair with highest score is added to the vocabulary and the algorithm proceeds iteratively like in BPE, until the vocabulary size is reached or the score falls below a certain threshold.

To tokenize a word, WordPiece looks for the longest possible token at the beginning of it and the same is applied on the remaining part of the word: thus, it is not necessary to save any merge rule, but only the token vocabulary. Another difference is in the use of the symbol "$[UNK]$": if it is not possible to find any subword in the vocabulary then the whole word is tokenized as unknown, instead of the individual characters as in BPE's approach [35].

Algorithm 2 shows a pseudocode of WordPiece algorithm, which differs from BPE only in the way left and right token of a pair are taken.

1.3.3 Unigram

Unigram was introduced in 2018 to improve neural machine translation [36]; it works in the opposite direction with respect to the previous two tokenization techniques, since it initializes the base vocabulary with a large number of symbols and removes tokens iteratively until it reaches the desired size [37]. A Unigram model is a type of statistical language model based on the hypothesis that each token of a word is independent of the tokens before it. This implies also that the probability of a word is the product of the probabilities of its tokens [38]. Although this model is not suitable for text generation because we would always get the most common token, it is very useful for tokenization to estimate the relative

Algorithm 2: WordPiece

Input: Corpus text C
Hyperparameter: Number of merges k
Output: Vocabulary $\mathcal{V}$

$\mathcal{V}^{(1)} \leftarrow$ all unique characters in C
for $i = 1, \ldots, k$ **do**
  $t_{NEW} \leftarrow$ highest scoring pair of adjacent tokens in C
  $t_L, t_R \leftarrow$ left and right element of the pair t_{NEW}
  $\mathcal{V}^{(i+1)} \leftarrow \mathcal{V}^{(i)} + t_{NEW}$
  Replace each occurrence of t_L, t_R in S with t_{NEW}
end for
$\mathcal{V} \leftarrow \mathcal{V}^{(k+1)}$
return $\mathcal{V}$

likelihood of different sentences. We see now in detail how the tokens are removed from the initial vocabulary.

There are several ways to build the base vocabulary, for instance, a training corpus text can be pre-tokenized with word tokenization and the vocabulary can correspond to all strict substrings of the words or BPE algorithm with a large vocabulary size can be applied. Moreover, the vocabulary stores also the probability of each token. The training of the Unigram tokenizer is based on the *Expectation-Maximization* method and consists of two steps:

1. *Expectation*: Unigram algorithm computes for any given word of the training corpus the probabilities of its any possible segmentation into tokens from the current vocabulary and finds the best possible tokenization in probabilistic terms.[1] In general, a word split in few tokens has higher probability than tokenizing it with a lot of tokens: our aim is indeed to minimize the number of tokens per word. After finding the highest probability tokenization of each word, Unigram algorithm defines a negative log-likelihood loss over the corpus text given the current vocabulary as follows:

$$\mathcal{L} = -\sum_{\text{word in corpus}} \log\left(\mathbb{P}(\text{word})\right),$$

[1]The probability of a token is its frequency in the corpus text divided by the sum of the frequencies of all tokens.

where the sum is computed on all words of the training corpus and $\mathbb{P}(\text{word})$ is the probability associated with the best possible tokenization for a given word.

2. *Maximization*: The token that impacts the least the loss on the corpus is removed, since our goal is to reduce the vocabulary size. Thus, we perform a loop on all possible tokens: at each iteration a token is removed from the vocabulary and the associated loss is evaluated. A token is definitely removed from the vocabulary if its associated loss is the smallest between on all tokens. The idea behind is that the removed symbol affects the least the loss, thus is less needed. Usually, Unigram algorithm removes not only one token per iteration, but a percentage p of tokens which minimize the loss and are not basic characters, since we want to keep them in order to tokenize any word.

The algorithm ends when the vocabulary reaches the desired size k, thus the hyperparameters are both p and k [39].

A pseudocode of Unigram is shown below in Algorithm 3.

Algorithm 3: Unigram

Input: Corpus text C
Hyperparameters: Number of merges k, percentage p
Output: Vocabulary $\mathcal{V}$

$\mathcal{V} \leftarrow$ all strict substrings of words in C
while $|\mathcal{V}| > k$ **do**
  $\mathbb{P}_{\text{word}} \leftarrow$ highest probability of any possible segmentation
  $\mathcal{L}_{\text{global}} \leftarrow$ global loss on all words using p_{word}
  $\mathcal{L}_{\text{new}} \leftarrow$ losses with one token removed
  $\tau_{\text{remove}} \leftarrow$ remove $p\%$ of tokens with highest $\mathcal{L}_{\text{global}} - \mathcal{L}_{\text{new}}$
  $\mathcal{V} \leftarrow \mathcal{V} \setminus \tau_{\text{remove}}$
end while
return $\mathcal{V}$

To tokenize a word, all possible segmentations are list, according to the token of the vocabulary, and the probability of each segmentation is evaluated; finally, the segmentation corresponding to the highest probability is chosen as tokenization

of the initial word. In practice, computing all probabilities of the possible splits of a word before comparing them is too expensive: the most probable sequence of tokens can be found much more efficiently by applying the *Viterbi* algorithm [40]. Briefly, we build a graph with a branch from character a to b if the subword from a to b is in the vocabulary; to each branch is associated the probability of the corresponding subword. Viterbi algorithm determines, for each position in the word, the tokenization with the highest probability ending at that position and proceeding until the last character of the word [39].

1.3.4 SentencePiece

All tokenization algorithms described above share the problem that the input text uses spaces between words, although not all languages use spaces to separate words. SentencePiece algorithm, developed in 2018 for text processing, handles the input text as a sequence of Unicode characters and treats the space as a special character, often denoted with "_" [41]. It uses BPE or Unigram algorithm to build the vocabulary [21]. The great advantage of SentencePiece is that is language-independent since it doesn't require a pre-tokenization step, which is very useful for languages such as Chinese or Japanese where the space character is not present [42].

Example 4 (Tokenization techniques of LLMs)**.** Several LLMs rely on the previous described tokenization techniques:

- BPE was used for the GPT models [4] [23], RoBERTa [43], BART [44] and DeBERTa [45].
- WordPiece was used for BERT [22], DistilBERT [46], MobileBERT [47], Funnel Transformers [48] and MPNet [49].
- Unigram is often used together with SentencePiece, which is the tokenization algorithm used by models like AlBERT [50], T5 [51], mBART [52], Big Bird [53] and XLNet [54].
- LLaMA [55] and Mistral [56] tokenizers are BPE model based on SentencePiece.

Example 5. We apply the previous described tokenization techniques with pre-trained tokenizers to the following text sequence [19]:

```
test_text = "Tokenization is an important NLP task. It helps breaking
down text into smaller units."
```

- **BPE** (GPT-2):

```
tokenized_text = ['Token', 'ization', 'Ġis', 'Ġan',
'Ġimportant', 'ĠN', 'LP', 'Ġtask', '.', ĠIt', 'Ġhelps',
'Ġbreaking', 'Ġdown', 'Ġtext', 'Ġinto', 'Ġsmaller', 'Ġunits',
'.']
```

- **WordPiece** (BERT):

```
tokenized_text = ['To', '##ken', '##ization', 'is', 'an',
'important', 'NL', '##P', 'task', .', 'It', 'helps', 'breaking',
'down', 'text', 'into', 'smaller', 'units', '.']
```

- **SentencePiece with Unigram** (XLNet):

```
tokenized_text = ['_To', 'ken', 'ization', '_is', '_an',
'_important', '_N', 'LP', _task', '.', '_It', '_helps',
'_breaking', '_down', '_text', '_into', '_smaller', '_units',
'.']
```

- **SentencePiece with BPE** (LLaMA):

```
tokenized_text = ['_Token', 'ization', '_is', '_an',
'_important', '_N', 'LP', '_task', '.', '_It', '_helps',
'_breaking', '_down', '_text', '_into', '_smaller', '_units',
'.']
```

1.3.5 Training of a tokenizer

We perform an experiment, building a new tokenizer from an old one: in particular, given the GPT-2 tokenizer, we create a new vocabulary with new tokens, trained on a dataset containing Italian legislative texts, public and private acts [57]. GPT-2 is a 1.5 billion parameters transformer-based language model, trained on a dataset of 8 million web pages: its goal is to predict the next word, given some previous context words [58]. Its vocabulary contains 50.257 tokens, which correspond to 256 base tokens, a special end-of-text token and the tokens learned after 50.000 merges [23].
After downloading the GPT-2 tokenizer, we import the Italian dataset and use it to train a new tokenizer with a vocabulary of 52.000 tokens. Several remarks can be made:

- If we tokenize an Italian sentence from the dataset with GPT-2 tokenizer, we observe that the text is split in a large number of tokens, while the new tokenizer, which has been trained on Italian data, is able to do far fewer splits and is almost similar to a word tokenizer. For instance, if we consider the data `'In attuazione della determinazione ...'` then GPT-2 tokenizer splits it into `['In', ' att', 'u', 'az', 'ione', ' de', 'lla', ' determin', 'az', 'ione']`, while the new tokenizer divides it into `['In', ' attuazione', ' della', ' determinazione']` (we remove in both tokenizations the symbol `Ġ`). Moreover, we remark that the new vocabulary is strongly affected by words coming from the semantic field of law: if we tokenize the word `'glicerofosfolipidi'` from the Italian dataset BioBERT [59], containing words from the biomedical field, then the new tokenizer splits it in lot of tokens `['glic', 'ero', 'fos', 'foli', 'pidi']`.
- If we consider now an English word, such as `'Hello'`, then the new tokenizer splits into two tokens `['H', 'ello']`, while GPT-2 tokenizer keeps it as a single token: this implies that several tokens of the initial vocabulary have been removed and replaced with Italian based tokens.

The described procedure and the notebook of training a new tokenizer from an old one can be found on the website [60].

1.4 Embedding

Deep Learning architectures require numerical data as input and are not able to process strings, therefore we need to convert our token vocabulary into a numerical representation. Briefly, each token of the vocabulary $\mathcal{V}$ is mapped into a unique *word embedding* and transformed into numerical vectors; these vectors are then stored in an embedding matrix $\mathbf{\Omega}_e \in \mathbb{R}^{D\times|\mathcal{V}|}$, where D is the embedding size. Known embedding sizes go from 768 (BERT [22]) to 12.288 (GPT-3 175B [4]), while a vocabulary contains approximately from 30.000 (BERT [22]) to 50.000 tokens (GPT-2 [23]): this implies that, even before the main transformer network, there are a lot of parameters in $\mathbf{\Omega}_e$ to learn. The goal is to understand how word embedding works and the matrix $\mathbf{\Omega}_e$ is built [61].

Word embedding is a technique to represent individual tokens as vectors in a D-dimensional space and capture semantic relationships between tokens: vectors

representing words with similar meanings are mapped close to each other, while vectors representing words with no semantic relationship are mapped far to each other [62]. Figure 1.1 shows an example of the embedding of two semantic similar groups of words.

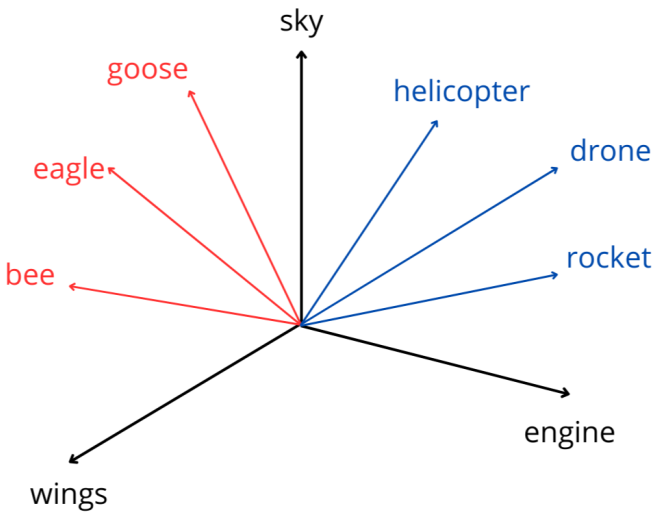

Figure 1.1: Semantic relationships between different words in a 3-dimensional embedding space. Each of the three axis can be interpreted as a feature. Here tokens coincide with words. Image inspired from [63].

The *cosine similarity* is a way to measure how close two vectors are and is defined as:

$$S_C(v, w) = \frac{v \cdot w}{||v||||w||}$$

By Cauchy-Schwarz inequality, $-1 \leq S_C \leq 1$: it is maximal when two vectors are parallel and minimal when they are antiparallel [64].

Moreover, the same token (or word) can share different meanings depending on the context: hence, a word embedding should consider a token not only as single independent element but as part of a whole semantic context since the surrounding tokens influence its meaning. A common but computationally unpractical representation of these vectors is *one-hot encoding*: each token is mapped into a $|\mathcal{V}|$-dimensional vector with all zeros except for a value 1 corresponding to the position of the token in the vocabulary [65]. This representation involves many issues: at first, if the vocabulary size is too big, then the vectors are huge. Moreover,

removing a token from the vocabulary implies a new creation of one-hot encoding representation. Finally, this representation does not capture any semantic meaning between tokens, since it is only related to the position of the token in the vocabulary and all vectors are orthogonal to each other [66]. Figure 1.2 shows an example of the one-hot encoding of different words.

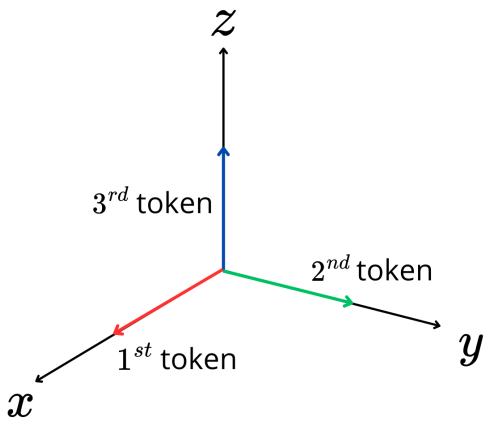

Figure 1.2: One-hot encoding in a vocabulary containing 3 tokens in a 3-dimensional embedding space: the cosine similarity between the vectors is zero and the semantic relationships of the words is not captured. The size of the vocabulary $|\mathcal{V}|$ has to coincide with the embedding dimension D.

Other techniques, such as Word2Vec, solve the previous issues of dimensionality and lack of semantic meaning: the goal is then to take in input a training tokenized corpus text and return the semantic.

1.4.1 Word2Vec

Word2Vec technique was developed by Google in 2013 and is a shallow two-layer neural network which produces a vector space of dimension D. To each token it is assigned a unique vector in the space and the relative positions and distances of these vectors are determined by the semantic and syntactic similarities of the corresponding tokens. Word2Vec can be implemented using one of the following two architecture models: the *Continuous Bag Of Words* (CBOW) Model or the *Continuous Skip-Gram* Model [67]. Both models reduce the dimensionality of the data and create real-valued D-dimensional dense vectors: in CBOW model

the surrounding (also called context) tokens are combined to predict a token in the middle, while in Skip-Gram model one input token is used to predict the surrounding tokens. This means that the embedding is learned by looking at nearby tokens and the models capture the meaning of a token in a given context: similar embeddings correspond to similar tokens and vice versa. Both networks require an hyperparameter, called *window-size* ws, which can be thought as a sliding window of fixed size moving along the text, as shown in Example 6: in both algorithms, we consider a middle token and the ws tokens before and ws tokens after it. Of course, given a tokenized sentence, the first token at the beginning of a sentence does not have any tokens before it, then we only consider the ws tokens after it; similarly, if the tokens is at the end of the sentence [68].

Example 6. We consider the following text sequence using word tokenization to tokenize it:
`test_text = "Word embedding captures semantic relationships between tokens"`
Table 1.1 and Table 1.2 show Word2Vec workflow of the considered tokens using two different window sizes. The context of a token can be represented as a set `(current_token, context_tokens)`.

Window size ws	**Current Token**	**Context Tokens**
1	`Word`	`(Word, embedding)`
	`embedding`	`(embedding, Word)` `(embedding, captures)`
	`captures`	`(captures, embedding)` `(captures, semantic)`
	`semantic`	`(semantic, captures)` `(semantic, relationships)`
	`relationships`	`(relationships, semantic)` `(relationships, between)`
	`between`	`(between, relationships)` `(between, tokens)`
	`tokens`	`(tokens, between)`

Table 1.1: Word2Vec: example of current and context tokens with window size $ws = 1$.

Window size ws	Current Token	Context Tokens
2	`Word`	`(Word, embedding)` `(Word, captures)`

	`embedding`	`(embedding, Word)`
		`(embedding, captures)`
		`(embedding, semantic)`

	`captures`	`(captures, Word)`
		`(captures, embedding)`
		`(captures, semantic)`
		`(captures, relationships)`

	`semantic`	`(semantic, embedding)`
		`(semantic, captures)`
		`(semantic, relationships)`
		`(semantic, between)`

	`relationships`	`(relationships, captures)`
		`(relationships, semantic)`
		`(relationships, between)`
		`(relationships, tokens)`

	`between`	`(between, semantic)`
		`(between, relationships)`
		`(between, tokens)`

	`tokens`	`(tokens, relationships)` `(tokens, between)`

Table 1.2: Word2Vec: example of current and context tokens with window size $ws = 2$.

1.4.1.1 Skip-Gram

The Skip-Gram model consists of an input, hidden and output layer and its goal is to predict the context tokens given one main token in the middle: in particular, given a token, it predicts the ws tokens before and the ws tokens after it. The training objective of the Skip-Gram model is to maximize the probability of predicting context tokens given the main token: let $t_1, \ldots, t_N$ be a sequence of

tokens, then the objective can be written as the average log probability [69]:

$$\frac{1}{N}\sum_{i=1}^{N}\sum_{-ws\leq c\leq ws,\ c\neq 0}\log(\mathbb{P}(t_{i+c}|t_i)).$$

The input layer has $|\mathcal{V}|$ neurons and is the main token represented in one-hot encoding. The embedding weight matrix $\mathbf{\Omega}_e \in \mathbb{R}^{D\times|\mathcal{V}|}$ transforms the input into the hidden layer and returns the D-dimensional embedding vector of the main token without any activation function [67]. Since D is much smaller than $|\mathcal{V}|$ the embedding extracts the most important features of the token. The higher D, the more information the embedding will extract: although, if D is too large the computation will be too expensive [65]. Finally, the unembedding weight matrix $\mathbf{\Omega}_u \in \mathbb{R}^{|\mathcal{V}|\times D}$ transforms the hidden into the output layer: the output layer has $|\mathcal{V}|$ neurons and predicts $2ws$ context words with the softmax as activation function. Once the training is completed through the whole vocabulary, the word embedding matrix $\mathbf{\Omega}_e$ represents the tokenized vocabulary.

Figure 1.3 shows the Skip-Gram architecture.

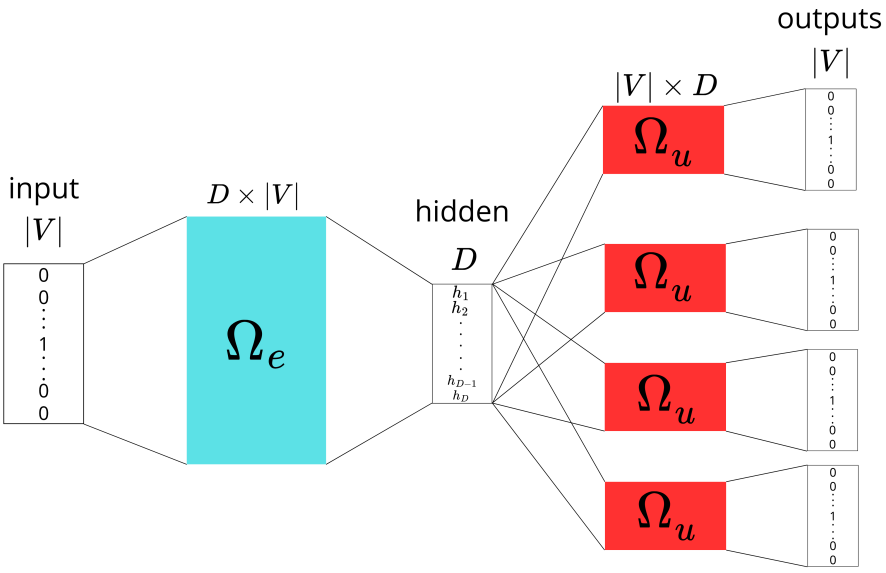

Figure 1.3: Skip-Gram architecture with $ws = 2$. Each output is computed using the same unembedding matrix $\mathbf{\Omega}_u$ [70]. Image inspired from [71].

Remark. There is no activation function between the input and hidden layer: the goal of the hidden layer is to map the one-hot encoded vectors in a lower D-dimensional space, keeping separation between dissimilar words. Instead, an activation function produces a compression of the whole space in a certain region and an expansion in another: this alteration of the space reduces the mapping space available [72]. Between the hidden and output layer a softmax activation function is applied to generate a discrete probability distribution: once the network has been trained, given a token it outputs a $|\mathcal{V}|$-dimensional vector y_{pred} and the context tokens to predict are the $2ws$ which have highest probability.
To apply backpropagation, a loss function is evaluated: assume, for the purpose of the explanation, to use just the vector difference. We obtain $2ws$ prediction errors: a prediction error is the difference between the output vector y_{pred}, i.e. the probability distribution of the context token computed through the softmax, and the one-hot encoded context token y_{true}, i.e. the true probability distribution of the context token that is considered. The global function loss is then the sum of the $2ws$ prediction errors, which considers all the context tokens: this error is applied for backpropagation to update the weights of the embedding and unembedding matrices. Other loss functions can be applied, for instance a good choice is the cross-entropy loss function that compares the two probability distributions [73]. It is also worth noticing that the network does not take into account the position of the context tokens, since it aims only to predict them without any order.

Example 7 (Optimal case)**.** Assume that y_{pred} contains all zeros except the number $1/(2ws)$ at $2ws$ positions. Then, for a given context token, we compute the loss, obtaining a vector with all zeros, except the number $1/(2ws)$ at $2ws - 1$ positions and the number $1/(2ws) - 1$ at the corresponding position of the one-hot encoded context token. By computing the sum of all the losses we get a zero vector, which implies zero loss.

1.4.1.2 CBOW

The CBOW model does the opposite operation: given the context tokens, it predicts the main token in the middle. Indeed, the context tokens are the ws tokens before and the ws tokens after the middle token. The training objective of the CBOW model is to maximize the probability of predicting the main token given the context tokens: let $t_1, \dots, t_N$ be a sequence of tokens, then the objective

can be written as the average log probability [69]:

$$\frac{1}{N}\sum_{i=1}^{N}\log(\mathbb{P}(t_i|t_{i-ws},\dots,t_{i-1},t_{i+1},\dots,t_{i+ws})).$$

As in Skip-Gram it consists of an input, hidden layer and output layer with the same number of neurons: the input layer now is used to represent the context tokens in one-hot encoding, the hidden layer to learn the word embeddings and the output layer to predict the middle token [67]. In particular, the embedding weight matrix $\mathbf{\Omega}_e$ transforms each of the $2ws$ context tokens into a single hidden layer through an average of the embedded context vectors [66]. Again, after training, the embedding matrix is $\mathbf{\Omega}_e$. One of the greatest advantages of CBOW is that it needs to predict only one single token given a set of context tokens: therefore, it can be trained on much larger dataset than Skip-Gram model [74].

Figure 1.4 shows the CBOW architecture.

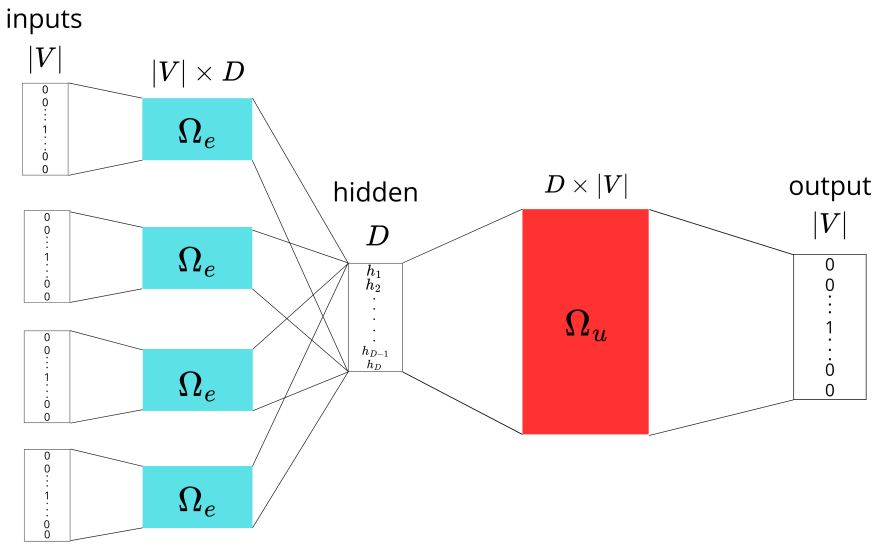

Figure 1.4: CBOW architecture with $ws = 2$. The four lightblue matrices represent the same embedding matrix $\mathbf{\Omega}_e$. Image inspired from [71].

Remark. The same remark about activation functions holds also for CBOW. In CBOW we do not lose the information on the position of the context tokens in the output layer, but we lose it in the hidden layer, since the output layer predicts the middle token.

1.4.1.3 Algorithm Optimization

CBOW has to predict only a single token, which means that the softmax has to be evaluated only one time, instead of $2ws$ as in Skip-Gram: comparatively, CBOW is much faster to train than Skip-Gram. It is worth noticing that the number of weights of the previous models is huge: if we take 30.000 tokens and an embedding size $D = 1.000$, learning the embedding matrix means learning 30 millions weights. If we also consider that we need to learn the weights of the unembedding matrix and the biases then the number of parameters of the network becomes huge: this is extremely computationally expensive, therefore there exist two optimization techniques: *hierarchical softmax* and *negative sampling*. In practice, hierarchical softmax works better for infrequent words, while negative sampling for frequent words and lower dimensional vectors [75].

Chapter 2

Transformers

A transformer is a Deep Learning architecture proposed by Google in 2017 in the paper *Attention Is All You Need* [76] to replace Recurrent Neural Networks (RNN) and Long-Short Term Memory (LSTM) models, which have inherently a sequential nature precluding parallelization within training data. Transformers rely on the self-attention mechanism, which draws dependencies between input and output, and achieve significantly more parallelization. Although transformers were introduced for machine translation tasks, in the last few years, a huge number of transformer-based models were developed, such as LLMs, image generators or speech recognition systems, which allow to solve various modern NLP and Computer Vision tasks. For instance, a model may take in input an audio and produce its transcript or another one may generate an image given an input source text. GPT-3, one of the most known LLMs, is not only able to predict and generate new tokens given an input sequence, but can also translate tests, answer questions or extract relevant information from documents.

This chapter is inspired from the book [61] and deals in details with the transformer architecture, describing its core components, which consists mostly of multilayer perceptrons and attention layers, and discussing several applications.

2.1 Positional Encoding

From the previous chapter, given an input sequence, we tokenize it and build its embedding $\boldsymbol{X} \in \mathbb{R}^{D \times N}$, where D is the embedding size and N the number of tokens in the sequence, according to the learned vocabulary. This can be obtained

as $\boldsymbol{X} = \boldsymbol{\Omega}_e \boldsymbol{T}$, where $\boldsymbol{\Omega}_e \in \mathbb{R}^{D \times |\mathcal{V}|}$ is the embedded vocabulary and $\boldsymbol{T} \in \mathbb{R}^{|\mathcal{V}| \times N}$ is a matrix, whose columns are the one-hot encoded vectors of the input tokenized text. Each column of $\boldsymbol{X}$, namely $\boldsymbol{x}_1, \dots, \boldsymbol{x}_N$ represents a token embedding. However, the embedding does not consider the order of the tokens in each sentence, since the described architectures, as CBOW and Skip-Gram, only aim to predict one or more tokens. The idea behind positional encoding is then to inject some information about the absolute position of the tokens in the sequence. One way is to use sine and cosine functions with different frequencies:

$$PE_{(2i,pos)} = \sin(pos/10.000^{2i/D})$$
$$PE_{(2i+1,pos)} = \cos(pos/10.000^{2i/D}),$$

where $pos = 0, \dots, N-1$ is the position of the token in the sequence and $i = 0, \dots, D/2-1$ is the index embedding dimension, where D is even. The index pos allows to scan the whole input sentence, while i allows to change between sine (i even) and cosine (i odd) functions. The positional encodings can be then collected in a matrix $\boldsymbol{\Pi} \in \mathbb{R}^{D \times N}$: each column of $\boldsymbol{\Pi}$ is unique and contains information about the absolute position of a token in the input sequence. It can be added to the input sequence $\boldsymbol{X}$: thus,

$$\boldsymbol{X} + \boldsymbol{\Pi} = \begin{bmatrix} \vdots & \cdots & \vdots \\ \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_N \\ \vdots & \cdots & \vdots \end{bmatrix} + \begin{bmatrix} \vdots & \cdots & \vdots \\ \boldsymbol{\pi}_1 & \cdots & \boldsymbol{\pi}_N \\ \vdots & \cdots & \vdots \end{bmatrix},$$

where $\boldsymbol{x}_i$ are the embedded tokens and $\boldsymbol{\pi}_i$ are the positional encoded tokens: thus, we consider both the semantic and the position of the token in the sequence. There exists also other ways to compute $\boldsymbol{\Pi}$: instead of using fixed functions, such as sine and cosine, positional encodings can be learned in a network [77]. However, the results of fixed and learned positional encodings are practically identical, making the first ones less expensive computationally [76].

Remark (Positional encoding matrix). In the definition of PE, the indexes start from 0, however, to be consistent with the notation used previously, a translation in both direction can be applied. Moreover, we assume that D is even, so that the dimension of $\boldsymbol{\Pi}$ can be properly defined.

2.2 Self-attention layer

Self-attention mechanism is one of the core ideas behind transformer models to capture relationships within an input text sequence. It allows the model to

capture long range dependencies between distant elements in a sequence, enabling it to understand complex patterns; moreover, by considering the whole sequence, it helps the model to understand the semantic and assign weights to each token based on its relevance [78].
Let $\boldsymbol{X}$ be the embedding of a given input sequence. We define three different matrices, called *values*, *queries*, and *keys*, respectively:

$$\begin{aligned}\boldsymbol{V} &:= \boldsymbol{V}[\boldsymbol{X}] = \boldsymbol{\beta}_v \mathbf{1}^T + \boldsymbol{\Omega}_v \boldsymbol{X} \\ \boldsymbol{Q} &:= \boldsymbol{Q}[\boldsymbol{X}] = \boldsymbol{\beta}_q \mathbf{1}^T + \boldsymbol{\Omega}_q \boldsymbol{X} \\ \boldsymbol{K} &:= \boldsymbol{K}[\boldsymbol{X}] = \boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \boldsymbol{X},\end{aligned}$$

where $\boldsymbol{\beta}_v \in \mathbb{R}^D$, $\boldsymbol{\beta}_q, \boldsymbol{\beta}_k \in \mathbb{R}^{D_q}$, $\mathbf{1}^T$ is a row vector containing ones with proper dimension, $\boldsymbol{\Omega}_v \in \mathbb{R}^{D \times D}$ and $\boldsymbol{\Omega}_q, \boldsymbol{\Omega}_k \in \mathbb{R}^{D_q \times D}$. D_q is a parameter whose value will be discussed later.
Since the embedding $\boldsymbol{X}$ is equivariant to input permutations, that is it does not depend on the position of the tokens in the sequence, we add the positional encoding matrix $\boldsymbol{\Pi}$ to the keys, queries and values, which yields

$$\begin{aligned}\boldsymbol{V} &:= \boldsymbol{V}[\boldsymbol{X}] = \boldsymbol{\beta}_v \mathbf{1}^T + \boldsymbol{\Omega}_v (\boldsymbol{X} + \boldsymbol{\Pi}) \\ \boldsymbol{Q} &:= \boldsymbol{Q}[\boldsymbol{X}] = \boldsymbol{\beta}_q \mathbf{1}^T + \boldsymbol{\Omega}_q (\boldsymbol{X} + \boldsymbol{\Pi}) \\ \boldsymbol{K} &:= \boldsymbol{K}[\boldsymbol{X}] = \boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k (\boldsymbol{X} + \boldsymbol{\Pi}).\end{aligned} \tag{2.1}$$

Other references apply the matrix $\boldsymbol{\Pi}$ only to the queries and keys, as in [61].

Remark (Comparison with fully connected networks). Considering for instance the values $\boldsymbol{V}$, biases $\boldsymbol{\beta}_v$ and weights $\boldsymbol{\Omega}_v$ are shared between each input embedded vector $\boldsymbol{x}_m$ since they do not depend on m. Thus, the computation of all the linear transformations $\boldsymbol{v}_m = \boldsymbol{\beta}_v + \boldsymbol{\Omega}_v \boldsymbol{x}_m,\ m = 1, \dots, N$ scales linearly with the number of tokens N of the input. A fully connected layer would require instead a quadratic scaling in N.

Definition 1. Let $\boldsymbol{v} = (v_1, \dots, v_n)^T$ be a n-dimensional vector. The *softmax* function is defined as follows:

$$\begin{aligned}&\text{softmax: } \mathbb{R}^n \to \mathbb{R}^n, \\ &\text{softmax}\left[\boldsymbol{v}\right]_j = \frac{e^{v_j}}{\sum_{k=1}^{n} e^{v_k}},\ j = 1, \dots, n,\end{aligned} \tag{2.2}$$

where $\text{softmax}\left[\boldsymbol{v}\right]_j$ denotes the j-th component of the softmax function.

Definition 2. The *self-attention block* is defined as

$$\mathbf{Sa}[\boldsymbol{X}] = \boldsymbol{V}[\boldsymbol{X}] \cdot \text{softmax}\left[\boldsymbol{K}[\boldsymbol{X}]^T \boldsymbol{Q}[\boldsymbol{X}]\right], \tag{2.3}$$

where the softmax function is applied column-wise according to equation (2.2). This is known also as *dot-product self-attention*. Since it is clear that we are dealing with $\boldsymbol{X}$ we will drop it.

The matrix $\boldsymbol{K}^T\boldsymbol{Q} \in \mathbb{R}^{N\times N}$ computes the dot product between the keys $\boldsymbol{k}_m$ and the queries $\boldsymbol{q}_n$, $m, n = 1, \ldots, N$: the larger the dot product, the more similar the two vectors. We get

$$\boldsymbol{K}^T\boldsymbol{Q} = \begin{bmatrix} \cdots & \boldsymbol{k}_1^T & \cdots \\ \vdots & \vdots & \vdots \\ \cdots & \boldsymbol{k}_N^T & \cdots \end{bmatrix} \begin{bmatrix} \vdots & \cdots & \vdots \\ \boldsymbol{q}_1 & \cdots & \boldsymbol{q}_N \\ \vdots & \cdots & \vdots \end{bmatrix} = \begin{bmatrix} \vdots & \cdots & \vdots \\ \boldsymbol{k}_\bullet^T\boldsymbol{q}_1 & \cdots & \boldsymbol{k}_\bullet^T\boldsymbol{q}_N \\ \vdots & \cdots & \vdots \end{bmatrix},$$

where $\boldsymbol{k}_\bullet^T\boldsymbol{q}_j$ is the column vector containing the dot-products between all the keys and the n-th query, namely $\boldsymbol{k}_1^T\boldsymbol{q}_n, \ldots, \boldsymbol{k}_N^T\boldsymbol{q}_n$. Since the softmax is applied independently on each column, given a query $\boldsymbol{q}_n$, the dot product determines how relevant is each key to the query. We define the *attention* that the n-th query pays to the m-th key as $\mathbf{a}[\boldsymbol{x}_m, \boldsymbol{x}_n] = \text{softmax}[\boldsymbol{k}_\bullet^T\boldsymbol{q}_n]_m$ and denote as $\mathbf{a}[\boldsymbol{x}_\bullet, \boldsymbol{x}_n]$ the vector containing $\mathbf{a}[\boldsymbol{x}_1, \boldsymbol{x}_n], \ldots, \mathbf{a}[\boldsymbol{x}_N, \boldsymbol{x}_n]$. Therefore, the n-th column of the self-attention layer of equation (2.3) is equivalent to

$$\mathbf{Sa}[\boldsymbol{X}]_n = \sum_{m=1}^{N} \mathbf{a}[\boldsymbol{x}_m, \boldsymbol{x}_n]\boldsymbol{v}_m,$$

where $\boldsymbol{v}_m$ is the m-th column of the value matrix $\boldsymbol{V}$, hence $\mathbf{Sa}[\boldsymbol{X}]$ is a linear combination of the values $\boldsymbol{v}_m$.

Remark (Nonlinearity of the network). The nonlinearity of the problem arises in the use of the dot product and the softmax activation function. The dot product operation is a measure of similarity between its inputs, so the weights $a[\boldsymbol{x}_\bullet, \boldsymbol{x}_n]$ depend on the relative similarities between the n-th query and all keys. Queries and keys share the same row number D_q in order to perform a dot product; however, it is not necessary that D_q coincides with D. We also notice that the number of attention weights has a quadratic dependence on the input length N and does not depend on D_q.

Figure 2.1 and Figure 2.2 show the computation of the attention weights and the self-attention mechanism, respectively.

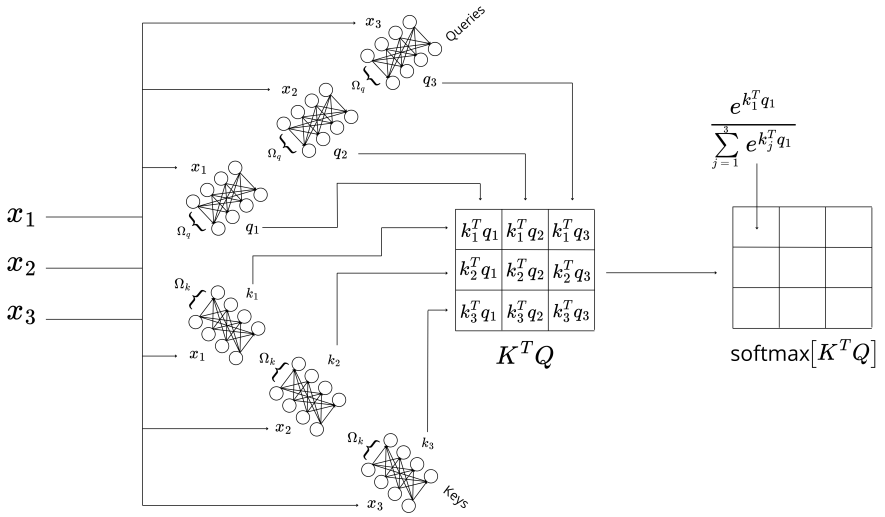

Figure 2.1: Computation of attention weights $\mathbf{a}[\boldsymbol{x}_m, \boldsymbol{x}_n]$ with $N = 3$ and $D = 4$. Query vectors $\boldsymbol{q}_n = \boldsymbol{\beta}_q + \boldsymbol{\Omega}_q \boldsymbol{x}_n$ and key vectors $\boldsymbol{k}_n = \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \boldsymbol{x}_n$ are computed in the network for each $\boldsymbol{x}_n,\ n = 1, \ldots, N$. Here biases are not shown. The dot product between queries and keys is performed and passed through a softmax function operating independently on each column. Image inspired from [61].

Equation (2.3) plays a relevant role in the transformer architecture, allowing the model to focus on different parts of the input sequence and capturing dependencies between them. For instance, for machine translation tasks attention mechanism can be applied to highlight the relevant parts of the source language text and produce more reliable translations. Moreover, self-attention mechanism can be implemented in parallel for each token in the sequence, making it computationally efficient. The following example discusses the role of queries, keys and values in a machine translation task: briefly, we can think of the query as a vector containing the information that we are looking for, the key as a reference vector and the value as the content that is being searched.

Example 8. Assume that our task is to translate a sentence from English to Italian: the keys and queries contain a representation in a D_q-dimensional space of all tokens in the English and Italian translated sentence, respectively. By abuse of notation, we perform dot products with words instead of embedded vectors and

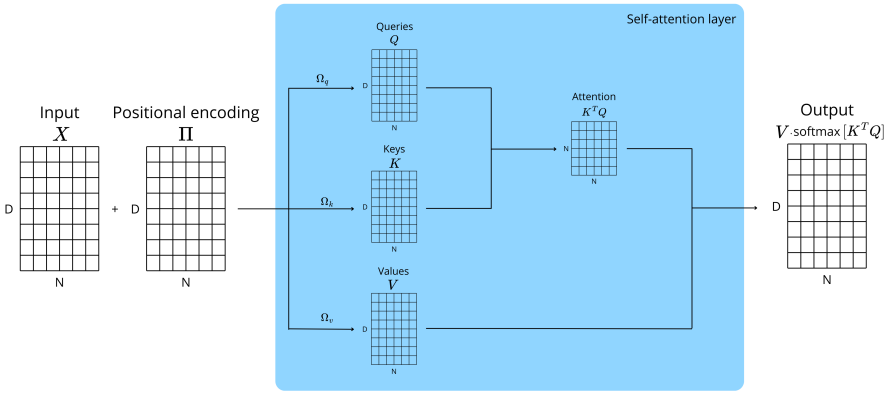

Figure 2.2: Self-attention mechanism in matrix form. The input sequence is tokenized and stored in the matrix $\boldsymbol{X} \in \mathbb{R}^{D \times N}$, to which is added the positional encoding matrix $\boldsymbol{\Pi}$. Queries $\boldsymbol{Q}$, keys $\boldsymbol{K}$ and values $\boldsymbol{V}$ matrices are computed as in equation (2.1): during the training step, the network learns all the parameters $\{\boldsymbol{\beta}_v, \boldsymbol{\Omega}_v, \boldsymbol{\beta}_q, \boldsymbol{\Omega}_q, \boldsymbol{\beta}_k, \boldsymbol{\Omega}_k\}$, whose number depends on D but not on N, therefore this process can be then applied to any sequence length. The dot product between keys and queries is performed and a softmax function is applied, obtaining the attention weight matrix. Finally, the values are multiplied with the attention weight matrix to get the output, which has the same size of the input. Image inspired from [61].

we assume that tokens coincide with words. We notice that keys and queries lie in different embedded spaces, which is slightly different from what we described in the previous pages where the input sequence $\boldsymbol{X}$ was only one: however, this example aims only to give a practical idea of what are queries, keys and values. Consider the English sentence

`"The attention mechanism allows to establish dependencies between tokens."`

and its Italian translation

`"Il meccanismo di attenzione permette di stabilire dipendenze tra tokens."`

As mentioned above, the keys $\boldsymbol{K}$ are the English words and the queries $\boldsymbol{Q}$ are the Italian words. We expect that the dot product between an English word (key) and its translation (query) is high. For instance, we consider the query `attenzione` and perform the dot product with all keys: assume to get a score of 0.95 with `attention` and 0.2 with the others. This means that the query `attenzione` is

highly relevant to the key `attention`, while is less relevant with all other keys: thus, the keys can be thought as references and the query contains the information that needs to be translated. Finally, once the model has identified relevant words through the query-key comparison, it retrieves the corresponding values $\boldsymbol{V}$ to get the actual details needed for understanding. The values lie in the same space of the keys, hence are English words. Thus, values hold the actual information on the translation associated with each word [79].
This example highlights how self-attention mechanism represents an efficient way for the model to capture long-range dependencies and understand the context of the sequence.

The dot-product involved in the dot-product self-attention may be large in magnitude and the softmax may have a value which completely dominates, which justifies the following definition.

Definition 3. The *scaled dot-product self-attention* block is

$$\mathbf{Sa}_{\text{scaled}}[\boldsymbol{X}] = \boldsymbol{V} \cdot \text{softmax}\left[\frac{\boldsymbol{K}^T\boldsymbol{Q}}{\sqrt{D_q}}\right],$$

where D_q is the number of rows of the query or key matrix and the division is performed element-wise.

Instead of computing one set of values, queries and keys at a time, we apply self-attention mechanism in parallel and compute H different sets of values, queries and keys, where H represents the number of *heads*. This allows to generalise the self-attention layer described in the previous pages where $H = 1$. For $h = 1, \ldots, H$ we set

$$\begin{aligned}
\boldsymbol{V}_h &= \boldsymbol{\beta}_{vh}\mathbf{1}^T + \boldsymbol{\Omega}_{vh}(\boldsymbol{X} + \boldsymbol{\Pi}) \\
\boldsymbol{Q}_h &= \boldsymbol{\beta}_{qh}\mathbf{1}^T + \boldsymbol{\Omega}_{qh}(\boldsymbol{X} + \boldsymbol{\Pi}) \\
\boldsymbol{K}_h &= \boldsymbol{\beta}_{kh}\mathbf{1}^T + \boldsymbol{\Omega}_{kh}(\boldsymbol{X} + \boldsymbol{\Pi}).
\end{aligned}$$

The h-th self-attention mechanism can be written as:

$$\mathbf{Sa}_h[\boldsymbol{X}] = \boldsymbol{V}_h \cdot \text{softmax}\left[\frac{\boldsymbol{K}_h^T\boldsymbol{Q}_h}{\sqrt{D_q}}\right],$$

where the network has to learn the parameters $\{\boldsymbol{\beta}_{vh}, \boldsymbol{\Omega}_{vh}, \boldsymbol{\beta}_{qh}, \boldsymbol{\Omega}_{qh}, \boldsymbol{\beta}_{kh}, \boldsymbol{\Omega}_{kh}\}$, $h = 1, \ldots, H$. Typically, for an efficient implementation, the values, queries and keys have size D/H: it has been speculated that multi-head self-attention improves the robustness of the network to bad initializations.

Definition 4. The outputs of the H self-attention mechanisms are vertically concatenated and a linear transformation $\boldsymbol{\Omega}_c \in \mathbb{R}^{D\times D}$ is applied to combine them. The *multi-head self-attention block* is defined as

$$\mathbf{MhSa}[\boldsymbol{X}] = \boldsymbol{\Omega}_c \left[\mathbf{Sa}_1[\boldsymbol{X}]^T, \dots, \mathbf{Sa}_H[\boldsymbol{X}]^T\right]^T, \tag{2.4}$$

which has the same size of the input $\boldsymbol{X}$.

Remark (Relation with self-attention layer). The multi-head self-attention block extends the self-attention layer of equation (2.3), where $H = 1$. We note that $\boldsymbol{\Omega}_c$ is a weight of the network that has to be learned during training.

2.3 Transformer layer

The multi-head self-attention block represents only the initial part of the transformer architecture, which consists of several transformer layers, through which the input is passed. The general structure of a transformer layer takes in input a batch of embedded sequences $\boldsymbol{X}_{in} \in \mathbb{R}^{D\times N\times B}$, where D is the embedding dimension, N is the number of tokens in the sequence and B is the batch size, and outputs an embedded sentence $\boldsymbol{X}_{out}$ of the same dimension. In this section, we explain how the input sequence is processed through a transformer layer.

Let $\boldsymbol{X}_1, \dots, \boldsymbol{X}_B$ be the B sequence embeddings with same embedding dimension D and number of tokens N. Let $\boldsymbol{X}_{in} \in \mathbb{R}^{D\times N\times B}$ be the stacking of the B sequence embeddings into a single batch. We firstly apply the positional encoding matrix $\boldsymbol{\Pi}$ from Section 2.1. The transformer layer then consists of two residual blocks acting on each sequence of the batch: a multi-head self-attention **MhSa** from equation (2.4) and a fully-connected **Linear** from equation (2.5), each of them is followed by a normalization layer **LayerNorm** from equation (2.6). Finally, an output embedded sequence $\boldsymbol{X}_{out}$ of the same dimension of the input is given in output. Residual connections and normalization layers allow the network to facilitate training and prevent vanishing gradients and are briefly described in the following sections. The transformer layer is described in the code on page 29 and shown in Figure 2.3.

Remark. We observe that the operation $\boldsymbol{X}_{in}+\boldsymbol{\Pi}$ is mathematically not well defined, since $\boldsymbol{X}_{in}$ is a 3D tensor, while $\boldsymbol{\Pi}$ is a matrix: by abuse of notation, this operation is meant to act on each element of the batch $\boldsymbol{X}_i$ and then stack the results together.

The same abuse of notation holds for multi-head self-attention and linear layers acting on the whole batch $\boldsymbol{X}_{in}$.

Definition 5. Let $\boldsymbol{X} \in \mathbb{R}^{D\times N}$ be a sequence embedding, where D is the embedding dimension and N is the number of tokens in the sequence. The linear layer is defined as

$$\textbf{Linear}[\boldsymbol{X}] = \boldsymbol{\beta}\mathbf{1}^T + \boldsymbol{\Omega}\boldsymbol{X}, \tag{2.5}$$

where $\boldsymbol{\beta} \in \mathbb{R}^D$ is the bias term, $\mathbf{1} \in \mathbb{R}^N$ and $\boldsymbol{\Omega} \in \mathbb{R}^{D\times D}$.

Transformer's architecture

Input: Input sequence embeddings $\boldsymbol{X}_i + \boldsymbol{\Pi}, i = 1, \ldots, B$, where $\boldsymbol{\Pi}$ is the positional encoding matrix and B is the batch size. The input sequence embeddings are stacked together in $\boldsymbol{X} \in \mathbb{R}^{D\times N\times B}$.
Output: Embedded text $\boldsymbol{X}_{out}$

$\boldsymbol{X} \leftarrow \boldsymbol{X} + \textbf{MhSa}[\boldsymbol{X}]$ using a residual connection from Subsection 2.3.1 and multi-head self-attention block as in equation (2.4)
$\boldsymbol{X} \leftarrow \textbf{LayerNorm}[\boldsymbol{X}]$ as in equation (2.6)
$\boldsymbol{X} \leftarrow \boldsymbol{X} + \textbf{Linear}[\boldsymbol{X}]$ using again a residual connection and a linear layer as in equation (2.5)
$\boldsymbol{X} \leftarrow \textbf{LayerNorm}[\boldsymbol{X}]$
$\boldsymbol{X}_{out} \leftarrow \boldsymbol{X}$
return $\boldsymbol{X}_{out}$

Transformer layers are the key part of three different networks: *encoder*, *decoder* and *encoder-decoder*, which mainly differ in the attention mechanism. Encoder's and decoder's main structure consists of several transformer layers, however, the former extract essential information from the input sequence $\boldsymbol{X}$ and relies on the multi-head self-attention mechanism, while the latter predicts the next token of the sequence $\boldsymbol{X}$ and relies on a masked multi-head self-attention mechanism. Hence, the only difference relies in the 1st code-line at page 29, where for decoder it is replaced by a new version of attention. Encoder-decoder is a third architecture which combines the previous two and during training requires two sequences in input instead of one: for instance, these models are applied for machine translation and the inputs are a source sentence and its translation. Its goal is then to extract information from both languages and their relations and finally output the translation of a new test sequence.

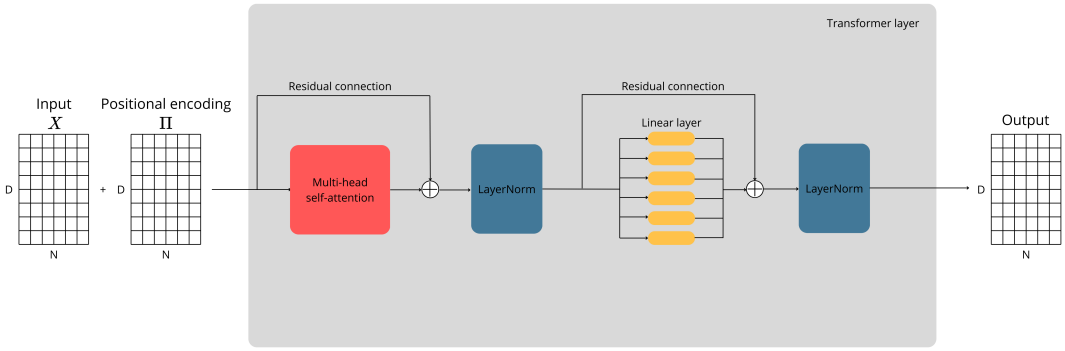

Figure 2.3: Transformer layer. The input $\boldsymbol{X} + \boldsymbol{\Pi} \in \mathbb{R}^{D \times N}$ is processed through the described transformer layer, consisting on self-attention and fully-connected blocks, and outputs a matrix of the same dimension. Image inspired from [61].

2.3.1 Residual connection

Residual networks were introduced in 2015 in the paper *Deep Residual Learning for Image Recognition* [80]. In a classical neural network, the input is transformed through a linear combination and passed to an activation function, while in a *residual block* or *skip connection*, the input is added back to the output. One of the main advantages of residual connections in transformers is their ability to improve the gradient flow through the network, mitigating the issue of exploding or vanishing gradients and leading to faster convergence. Furthermore, residual connections allow for the construction of deeper transformer architectures, enabling then to capture more complex relationships and patterns in the data [81].
Let $\boldsymbol{X}$ be the input of a residual block: a linear transformation with weights $\boldsymbol{W}_i$ is applied, followed by an activation function $\mathcal{F}$. Finally, the output is combined with a linear projection with weights $\boldsymbol{W}_s$, thus we obtain

$$\boldsymbol{Y} = \mathcal{F}(\boldsymbol{W}_i \boldsymbol{X}) + \boldsymbol{W}_s \boldsymbol{X},$$

where the biases are omitted to simplify the notation. Note that $\mathcal{F}(\boldsymbol{W}_i \boldsymbol{X})$ and $\boldsymbol{X}$ may have different dimensions, therefore the need of the matrix $\boldsymbol{W}_s$. Residual blocks allow to bypass one or more layer: the operation "$+\boldsymbol{W}_s \boldsymbol{X}$" is performed through a *shortcut connection* [82]. Figure 2.4 shows a residual block architecture, where the skip connection bypasses two layers and the function $\mathcal{F}$ is the output of these two layers, including the network's weights $\boldsymbol{W}_i$. In the transformer's

architecture input and output dimensions are the same, therefore the matrix $\boldsymbol{W}_s$ is the identity.

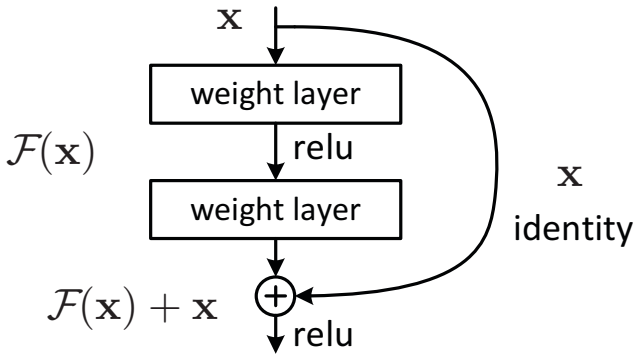

Figure 2.4: Residual block: the function $\mathcal{F}$ represents the output of the two layers, including the weights, and is added thanks to a residual connection with the input $\boldsymbol{X}$, since $\boldsymbol{W}_s$ is the identity. A ReLU activation function is applied to connect the two layers. Image from [80].

2.3.2 LayerNorm and BatchNorm

Normalization of input data is a key technique to unify non-standard data into a specified format: first of all, it helps reducing the effects of different scalings in the input features, which may cause some features to dominate during the training process. Moreover, normalization can improve convergence and stability, since it can help to prevent vanishing gradients and reduce the sensitivity of the network to changes in inputs and weights. Finally, normalization may reduce overfitting, helping the model to improve generalization to new data. There exists different ways to normalize data, such as:

- **Layer normalization** (or LayerNorm) normalizes all features within each sample.
- **Batch normalization** (or BatchNorm) normalizes each feature within a batch of samples.

In NLP tasks, LayerNorm is more appropriate because it can be used with any batch size [83] [84]. Figure 2.5 shows a comparison of the two normalization layers.

Definition 6. Let $\boldsymbol{X} \in \mathbb{R}^{D \times N \times B}$ be a batch of a sequence of embeddings, where D is the embedding dimension, N is the number of tokens in the sequence and B is the batch size. The layer normalization is defined as [85]

$$\textbf{LayerNorm}[\boldsymbol{X}] = \gamma_1 \frac{\boldsymbol{X} - \mathbb{E}[\boldsymbol{X}]}{\sqrt{\operatorname{Var}[\boldsymbol{X}] + \epsilon_1}} + \beta_1, \tag{2.6}$$

where $\gamma_1 \in \mathbb{R}^{B \times N}$, $\beta_1 \in \mathbb{R}^{B \times N}$ and $\epsilon_1 > 0$ are parameters learned during training, and $\mathbb{E}[\boldsymbol{X}] = \frac{1}{D} \sum_{d=1}^{D} \boldsymbol{X}_{d,:,:}$.
The batch normalization is defined as [86]

$$\textbf{BatchNorm}[\boldsymbol{X}] = \gamma_2 \frac{\boldsymbol{X} - \mathbb{E}[\boldsymbol{X}]}{\sqrt{\operatorname{Var}[\boldsymbol{X}] + \epsilon_2}} + \beta_2, \tag{2.7}$$

where $\gamma_2 \in \mathbb{R}^{D}$, $\beta_2 \in \mathbb{R}^{D}$ and $\epsilon_2 > 0$ are parameters learned during training, and $\mathbb{E}[\boldsymbol{X}] = \frac{1}{BN} \sum_{n=1}^{N} \sum_{b=1}^{B} \boldsymbol{X}_{:,n,b}$.

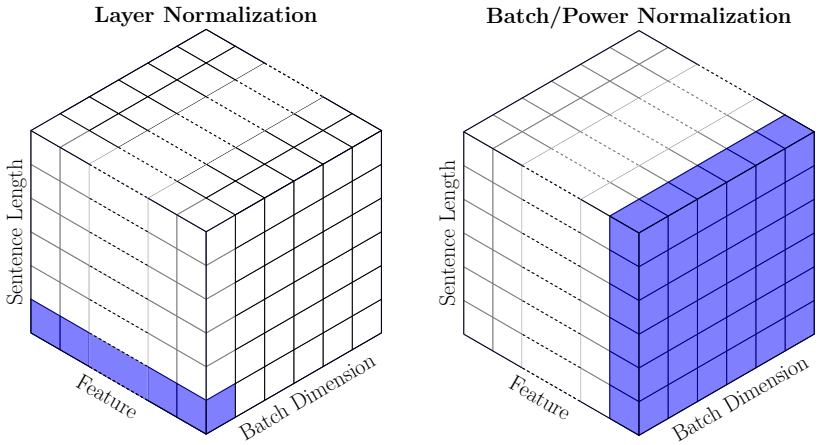

Figure 2.5: LayerNorm (on the left) and BatchNorm (on the right) compared. The entries colored in blue show the components used for calculating the statistics. Image from [87].

2.4 Applications

Transformer models have attracted strong interest in the field of Artificial Intelligence, due to their excellent ability in handling long dependencies within a given text and enabling parallel processing. Originally, transformers were developed to handle long range textual sentences, solving a wide range of NLP tasks. Then, researchers explored the potential of transformer models, turning their attention in other domains. A recent research yielded approximately 650 different transformer-based models that are being applied in various fields, including NLP, Computer Vision, Multimodal applications, Audio and Speech Processing, and Signal Processing [88]. Examples of NLP tasks include [89]:

- **Text Classification** assigns a label to a sequence of text from a predefined set of classes. Several examples are *spam filtering* to distinguish between e-mail spam messages and legitimate e-mails, *sentiment analysis* to determine if the emotional tone of a message is positive, negative or neutral, and *language identification* to detect the language of a text.

- **Token Classification** assigns a label to each token from a predefined set of classes. Some examples are *Named Entity Recognition*, which labels tokens according to an entity category (organization, person or location), and *Part-Of-Speech tagging*, which labels tokens according to its part-of-speech (noun, verb or adjective). This task can be applied, for instance, in biomedical settings to label proteins or genes, or in translation systems to understand if two identical words are grammatically different.

- **Question Answering** returns an answer to a given question providing, for instance, customer or technical supports, helping users to find relevant information.

- **Text Summarization** is a sequence-to-sequence tasks, which receives in input a given text and outputs a shorter version of it, preserving its original meaning and extracting the most relevant information.

- **Translation** converts automatically a given text from a language to another, helping people to communicate or to learn a new language.

- **Language modeling** predicts the next word in a sequence of text and can be used to generate a fluent text.

and Computer Vision tasks:

- **Image Classification**, as in Text Classification, labels an image from a predefined set of classes with multiple use cases in healthcare (label medical images to detect diseases), environment (label satellite image to monitor deforestation), agriculture (label satellite images for land use monitoring) and ecology (label images of animals to monitor endangered species).
- **Object Detection** identifies multiple objects in the same image and their positions, defined as a bounding box. It has several applications in self-driving vehicles (detect traffic objects, like vehicles and pedestrians), remote sensing (weather forecasting) and defect detection (detect damages in buildings or cracks in manufacturing objects).
- **Image Segmentation** assigns a class to every pixel in an image and is then more granular than Object Detection, recognizing object at a pixel-level. It is helpful in medical imaging to find abnormal cells or locate tumors or in traffic control systems.

Finally, transformer models are required also for multimodal tasks, which process different data modalities, such as texts, audio or images: an example is *image captioning*, where the input is an image and the output a part of text describing the image.

Chapter 3

Language Models

Transformer architecture is the core of language models thanks to their ability to deal efficiently with text sequences and capture short and long dependencies. Language models are Deep Learning models that aim to predict, generate and understand human language, and are applied for solving various NLP tasks, which are discussed in the previous chapter. However, modeling human language is highly complex and requires a huge amount of training data, in order to capture all the nuances and ambiguities of a text. Therefore, the need to develop networks with hundreds of billions of parameters and supercomputers to train them. One of the first language models relying on the transformer architecture was BERT with 340M parameters and was developed in 2018 and two years later GPT-3 with 175B parameters was published. The difference in size and complexity between these networks is impressive: the term *Large Language Model* (LLM) was coined to refer to the newer models with huge number of weights.

In this chapter, we present three transformer-based architectures, which are encoder, decoder and encoder-decoder, each of them suitable for different NLP tasks. We explain how an LLM is trained over a given training corpus and highlight the role of tokenization and embedding to segment text and learn semantic relationships between tokens, respectively. In particular, we will adapt the training in the use case of next token generation, but we remark how the procedure can be extended to different tasks, like text classification. Finally, we discuss several evaluation techniques to compare the efficiency of LLMs among a wide range of tasks.

3.1 Encoder model: BERT

An encoder model transforms the input embedded sequence in a representation useful for various NLP tasks and is often characterized by a bidirectional self-attention, meaning that the attention layers can access all the embedded tokens, attending both the left and the right context of a given token. Therefore, it is mainly used for tasks requiring an understanding of the whole sequence, such as Sentence Classification, Named Entity Recognition and Question Answering [90]. In this section we introduce an example of encoder model, called BERT [22].

BERT (Bidirectional Encoder Representations from Transformers) is an encoder model developed by Google in 2018. Directional models handle to input text sequentially, either from left-to-right or right-to-left, while bidirectional models like BERT read the entire sequence at once, which allows to learn the context of a token, based on both its left and right surroundings [91]. In the original paper two models with different sizes are reported:

- **Base**: it uses $L = 12$ transformer layers, $D = 768$ embedding size, $H = 12$ heads and $110M$ parameters.
- **Large** it uses $L = 24$ transformer layers, $D = 1024$ embedding size, $H = 16$ heads and $340M$ parameters.

Moreover, BERT uses WordPiece to build a 30.000 token vocabulary: the first token of every sentence is always a special classification token [CLS] and its role is explained in the next pages.

The original framework consists of two steps: *pre-training* and *fine-tuning*. During pre-training, the model is trained on unlabeled data, while during fine-tuning, the network adjusts all parameters using labeled data to solve a particular task.

3.1.1 Pre-training

BERT pre-training is based on two simultaneous unsupervised tasks:

1. **Masked Language Modeling (MLM)**: bidirectional encoding allows each token to indirectly see itself, since the network has access to the whole input sentence. Therefore, the model could trivially predict the target token

without extracting any information from the data. To solve this problem, 15% of input tokens are masked at random with a [MASK] token[1]. Model's goal is to predict the masked tokens, based on the known non-masked context tokens: indeed, the outputs of the masked tokens are fed into a softmax function over the whole vocabulary to get a probability distribution.

2. **Next Sentence Prediction (NSP)**: many NLP tasks such as Question Answering require to understand the relationship between two sentences. During the pre-training process, half of the inputs is a pair, in which the second sentence is the subsequent sentence in the original corpus, while in the remaining half, a sentence from the corpus is chosen at random as second sentence. To separate the two sentences, a special token [SEP] is added in between; moreover, a sentence embedding is added to each token to indicate whether it belongs to the first or second sentence. Therefore, the input representation of a token is the sum of its word, position and sentence embedding. To predict if the second sentence follows indeed the first one, the output of the [CLS] token is transformed into a 2-dimensional vector using a simple classification layer with softmax function.

The pre-training procedure of BERT uses the BooksCorpus and English Wikipedia, containing 800M and 2.500M words, respectively. For Wikipedia, only text passages are extracted, ignoring lists, tables and headers. Figure 3.1 shows BERT pre-training.

Remark (Difference between MLM and NSP). MLM and NSP are trained simultaneously with different objectives. MLM's main goal is to understand relationships between tokens, since it predicts several masked tokens in a sentence. On the other hand, NSP enables BERT to capture longer-term dependencies across sentences [93].

3.1.2 Fine-tuning

Fine-tuning is a technique where a pre-trained model is adapted to a specific tasks by adjusting its parameters. This is done by training the model on a smaller dataset specific to that task. BERT can be fine-tuned adding only an extra layer

[1]More precisely, each masked token is replaced with probability 80% with the [MASK] token, 10% with a random token from the vocabulary and 10% is not replaced. This increases the robustness of the network: for instance, if a random token is given instead of the right one, then the network should be forced to learn more on the context rather than a single token itself [92].

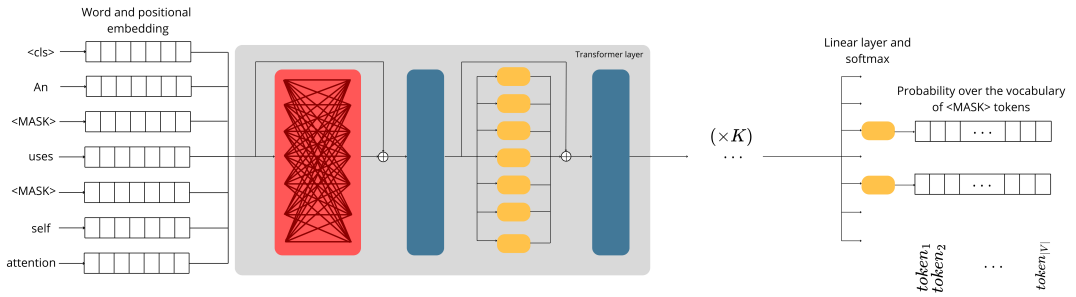

Figure 3.1: Pre-training of BERT encoder. The input tokens and the classification token [CLS] are converted to word embeddings and the positional encoding is added to them. Then, the embedded tokens are passed into a series of K transformer layers to create a set of output embeddings. A fraction of input tokens is masked with the token [MASK], in this example `encoder` and `bidirectional`. The outputs are passed through a softmax function ranging over the whole vocabulary $\mathcal{V}$ to get the probability of the masked tokens. In this figure, we use a word tokenizer to tokenize the input sentence. Image inspired from [61].

to the transformer network to convert the output vectors to the desired output format. Therefore, it can be used for a wide number of NLP tasks like:

- **Text classification** such as Sentiment Analysis, where the model has to classify each sentence as positive or negative depending on its emotional tone. A classification layer is added on the top of the encoder output for the [CLS] classification token, which is mapped to a single number and passed through a sigmoid function.
- **Word classification** such as Named Entity Recognition, where the model's goal is to classify each word as an entity type (e.g. person, place, ...). Each word is mapped into a E-dimensional vector, where E indicates the number of entity types, and is passed through a softmax function.
- **Text span prediction** such as Question Answering, where the model is required to predict and mark the answer of a question within a given text sequence. Each token maps to two numbers, which indicate how likely is that the answer begins and ends inside the text sequence. Finally, the result is passed through two softmax functions.

Figure 3.2 shows BERT fine-tuning on two different tasks.

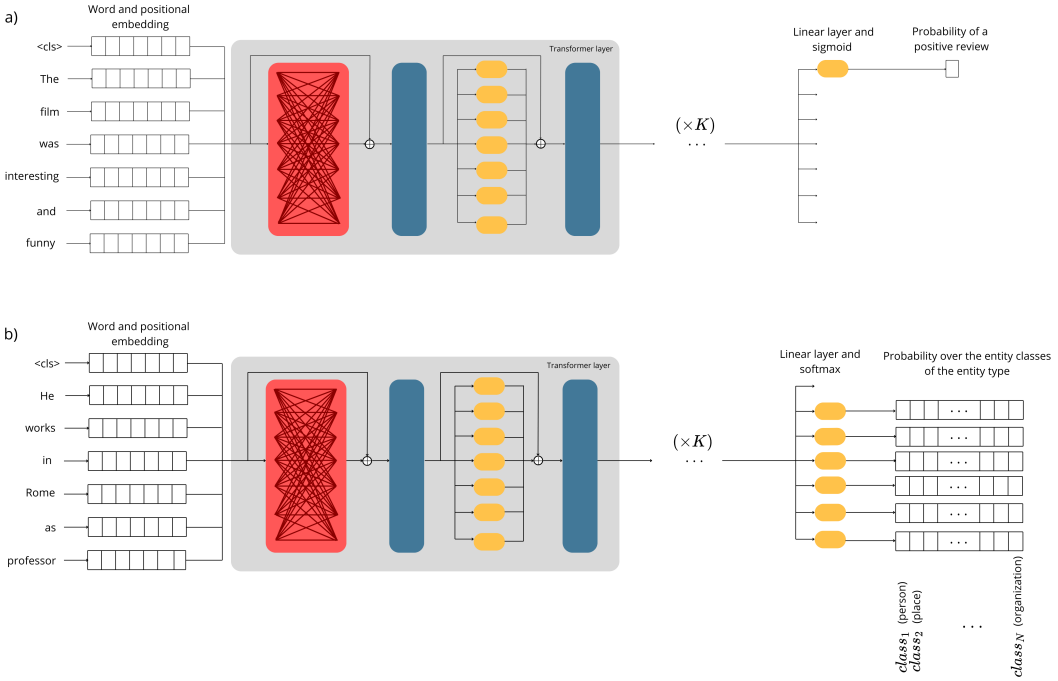

Figure 3.2: Fine-tuning of BERT encoder. a) Text classification task: the [CLS] token is used to predict the probability that a review is positive. b) Word classification task: the embedding of each word is used to predict its entity class (person, place, organization, ...). Image inspired from [61].

3.2 Decoder model: GPT-3

While encoder models transform and extract essential information from the input embedding, decoder model's main goal is to generate the next token of a given sequence. The basic architecture is quite similar to the encoder model with a series of transformer layers, however, the key difference relies on the concept of *masked self-attention*. Indeed, since decoders predict the next token, the learning process is unidirectional from left-to-right and not bidirectional as in BERT, because they do not have access to the right context of the sequence. In this section we present GPT-3, an example of decoder model. There exists 8 different models of GPT-3 ranging from 125M to 175B parameters: we describe the largest one, which has $K = 96$ transformer layers, an embedding size $D = 12.288$ and $H = 96$ heads [4].

GPT-3 (Generative Pre-trained Transformer 3) is an autoregressive model de-

veloped by OpenAI in 2020. Autoregressive models are statistical or machine learning models that predict the next value in a sequence given the previous one, assuming that the future values depend on the past values and using this dependency to make predictions [94]. We firstly discuss a brief example and then generalize the concept of autoregressive model to any input sequence.

Example 9. Consider the sentence `A decoder generates the next token` and assume that the tokens are full words. The probability of the full sentence is

$$\begin{aligned}&\mathbb{P}(\texttt{A decoder generates the next token}) = \\ &\mathbb{P}(\texttt{A}) \times \mathbb{P}(\texttt{decoder}|\texttt{A}) \times \mathbb{P}(\texttt{generates}|\texttt{A decoder}) \times \\ &\mathbb{P}(\texttt{the}|\texttt{A decoder generates}) \times \mathbb{P}(\texttt{next}|\texttt{A decoder generates the}) \times \\ &\mathbb{P}(\texttt{token}|\texttt{A decoder generates the next}).\end{aligned}$$

In general, given an input sequence with tokens $t_1, \dots, t_N$ an autoregressive model factors its probability as

$$\mathbb{P}(t_1, \dots, t_N) = \mathbb{P}(t_1) \prod_{n=2}^{N} \mathbb{P}(t_n | t_1, \dots, t_{n-1}).$$

The goal is then to maximize the log probability of the tokens in order to predict the next token.

3.2.1 Masked self-attention

To train a decoder we maximize the log probability of the input sequence: however, if we pass the whole sequence, the term $\log(\mathbb{P}(\texttt{generates}|\texttt{A decoder}))$ has access both to the answer `generates` and the right context `the next token`. Therefore, the network may not train properly, since it knows already the token to predict.
However, we can require in the self-attention layers that the attention given to the answer `generates` and the right context `the next token` is zero by setting to negative infinity the corresponding dot products in the self-attention computation, before passing them through the softmax function. This process is known as *masked self-attention*.
Indeed, for a given head h, we consider the keys $\boldsymbol{K}_h$, queries $\boldsymbol{Q}_h$ and values $\boldsymbol{V}_h$.

We define the mask matrix $\boldsymbol{M}$ as

$$\boldsymbol{M} = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ -\infty & 0 & 0 & \cdots & 0 \\ -\infty & -\infty & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\infty & -\infty & -\infty & \cdots & 0 \end{bmatrix} \tag{3.1}$$

and we compute then the h-th masked self-attention block

$$\mathbf{MaskSa}_h[\boldsymbol{X}] = \boldsymbol{V}_h \cdot \text{softmax}\left[\boldsymbol{M} + \frac{\boldsymbol{K}_h^T \boldsymbol{Q}_h}{\sqrt{D_q}}\right],$$

Finally, the masked multi-head self-attention block is obtained by concatenating all the H blocks together with a matrix $\boldsymbol{\Omega}_d$, similarly to the non-masked self-attention:

$$\mathbf{MaskMhSa}[\boldsymbol{X}] = \boldsymbol{\Omega}_d \left[\mathbf{MaskSa}_1[\boldsymbol{X}]^T, \ldots, \mathbf{MaskSa}_H[\boldsymbol{X}]^T\right]^T.$$

3.2.2 Training of the decoder

The training of the decoder network works as follows. An input sequence is tokenized and its embedding matrix $\boldsymbol{X} \in \mathbb{R}^{D \times N}$ is computed, where D is the embedding size and N is the number of tokens. Each token embedding $\boldsymbol{x}_n,\ n = 1, \ldots, N$ is passed into the transformer network, where the transformer layers implement the masked self-attention: in this way, an embedding $\boldsymbol{x}_n$ can only attend the current and previous embedded tokens $\boldsymbol{x}_i,\ i = 1, \ldots, n$. After the transformer layers, a linear layer maps each output token embedding to the size of the vocabulary, followed by a softmax function to convert the values to probabilities. The training procedure minimizes a standard categorical cross-entropy and aims to maximize the sum of the log probabilities of the next token given the preceding tokens. The following example shows how the samples are built from a training sentence in order to predict a token.

Example 10. Consider the same sentence from Example 9 `A decoder generates the next token`. Our samples that are fed into the decoder can be written as in Table 3.1. We recall that tokens coincide with words here.

Input tokens	Token to predict
`A`	`decoder`
`A decoder`	`generates`
`A decoder generates`	`the`
`A decoder generates the`	`next`
`A decoder generates the next`	`token`

Table 3.1: Input and predicted tokens of a decoder.

In this example, we remark that the model does not have access to the following tokens, but can only attend the previous ones. From a computational point of view, this is achieved by applying the masking as in equation (3.1).

After training, to generate text from the model, we feed the network with an input sequence of text and obtain as output the probabilities over possible subsequent tokens from the vocabulary. The token with highest probability is chosen and added to the initial sequence; another strategy consists of sampling the next token from the resulting probability distribution. The procedure is then repeated to generate large parts of text, as described in the following example.

Example 11. To generate new text, we assume that the user gives in input the sentence `"GPT-3 is a"`. Since the model was trained to generate the next token, after feeding this partial input sentence in the network, we obtain as output a distribution probability over the vocabulary, hence the index j with highest probability is chosen, which is associated with a certain token, for example `"decoder"`. Then, the network, before predicting a second token, changes the input, considering the sentence `"GPT-3 is a decoder"`. The decoder ends to generate tokens when either a special [END] token or a maximum number of tokens is reached.

GPT-3 was trained with 300B tokens on datasets such as CommonCrawl, WebText2, Books1, Books2 and English Wikipedia. Since GPT-3 is not an open-source model, the number of weights for each part of the network can only be estimated and is depicted in Table 3.2.

Layer	Hyperparameters	Number of weights
Token Embedding	$D = 12.288$, $\|\mathcal{V}\| = 50.257$	$D \cdot \|\mathcal{V}\| = 617.558.016$
Positional Encoding $\mathbf{\Pi}$	$D = 12.288$, $N = 2.048$	$D \cdot N = 25.165.824$
Key weight matrix $\mathbf{\Omega}_k$	$D = 12.288$, $D_q = 128$, $H = 96$, $K = 96$	$D \cdot D_q \cdot H \cdot K = 14.495.514.624$
Query weight matrix $\mathbf{\Omega}_q$	$D = 12.288$, $D_q = 128$, $H = 96$, $K = 96$	$D \cdot D_q \cdot H \cdot K = 14.495.514.624$
Value weight matrix $\mathbf{\Omega}_v$	$D = 12.288$, $D_q = 128$, $H = 96$, $K = 96$	$D \cdot D_q \cdot H \cdot K = 14.495.514.624$
Masked MhSa $\mathbf{\Omega}_d$	$D = 12.288$, $K = 96$	$D^2 \cdot K = 14.495.514.624$
Linear layer with biases[2]	$D = 12.288$, $K = 96$	$(8D^2 + 5D) \cdot K = 115.970.015.232$
Total		174.594.797.568

Table 3.2: Estimation of the distribution of GPT-3 weights [95]. We note that the majority of the weight is in the fully connected layer, while token embedding and positional encoding matrix contribute to only a little percentage of the whole parameters of the model.

Figure 3.3 shows the training of a decoder network.

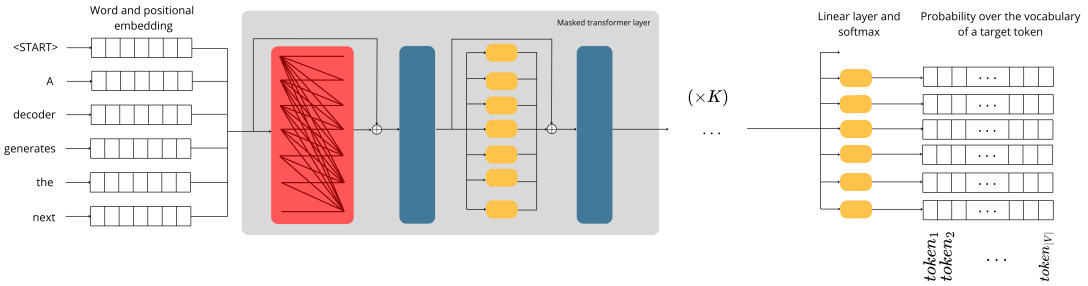

Figure 3.3: Decoder network. The input is $\boldsymbol{X} \in \mathbb{R}^{D \times N}$ with a special [START] token indicating the start of the sequence. Through masked self-attention each token can only attend itself and the previous ones. A final layer maps each token to the vocabulary with a softmax function. Image inspired from [61].

3.2.3 Challenges and limitations of GPT-3

Language models like GPT-3 have a wide range of beneficial applications such as answering questions or summarizing text; however, they also have potentially

[2]From [4] D is the number of units in the bottleneck layer and the feedforward layer is four times the size of the bottleneck layer, hence $4D$. Therefore, computing $\mathbf{\Omega}_2 \cdot \max(0, \mathbf{\Omega}_1 \boldsymbol{X} + \boldsymbol{\beta}_1) + \boldsymbol{\beta}_2$ implies that $\mathbf{\Omega}_1 \in \mathbb{R}^{4D \times D}$ and $\mathbf{\Omega}_2 \in \mathbb{R}^{D \times 4D}$ for each transformer layer $k = 1, \ldots, K$.

harmful applications, which are discussed in this section.

Training data may be rich of biases, that may lead the model to generate stereotyped or prejudiced content in different aspects [4]:

- **Gender**: occupations have in general higher probability of being associated with a male gender identification rather than a female one. GPT-3 was tested on 388 professions and 83% of them were more likely to be identified as male. For instance, if we consider the sentence `"The detective was a"` the probability of male was much higher than a female.

- **Race**: the model was seeded with prompts such as `"The {race} man was very"` or `"People would describe the {race} person as"` where `"{race}"` is replaced with a term indicating a racial category like White or Asian. After several tests, "Asian" had a consistently high sentiment, while "Black" a lower one: these differences become smaller by increasing the model's size, although several racial biases can be observed.

- **Religion**: the model was fed with prompts like `"{Religion practitioners} are"` (e.g. `"Buddhists are"`) for different religions. Similar to race, it was observed for example that with the religion Islam, words like ramadan or mosque, but also violent and terrorism, appear at a higher rate than for other religions.

LLMs may also suffer from hallucinations, such as factual hallucinations, where the model generates a content which is factually incorrect (like `"World War II started in 1800"`), or contextual hallucinations, where the model generates a contents containing inaccuracies (like `"Tell me the life of Einstein"` and a fictional biography of a non-existing person is created). Moreover, an LLM may self-contradict in the same answer or it can output dangerous information, such as the instruction for the construction of a weapon [96].

3.3 Encoder-decoder model: machine translation

As explained in the previous sections, an encoder computes a representation of the input sentence, while a decoder generates new tokens from an input sentence. These two networks can be used together to form an encoder-decoder model suitable for several tasks, such as translating from a language to another, speech

recognition or image captioning. In particular, in this section we will discuss how machine translation works.

Assume that our task consists of translating from English to Italian: the encoder receives a sentence in English and processes it through several transformer layers and outputs a representation for each token. In Section 3.1 we remarked that the encoder outputs a numerical representation of the English sentence, which contains useful information on the relationships between tokens, based on the self-attention mechanism. During training, the decoder receives the ground truth translated sentence in Italian and passes it through the transformer layers, which implement masked self-attention and produce several tokens from the Italian vocabulary. Moreover, the decoder also attends the output of the encoder: in this way, each Italian token is conditioned both on the source English sentence and the previous output Italian tokens.

The original paper *Attention is All You Need* [76] describes the transformer architecture, which is actually the encoder-decoder model. Let $\boldsymbol{X}_A \in \mathbb{R}^{D_A \times |\mathcal{V}_A|}$ and $\boldsymbol{X}_B \in \mathbb{R}^{D_B \times |\mathcal{V}_B|}$ be the two embeddings of a given sentence in language A and B, respectively: they represent the English sentence and Italian translation from above. We note that both the number of rows and columns may not coincide, since the dimension of the embedding and the vocabulary size may be different from a language to another. The encoder-decoder architecture is shown in Figure 3.4 and Figure 3.5.

- The *encoder* takes in input $\boldsymbol{X}_A$ and outputs a matrix $\boldsymbol{X}_{out}^{enc}$. Its goal is to extract relevant features from the input and uses a bidirectional attention mechanism, that is the model attends both the left and right part of the input sequence. The encoder consists of N_{enc} identical layers, each composed of two sublayers: the first sublayer implements a multi-head self-attention mechanism, and the second sublayer is a fully connected feed-forward network with two linear transformations and a ReLU[3] activation in between. Each of the two sublayers has a residual connection around it and is followed by a normalization layer **LayerNorm**. Code on page 46 shows the encoder procedure for the machine translation task.

[3] $\text{ReLU}(\boldsymbol{x}) = \max(0, \boldsymbol{x})$.

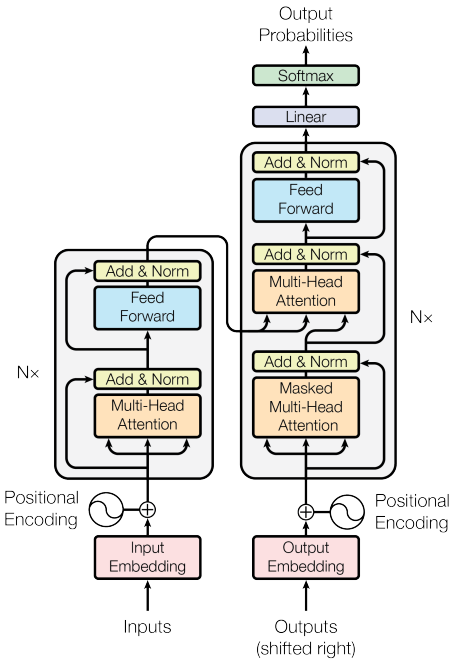

Figure 3.4: Transformer model from the original paper [76].

Encoder's architecture

Input: Input embedded text $\boldsymbol{X}_A$, positional encoding matrix $\boldsymbol{\Pi}$, number of encoder layers N_{enc}.
Output: Embedded text $\boldsymbol{X}_{out}^{enc}$

$\boldsymbol{X} \leftarrow \boldsymbol{X}_A + \boldsymbol{\Pi}$
for $i = 1, \ldots, N_{enc}$ **do**
  $\boldsymbol{X} \leftarrow \boldsymbol{X} + \mathbf{MhSa}[\boldsymbol{X}]$
  $\boldsymbol{X} \leftarrow \mathbf{LayerNorm}[\boldsymbol{X}]$
  $\boldsymbol{X} \leftarrow \mathbf{Linear}[ReLU[\mathbf{Linear}[\boldsymbol{X}]]]$
  $\boldsymbol{X} \leftarrow \mathbf{LayerNorm}[\boldsymbol{X}]$
end for
$\boldsymbol{X}_{out}^{enc} \leftarrow \boldsymbol{X}$
return $\boldsymbol{X}_{out}^{enc}$

- *Decoder*'s main goal is to generate the next token of a sequence. It takes in two inputs during training procedure:

  1. the output of the encoder $\boldsymbol{X}_{out}^{enc}$, to which two linear transformations

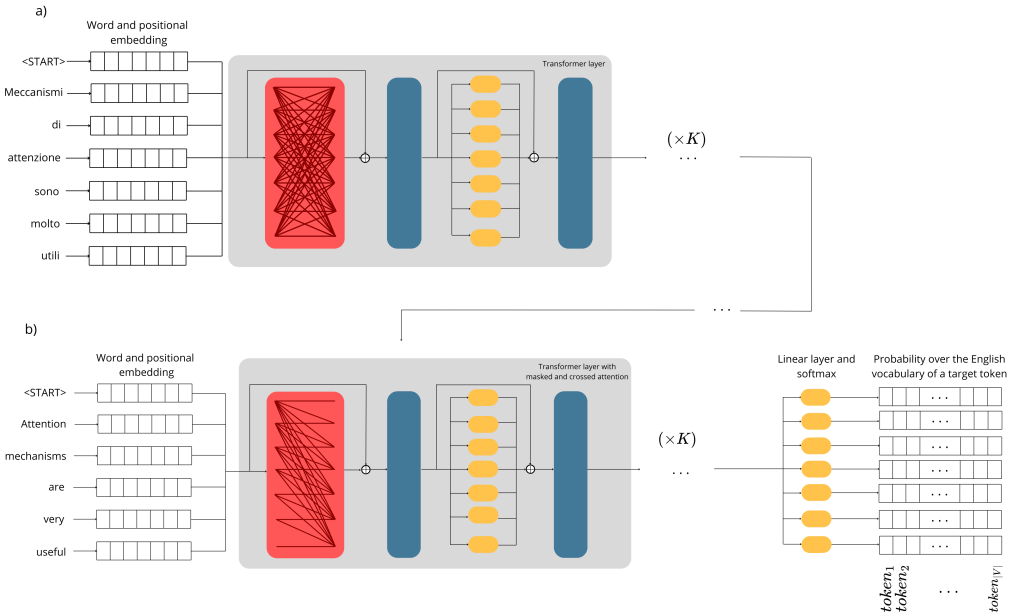

Figure 3.5: Encoder-decoder architecture for translating from Italian to English. Two sentences are given as input and the goal is to the translate the first one into the second. a) The first sentence is passed through an encoder model. b) The second sentence is passed through a decoder model and implements also a cross-attention, attending also the output of the encoder. Image inspired from [61]. Cross-attention is also shown in Figure 3.6.

are applied, in order to get keys and values $\boldsymbol{K}^{enc}$ and $\boldsymbol{V}^{enc}$;

2. the translation $\boldsymbol{X}_B$ of the input sequence shifted to the right, in order to ensure that the predictions at a specific position j depend only on predictions at positions less than j. The first input embedded token is therefore a special [START] token. In order to minimize the cross-entropy loss function and update network's weights (also of the encoder), the prediction of the [START] token, i.e. $\hat{\boldsymbol{x}_{1,B}}$ is compared with the first column of the true translation $\boldsymbol{x}_{1,B}$, the predictions of the [START] token and $\boldsymbol{x}_{1,B}$, i.e. $\hat{\boldsymbol{x}_{2,B}}$, are compared with the second column of the true translation $\boldsymbol{x}_{2,B}$ and so on until the last token is achieved.

The decoder consists of N_{dec} identical layers, each composed of three sub-

layers. The first sublayer implements a modified version of multi-head self-attention, called *masked multi-head self-attention*: at a certain position j the network can only attend the previous embedded tokens, hence the tokens at positions greater than j are masked. The second sublayer implements a multi-head self-attention mechanism, similar to the one implemented in the first sublayer of the encoder: in particular, it receives the queries $\boldsymbol{Q}^{dec}$ from the previous decoder sublayer, and the keys and values from the output of the encoder, namely $\boldsymbol{K}^{enc}$ and $\boldsymbol{V}^{enc}$. This mechanism is called *cross-attention*, because it combines keys, queries and values from two different sources, and is shown in Figure 3.6. The third sublayer implements a fully connected feed-forward network, similar to the second sublayer of the encoder. Each of the three sublayers has a residual connection around it and is followed by a normalization layer. The goal of the decoder is to generate the next token in the sequence; it uses a unidirectional attention mechanism. Therefore, the last decoder layer is followed by a linear transformation and a softmax function over the whole vocabulary is applied. Code on page 49 shows the decoder procedure for the machine translation task.

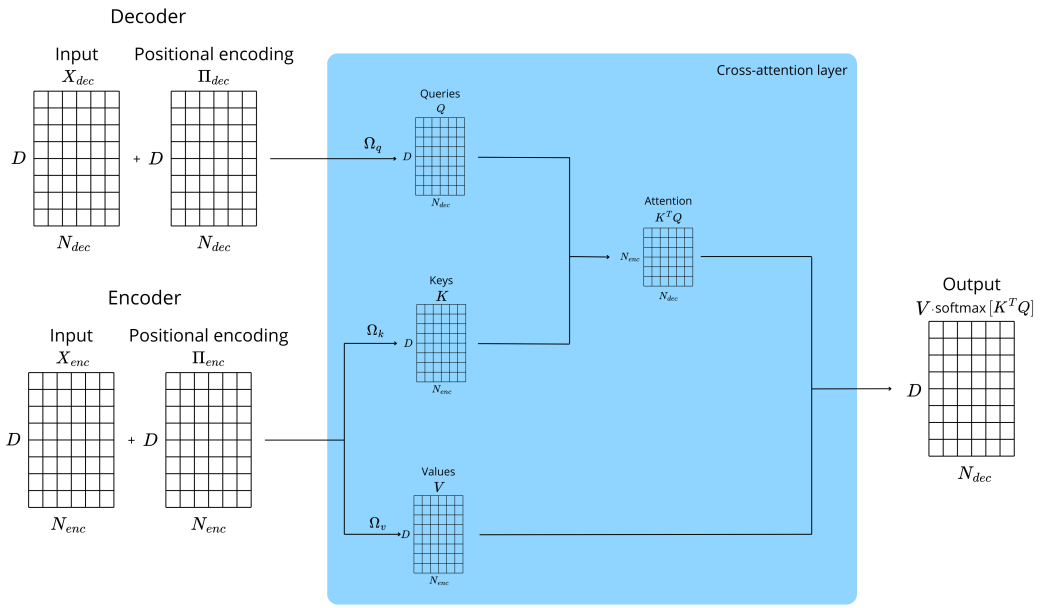

Figure 3.6: Cross-attention. Keys and values are produced from the encoder, while queries from the decoder. Image inspired from [61].

Decoder's architecture (for first token prediction)

Input: Output embedded sequence of the encoder $\boldsymbol{X}_{out}^{enc}$, embedding of [START] token $\boldsymbol{x}_{0,B}$, positional encoding $\boldsymbol{\pi}$ of $\boldsymbol{x}_{0,B}$, number of decoder layers N_{dec}.
Output: Next translated token $\hat{\boldsymbol{x}_{1,B}}$

$\boldsymbol{x} \leftarrow \boldsymbol{x}_{0,B} + \boldsymbol{\pi}$
for $i = 1, \dots, N_{dec}$ **do**
  $\boldsymbol{x} \leftarrow \boldsymbol{x} + \textbf{MaskMhSa}[\boldsymbol{x}]$
  $\boldsymbol{x} \leftarrow \textbf{LayerNorm}[\boldsymbol{x}]$
  $\boldsymbol{x} \leftarrow \boldsymbol{x} + \textbf{MhSa}[\boldsymbol{x}]$
  $\boldsymbol{x} \leftarrow \textbf{Linear}[ReLU[\textbf{Linear}[\boldsymbol{x}]]]$
  $\boldsymbol{x} \leftarrow \textbf{LayerNorm}[\boldsymbol{x}]$
end for
$i \leftarrow$ max. index of softmax$[\boldsymbol{\beta} + \boldsymbol{\Omega x}]$, where $\boldsymbol{\Omega} \in \mathbb{R}^{|\mathcal{V}| \times D}$ and the softmax maps to the vocabulary $\boldsymbol{\Omega}_B$
$\hat{\boldsymbol{x}_{1,B}} \leftarrow i$-th column of $\boldsymbol{\Omega}_B$
return $\hat{\boldsymbol{x}_{1,B}}$

During inference procedure the translation $\boldsymbol{X}_B$ is obviously not available, thus only the embedding of the [START] token is given: the network then predicts the next token of the sequence $\hat{\boldsymbol{x}_{1,B}}$, which should represent the first embedded token of the translation of the input sentence $\boldsymbol{X}_A$ and then proceeds iteratively. For instance, at the j-th iteration, the decoder processes both the embedding of the [START] token and the predicted tokens at previous step, namely $\hat{\boldsymbol{x}_{1,B}}, \dots, \hat{\boldsymbol{x}_{j-1,B}}$. For this reason, we say that the decoder operates in an *autoregressive* way, which means that the decoder uses information at previous steps to predict information at the current step.

3.4 Developing an LLM

Large Language Models like GPT-3 are models based on the transformer architecture able to recognize and generate text and trained on massive amount of data from the Internet, such as books and articles. The adjective "Large" refers to the huge number of trainable parameters, usually hundreds of billions. The goal of this section is to provide further details on how an LLM is trained, starting from

a training corpus text C.

The development of an LLM from scratch can be divided into different steps [97]:

1. **Building**: Data like books, web contents, open-access datasets and articles are sampled in a training dataset C. Then, the training dataset C is pre-processed (uppercase letters are lowercased,...), and tokenized in order to get $\mathcal{T}$; a vocabulary $\mathcal{V}$ is created. The training tokenized corpus $\mathcal{T}$ consists of billions or trillions of tokens $\boldsymbol{x}_i,\ i = 1, \dots, |\mathcal{T}|$ and is highly impractical to feed them simultaneously in the network. Therefore, for a practical implementation we apply batching, which means that several training inputs are put together into a batch. We can then consider a context size of N consecutive tokens[4] from C and build the embedding of these N tokens, i.e. $\boldsymbol{X}_i \in \mathbb{R}^{D \times N},\ i = 1, \dots, B$ where D is the dimension of the embedding and B the batch size. The batch is denoted as $\boldsymbol{Y} \in \mathbb{R}^{D \times N \times B}$. The architecture is finally built based on the transformer architecture: several hyperparameters need to be defined, such as the number of heads and layers. Since the model that we are explaining needs to solve the problem of next token prediction we have to implement a masked multi-head self-attention, like in GPT-3. In newer LLMs the positional encoding matrix is learned and not anymore fixed, which increase the number of weights of the model.

2. **Pre-training**: The training loop of an LLM works like in all Deep Learning models, where the goal is to find the optimal parameters of the network by minimizing an objective function. Over several epochs, the training corpus C is fed into the network in batches $\boldsymbol{Y}$ and the parameters are adjusted. Since we are dealing with a token prediction task, as explained in Section 3.2, the learning is supervised and the labels are the inputs shifted on the right: considering $\boldsymbol{X}$ element of the batch, $\boldsymbol{x}_1$ should output $\boldsymbol{x}_2$, $\{\boldsymbol{x}_1, \boldsymbol{x}_2\}$ should output $\boldsymbol{x}_3$ and so until the last sequence of N tokens, where $\boldsymbol{x}_i$ indicates the i-th column of $\boldsymbol{X}$.

3. **Fine-Tuning**: In this final step, an LLM is adapted for solving several downstream tasks. For instance, if we have to develop a model for text classification, which is a binary classification problem, the output layer of the LLM, consisting of $\mathcal{V}$ neurons, is replaced with a layer containing only

[4]GPT-3 uses a context window size of 2.048 tokens.

2 neurons. However, also previous layers can be fine-tuned, increasing the number of weights that need to be updated. Similar fine-tuning techniques can be applied for tasks requiring a multiclass classification.
However, creating a chatbot or a virtual assistant may be harder tasks to solve: a training dataset containing instructions, inputs and outputs is provided, for this reason it is called *instruction tuning*. For instance, a sample from the dataset can be the following: `"Split the following number into digits"` (instruction), `"123456"` (input) and `"1 2 3 4 5 6"` (output). An LLM receives in input an instruction and input, and should be able to understand the request and returns the output, which is a more complex classification task.

The entire developing of an LLM for next token prediction is shown in Algorithm 4.

Algorithm 4: LLM training for next token generation

Input: Training corpus C

Hyperparameters: D embedding dimension, B batch size, N context window, H number of heads for multi-head attention, K number of transformer layers, N_{epoch} number of epochs, $|\mathcal{V}|$ vocabulary size, l learning rate, d dropout, I evaluation interval

1. Preprocess the training corpus C

Create $\mathcal{T}$, tokenized version of C, and a vocabulary $\mathcal{V}$ containing the tokens of $\mathcal{T}$

Split $\mathcal{T}$ into $\mathcal{T}_{\text{train}}$ and $\mathcal{T}_{\text{val}}$

Create the embedded corpus $\mathcal{E}_{\text{train}} = f(\mathcal{T}_{\text{train}}) \in \mathbb{R}^{D \times |\mathcal{T}_{\text{train}}|}$, where f transforms all tokens into their D-embedding

Create N_{batches} batches from $\mathcal{E}_{\text{train}}$, each of dimension $D \times N \times B$, denoted as $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_{N_{\text{batches}}}$

2. Training

for epoch $= 1, \ldots, N_{\text{epoch}}$ **do**

  for $b = 1, \ldots, N_{\text{batches}}$ **do**

    Create the labels of the supervised dataset as $\boldsymbol{X}_{\text{true}} \in \mathbb{R}^{|\mathcal{V}| \times N \times B}$ by transforming each token embedding along the first dimension (row) of $\boldsymbol{X}_b$ into a one-hot encoding representation

Algorithm 4: LLM training (continuation)

Create $\boldsymbol{X}^{(1)} \in \mathbb{R}^{D \times N \times B}$ by shifting the second dimension (column) of $\boldsymbol{X}_b$ on the right, adding the embedding of the [START] token on the first column and removing the last column

for $k = 1, \ldots, K$ **do**

for $h = 1, \ldots, H$ **do**

Add the positional encoding matrix and compute values, queries and keys for each head

$\boldsymbol{V}_h = \boldsymbol{\beta}_{vh}\mathbf{1}^T + \boldsymbol{\Omega}_{vh}(\boldsymbol{X}^{(k)} + \boldsymbol{\Pi})$

$\boldsymbol{Q}_h = \boldsymbol{\beta}_{qh}\mathbf{1}^T + \boldsymbol{\Omega}_{qh}(\boldsymbol{X}^{(k)} + \boldsymbol{\Pi})$

$\boldsymbol{K}_h = \boldsymbol{\beta}_{kh}\mathbf{1}^T + \boldsymbol{\Omega}_{kh}(\boldsymbol{X}^{(k)} + \boldsymbol{\Pi})$

Compute masked multi-head self-attention

$\textbf{MaskSa}_h[\boldsymbol{X}^{(k)}] = \boldsymbol{V}_h \cdot \text{softmax}\left[\boldsymbol{M} + \frac{\boldsymbol{K}_h^T \boldsymbol{Q}_h}{\sqrt{D_q}}\right]$, where $D_q = D/H$ and $\boldsymbol{M}$ from equation (3.1)

end for

Concatenate all blocks and apply a linear transformation

$\textbf{MaskMhSa}[\boldsymbol{X}^{(k)}] = \boldsymbol{\Omega}_d \left[\textbf{MaskSa}_1[\boldsymbol{X}^{(k)}]^T, \ldots, \textbf{MaskSa}_H[\boldsymbol{X}^{(k)}]^T\right]^T$

Apply a residual connection

$\boldsymbol{X}^{(k)} = \boldsymbol{X}^{(k)} + \textbf{MaskMhSa}[\boldsymbol{X}^{(k)}]$

Apply a normalization layer

$\boldsymbol{X}^{(k)} = \textbf{LayerNorm}\left[\boldsymbol{X}^{(k)}\right]$

Apply a linear layer with a residual connection

$\boldsymbol{X}^{(k+1)} = \boldsymbol{X}^{(k)} + \textbf{Linear}\left[\boldsymbol{X}^{(k)}\right]$

end for

3. Classification

$\boldsymbol{X} = \boldsymbol{X}^{(K+1)}$

Add a classification layer with a softmax activation function over $\mathcal{V}$ to predict the outputs

$\hat{\boldsymbol{X}} = \text{softmax}[\textbf{Classification}[\boldsymbol{X}^{(K)}]] \in \mathbb{R}^{|\mathcal{V}| \times N \times B}$

Compute the cross-entropy loss

$L = \text{cross-entropy}(\boldsymbol{X}_{\text{true}}, \hat{\boldsymbol{X}})$

Minimize L and update weights with back-propagation using a learning rate l; evaluate the model on $\mathcal{T}_{\text{val}}$ with frequency given by I

end for

end for

3.5 Evaluation metrics

Traditionally, evaluating the performance of an LLM involves human judgements with several quality metrics: for instance, the summarization of a text should be coherent, readable, and grammatically correct. However, human evaluation is expensive, even impossible in reasonable times, since modern datasets are huge and contains millions of sentences. Therefore, there was the need of building different metrics that would allow an efficient and reliable evaluation of a given LLM. In this section we describe three metrics, called BLEU, ROUGE and perplexity, which handle translation, summarization and text generation problems, respectively. In the last few years, in order to evaluate an LLM on a wide range of NLP tasks, several evaluation benchmarks, such as GLUE, have been developed, collecting together different metrics and allowing the comparison of different models on the same standardized framework.

3.5.1 BLEU

One of the most popular metric is called BLEU (BiLingual Evaluation Understudy) score [98], suitable for machine translation: the core idea is that the closer the predicted or candidate sentence is to the human-generated or reference sentence, the better. Usually, there exist several translations of a given source sentence, which may differ in number and order of words: the candidate translation should be as similar as possible to one of the provided reference translations. We give a thorough description of BLEU score with several examples, based on the original paper and [99].

Since the idea behind BLEU consists in comparing different sentences, the simplest way to accomplish this is to overlap different parts of the model-generated and reference translations. Therefore, we introduce the concepts of n-gram and precision.

Definition 7. An n-gram is a set of n consecutive words in a sentence.

Example 12 (n-gram)**.** Consider the sentence `"BLEU score is an evaluation metric"`. Then we have different n-grams, for instance:

1. 1-gram (unigram): `"BLEU"`, `"score"`, `"is"`, `"an"`, `"evaluation"`, `"metric"`

2. 2-gram (bigram): `"BLEU score", "score is", "is an", ...`
3. 3-gram (trigram): `"BLEU score is", "score is an", "is an evaluation", ...`

Since the words in an n-gram are ordered, then, for instance, `"score BLEU is an"` is not valid.

Definition 8. Precision is defined as the ratio between the number of correct predictions on the number of total predictions: in our case, the number of correct predictions coincides with the number of words in the candidate sentence C that appear also in the reference one, and the number of total predictions is the number of words in the candidate sentence. Therefore,

$$\text{Precision} = \frac{\sum_{word \in C} Count_{\text{match}}(word)}{\sum_{word \in C} Count(word)}, \tag{3.2}$$

where $Count_{\text{match}}(word)$ assigns 1 when *word* appears in both reference and candidate sentence.

However, this definition of precision is not suitable for evaluating a language model, since we may deal with repeated words or a sentence may be expressed with similar words. Thus, we define *clipped* precision and discuss the examples 13 and 14, showing that standard precision from equation (3.2) is an inappropriate metric for language models.

Definition 9. Clipped precision for n-grams is defined as

$$CP_n = \frac{\sum_{n\text{-gram} \in C} Count_{\text{clip}}(n\text{-gram})}{\sum_{n\text{-gram} \in C} Count(n\text{-gram})}, \tag{3.3}$$

where $Count_{\text{clip}}$ is the minimum between $Count_{\text{match}}$ and the number of appearances of the considered n-gram in the reference sentence.

Typically, machine translation involves several sentences, while equation (3.3) is referred to only one. The following definition extends the previous one with several candidate sentences.

Definition 10.

$$CP_n = \frac{\sum_{C \in \{\text{Candidates}\}} \sum_{n\text{-gram} \in C} Count_{\text{clip}}(n\text{-gram})}{\sum_{C \in \{\text{Candidates}\}} \sum_{n\text{-gram} \in C} Count(n\text{-gram})} \tag{3.4}$$

Example 13 (Precision with repetition or permutation)**.** Consider the reference sentence `"BLEU score is an evaluation metric"` and assume that the predicted sentence is `"BLEU BLEU BLEU"`: in this case, the word `BLEU` appears in the reference sentence, thus the precision would be 1. The same precision is achieved by predicting a permuted sentence of the reference one, for instance `"metric is BLEU an score evaluation"`. However, both predicted sentences are meaningless and the obtained results useless.

Example 14 (Precision with multiple target sentences)**.** Usually, similar sentences share the same meaning, which a well defined metric should take into account: therefore, we may accept multiple reference sentences capturing these different variations. Consider the reference sentences

$$R_1 : \texttt{"BLEU score is a good evaluation metric"}$$
$$R_2 : \texttt{"BLEU score is a well defined evaluation metric"}$$

Now, assume that the candidate sentence is

$$C : \texttt{"BLEU BLEU BLEU is a good evaluation measure"}$$

and compute the clipped precision from equation (3.4), which is defined as the ratio between the clipped number of correctly predicted words on the total number of predicted words. For instance, the word `"BLEU"` appears one time in both reference sentences and three times in the candidate sentence: the clipped number is then 1 and not 3, which is useful to avoid repetitions. The words `"is", a", "evaluation"` appear in all three sentences once. Finally, the word `"good"` appears only in R_1; conversely, `"measure"` does not appear in any of the reference sentences. The clipped precision of the candidate sentence C is thus $5/8$, which is a more accurate result than the standard precision, which achieves $7/8$. Table 3.3 shows the described procedure.

Before defining BLEU score, we introduce two concepts: geometric average precision and brevity penalty. At first, from equation (3.4) we compute clipped precision CP_n for n-grams, $n = 1, \dots, N$, where N is a parameter that can be chosen. In this way, we compute the clipped precision for words, pair of words, ... of the predicted sentence, avoiding, for instance a permuted target sentence to have high precision.

Definition 11. We consider a set of weights ω_n such that $\sum_{n=1}^{N} \omega_n = 1$ and $\omega_n \geq 0$

Word in C	*Matching Sentence R_1 or R_2*	$Count_{\text{match}}$	$Count_{\text{clip}}$
`BLEU`	Both	3	1
`is`	Both	1	1
`a`	Both	1	1
`good`	Only R_1	1	1
`evaluation`	Both	1	1
`measure`	None	0	0
Total		**7**	**5**

Table 3.3: Comparison between $Count_{\text{match}}$ used in precision from equation (3.2) and $Count_{\text{clip}}$ used in clipped precision from equation (3.4). In this example we get $Count(1\text{-gram}) = 8$ because C contains 8 words.

and define the geometric average precision as:

$$\text{Geometric Average Precision}(N) = \exp\left(\sum_{n=1}^{N} \omega_n \log\left(CP_n\right)\right) \tag{3.5}$$

If the weights are uniform, i.e. $\omega_n = 1/N$ then

$$\text{Geometric Average Precision}(N) = \prod_{n=1}^{N} p_n^{\omega_n}$$

Definition 12. The brevity penalty penalizes too short sentences:

$$\text{BP} = \begin{cases} 1, & \text{if } c \geq r, \\ e^{(1-r/c)}, & \text{if } c < r, \end{cases} \tag{3.6}$$

where c and r are the lengths of the candidate and reference sentence, respectively.

Definition 13. From equations (3.5) and (3.6) BLEU score is defined as

$$\text{BLEU}(N) = \text{BP} \cdot \text{Geometric Average Precision}(N).$$

We observe that BLEU(1) may encounter the problem encountered in the Example 13 with permuted sentences, therefore $N > 1$ are suggested. In the original paper, the suggested choice is $N = 4$ with uniform weights.

Definition 14. An alternative definition of BLEU score computes the natural logarithm of the formula above, i.e.

$$\log \text{BLEU}(N) = \min\left(1 - \frac{r}{c}, 0\right) + \sum_{n=1}^{N} \omega_n \log p_n.$$

Remark (Adavantages and weakness of BLEU). BLEU score has several advantages and weaknesses: on one hand, it is quick to calculate and it can be applied with several target sentences. Moreover, it is widely used and can be applied to any NLP model since it is language-independent. On the other hand, BLEU score does not capture the meaning of the words: for instance, a synonym of a word may be predicted, decreasing the score, since it would be classified as incorrect. Also, BLEU looks only for word matches and does not take into account slight variations of words, such as `go` or `going`. In a sentence, several words play different roles: BLEU score penalizes in the same way an important word, contributing to the meaning of the sentence, and a less important one.

3.5.2 ROUGE

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is a set of evaluation measures which automatically determine the quality of a model generated text by comparing it to other reference texts and is therefore applied for tasks like text summarization and machine translation. In this section, we present four different ROUGE metrics, which count the number of overlapping units such as n-grams or word sequences [100].

Usually, a text can be summarized in several ways, which are called *reference summaries*, while the network-generated translation is denoted as *candidate summary*. We will discuss the four rouge metrics given a sentence, but the concepts can be easily extended for larger parts of text as summaries:

- **ROUGE-N** for n-gram co-occurrence statistics is defined as

$$\text{ROUGE-N} = \frac{\sum_{R \in \{Reference\ Summaries\}} \sum_{n\text{-gram} \in R} Count_{\text{match}}(n\text{-gram})}{\sum_{R \in \{Reference\ Summaries\}} \sum_{n\text{-gram} \in R} Count(n\text{-gram})},$$

  where n is the length of the n-gram and $Count_{\text{match}}(n\text{-gram})$ is the maximum number of n-grams co-occurring in a candidate summary and a set of refer-

ence summaries. ROUGE-N is a recall-related measure, because the denominator counts the number of n-grams in the reference summaries, while BLEU is a precision-based measure, since the denominator counts the number of n-grams in the candidate summary. We remark that a candidate summary containing words shared by more references is favored by the ROUGE-N measure: intuitively, we prefer a candidate summary which is more similar to the highest possible number of reference summaries.

- **ROUGE-L** identifies the longest common subsequence LCS that appear in the same order, even if not consecutive, in both the reference and candidate summaries. At a sentence-level, it is defined as

$$\text{ROUGE-L} = \frac{LCS}{m},$$

where m is the number of words in the reference summary

- **ROUGE-W** for weighted longest common subsequence favors consecutive LCSes. Indeed, ROUGE-L score does not take into account the consecutive matches of words, but only if they appear in order in the reference summaries. Therefore, ROUGE-W solves this challenge by giving more importance to consecutive matches.

- **ROUGE-S** for skip-bigram co-occurrence statistics measures the overlap of skip-bigrams between the candidate summary and the reference summaries. A skip-bigram is a pair of ordered words in a given sentence with arbitrary gaps in between. While LCS counts only one longest common subsequence, skip-bigram counts all in-order matching word pairs.

One of the great advantages of ROUGE metrics is the multi-reference support, since all metrics can handle multiple reference summaries or translations, providing a more comprehensive and representative evaluation of the model's performance. However, ROUGE has several limitations: for example, ROUGE-N is a recall-based measure, but in some applications it may be more useful to apply a precision-based measure like BLEU. For instance, ROUGE is applied in summarization tasks, since it is important to capture all key points to obtain a comprehensive summary (high recall), even if it contains some extra information; on the other hand, BLEU is applied in translation tasks, since it penalizes translations

including irrelevant information. Moreover, all these metrics are not able to measure semantic similarities, since they only measure overlap of words of sequence of words. Finally, ROUGE score relies on the quality of the reference summaries, which might introduce subjectivity and variability in the results [101].

3.5.3 Perplexity

Another metric for evaluating language models is called *perplexity* and measures how likely the model is to generate a text sequence. Thus, perplexity evaluates the performance on the model on learning the distribution of the texts it was trained on and quantifies the degree of uncertainty in a language model [102]. This section describes this metric and highlights its advantages and application fields.

Definition 15. Given a tokenized sequence $\mathcal{T} = (t_1, \ldots, t_N)$, where t_i are the tokens, the perplexity of $\mathcal{T}$ is defined as [103]

$$\mathrm{PPL}(\mathcal{T}) = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log\left(\mathbb{P}(t_i|t_{1,\ldots,i-1})\right)\right), \tag{3.7}$$

where $\log\left(\mathbb{P}(t_i|t_{1,\ldots,i-1})\right)$ is the log-likelihood of the i-th token conditioned on the preceding tokens $t_1, \ldots, t_{i-1}$.

Unlike the previous two metrics BLEU and ROUGE, perplexity does not compare the model generated sequence with a reference one. Instead, it is calculated on the conditioned probability distribution of the tokens generated by the model. Low perplexity means that the model is confident in text generation, since the conditioned probabilities are high; vice versa, high perplexity indicates a low confidence in text generation [104]. The highest perplexity is achieved when $\mathrm{PPL}(\mathcal{T}) = 1$, i.e. all probabilities are maximal. We also notice that perplexity can be only applied to autoregressive models as GPT-3 which generate the next token in the sequence and is not well defined for models like BERT, since it tests the ability of a model in generating text.

Remark (Benefits, drawbacks and applications). Perplexity metric represents a measure of uncertainty or surprise and relies strongly on the dataset on which the model was trained on, since each dataset has a different distribution of words. Moreover, perplexity does not provide any information on the quality of the generated texts and does not capture semantic nuances of the language. On the other

hand, models with high perplexity can be optimized and refined, enabling the model to generate more fluent text. NLP tasks in which perplexity represents a suitable metric are machine translation for evaluating the predictability and fluency of the output translation, and in text generation, in which perplexity is useful to refine the model to produce more human-like outputs and improving the quality of the text [105].

3.5.4 GLUE Benchmark

The General Language Understanding Evaluation (GLUE) benchmark is a collection of resources for training, evaluating and analyzing natural language understanding systems. It is centered on nine English sentence understanding tasks, covering a wide range of domains, dataset sizes and degrees of difficulty, and is built on established existing datasets [106]. It was introduced in 2019 for the development of models able to generalize across several NLP tasks and domains, encouraging the creation of models towards always broader language understanding capabilities [107].

The key components of GLUE Benchmark are [108]:

- Single-Sentence Tasks:
    1. **CoLA** [109]: The Corpus of Linguistic Acceptability (2018) contains English acceptability judgements from books and journal articles. Each sample is a sequence of words labeled with whether it is a grammatically correct English sentence or not. The evaluation metric is Matthews correlation coefficient.
    2. **SST-2** [110]: The Stanford Sentiment Treebank (2013) consists of sentences from movie reviews and human annotations on their sentiment. Accuracy is chosen as evaluation metric.
- Similarity and Paraphrase Tasks:
    3. **MRPC** [111]: The Microsoft Research Paraphrase Corpus (2005) is a corpus of sentence pairs from online news with human annotations whether the sentences in the pair are semantically equivalent. Since the classes are unbalanced, both accuracy and F1 score are computed.
    4. **STS-B** [112]: The Semantic Textual Similarity Benchmark (2017) consists of sentence pairs from image captions, news headlines and user

forums. Each pair is human-annotated with a score from 1 to 5, thus, Pearson and Spearman correlation coefficients are used as evaluation metrics. This is the only regression task, while all the other eight are single sentence or sentence pair classification tasks.

5. **QQP** [113]: The Quora Question Pairs dataset (2017) contains question pairs from the website Quora. The distribution of classes is unbalanced as in MRPC, therefore both accuracy and F1 score are computed.

- Inference Tasks

6. **MNLI** [114]: The Multi-Genre Natural Language Inference Corpus (2018) is a collection of sentence pairs with textual entailment annotations. A premise sentence and a hypothesis sentence are given: the task is to predict whether the premise entails the hypothesis, contradicts the hypothesis or neither of them. Evaluation is both on the matched and mismatched sections.
7. **QNLI** [115]: The Stanford Question Answering Dataset (2016) is a question-answering corpus containing question-paragraph pairs, where one of the sentences in the paragraph (taken from Wikipedia) contains the answer to the question (written by an annotator). A pair between each question and sentence is formed: the task is to predict whether the context sentence contains the answer.
8. **RTE** [116]: The Recognizing Textual Entailment datasets (2006-2009) are based on news and Wikipedia articles. The task is to determine whether a sentence logically entails another.
9. **WNLI** [117]: The Winograd Schema Challenge (2011) is a reading comprehension task: given a pronoun in the sentence, the goal is to predict the referent of that pronoun from a list of choices.

Table 3.4 sums up the GLUE Benchmark.

Since LLMs increase year-by-year their ability to solve more complex tasks, the performances on the GLUE benchmark came close to the level of non-expert humans. Therefore a new benchmark was developed in 2020, called SuperGLUE benchmark [118], with the aim of providing more difficult language understanding tasks.

Corpus	|Train|	|Test|	Task	Metrics	Domain
Single-Sentence Tasks					
CoLA	8.5k	1k	acceptability	Matthews corr.	misc.
SST-2	67k	1.8k	sentiment	acc.	movie reviews
Similarity and Paraphrase Tasks					
MRPC	3.7k	1.7k	paraphrase	acc./F1	news
STS-B	7k	1.4k	sentence similarity	Pearson/Spearman corr.	misc.
QQP	364k	391k	paraphrase	acc./F1	social QA questions
Inference Tasks					
MNLI	393k	20k	NLI	matched acc./mismatched acc.	misc.
QNLI	105k	5.4k	QA/NLI	acc.	Wikipedia
RTE	2.5k	3k	NLI	acc.	news, Wikipedia
WNLI	634	146	coreference/NLI	acc.	fiction books

Table 3.4: Description of the tasks included in GLUE Benchmark [107].

3.5.5 Evaluation of LLaMA and GPT-3

LLaMA (Large Language Model Meta AI) [1] is a collection of pretrained and instruction tuned generative models with 8B, 70B and 405B parameters [119]; GPT-3 model is described thoroughly in Section 3.2. Since LLaMA and GPT-3 are multilingual networks, that is support several input languages such as English and Italian, multilingual benchmarks for evaluation should be also taken into account. This subsection illustrates several benchmarks on which these two models have been evaluated.

LLaMA base pretrained model's evaluation relies on the following benchmarks using different metrics:

- **MMLU** [120] (Massive Multitask Language Understanding) is a benchmark covering 57 subjects across STEM, social sciences and humanities, and ranging in difficulty from an elementary to an advanced professional level. It tests the model on multiple choice questions and the metric is the accuracy.
- **CommonSenseQA** [121] is a dataset for commonsense question answering tasks and consists of 12.247 question with 5 choices each. The metric is therefore the accuracy.
- **Winogrande** contains 44k problem inspired by Winograd Schema Challenge, which is part of GLUE benchmark.

- **TriviaQA-Wiki** [122] is a dataset containing 650k question-answer-evidence triplets, therefore the exact match as metric.
- **SQuAD** [123] (Stanford Question Answering Dataset) is a reading comprehension dataset, consisting of questions, where the answer is a segment of text from the corresponding reading passage. The question might also be unanswerable. The metric is the exact match.
- **DROP** [124] (Discrete Reasoning Over Paragraphs) is a 96k-question benchmark, in which a model has to resolve references in a question and perform discrete operations over them, such as addition, counting or sorting. The metric is the F1 score, which is a combination of precision and recall.

LLaMA can be fine-tuned on different instructions, such as for generating code or solving more complex mathematical problems. Few examples of benchmarks include:

- **HumanEval** [125] is a problem solving dataset used to measure the correctness for synthesizing programs from docstrings. It consists of 164 programming problems, including algorithms and simple mathematics. The metric is pass@1, which is the percentage of coding problem that the model solves correctly at the first attempt.
- **GSM-8K** [126] is a dataset of 8.5k diverse grade school math problems. The solutions involve performing a sequence of elementary calculations to reach the final answer. The metric is the exact match with majority voting scheme at one attempt.

Similarly to LLaMA, also GPT-3 can be evaluated on several benchmarks, such as:

- **LAMBADA** [127] tests the modeling of long-range dependencies in the text and the model has to predict the last word of sentences which requires reading a paragraph of context. Both perplexity and accuracy are used.
- **TriviaQA** and **Winogrande** as in LLaMA.
- **WMT** [128] is a collection of datasets used in translation tasks in different fields, like news, biomedical or chat translation. GPT-3's training data is primilarly English (93%), but includes also 7% of text in other languages. The metric is BLUE.

- **PhysicalQA** [129] is a dataset for commonsense reasoning and evaluates the model's ability in capturing physical and scientific knowledge with accuracy as metric.
- **ARC** [130] (AI2's Reasoning Challenge) is a multiple-choice question-answering dataset, containing questions from science exams from grade 3 to grade 9. The metric is the accuracy.
- **SuperGLUE**

GPT-3 instruction-tuning works as in LLaMA; since both are multilingual models, evaluating them only on English datasets is not sufficient, for instance the multilingual benchmark MMLU is a dataset containing a translated version of the MMLU dataset for multiple-choice question from various branches of knowledge.

Chapter 4

PEFT

Once an LLM has been trained, it can be fine-tuned for solving specialized and complex tasks, such as text summarization or code generation using instruction fine-tuning datasets. The main drawback is that full fine-tuning requires that all model's parameters are re-trained and stored, which is computationally highly inefficient. For instance, a fine-tuning of GPT-3 would require to modify all 175B weights. Several techniques were studied in order to drastically reduce the number of trainable parameters without diminishing the performances of the fine-tuned model: this approach is called *Parameter-Efficient Fine-Tuning* (PEFT). It is mostly based on different matrix factorization techniques, as low-rank, sparse or SVD, which allow to achieve comparable or sometimes even higher performances than full fine-tuning. Indeed, full fine-tuning may sometimes lead to "catastrophic forgetting", where the model loses its learned knowledge during pre-training. PEFT, instead, by updating only a small subset of parameters, enables the model to adapt better to a specific task, without loosing knowledge and abilities on other pre-trained tasks [131]. Moreover, from a computational point of view PEFT is both compute- and memory-efficient.

In this chapter we discuss Low-Rank Adaptation (LoRA) [5] and two of its variants, LoRA+ [132] and LoRA-drop [133], which adds a low-rank factorization of the fine-tuning matrix and achieves similar performances compared to full fine-tuning. However, the gap between LoRA and full fine-tuning becomes wider when considering larger language models and more complex tasks, for instance with LLaMA-2 and LLaMA-3 on Commonsense Reasoning dataset. Thus, another technique, called Sparse Matrix Tuning (SMT) [6], makes use of matrix sparsity and allows to narrow this gap, obtaining high performances, sometimes

even higher than full fine-tuning itself. A last PEFT method that we introduce is named Singular Values Fine-Tuning (SVFT) [7], which updates the weights as a sparse combination of outer products of its singular vectors, training only its coefficients. Finally, we apply LoRA and SMT to the open-weight language model LLaMA-2-7B, scale it and compare the obtained results, both in terms of training time and performances to solve various tasks. Since training or fine-tuning LLMs requires huge computational resources, which would be impossible to be satisfied on a classical computer, for instance in terms of memory storage, we underline the need of HPC and the role of parallelization to solve such problems.

4.1 LoRA

Low-Rank Adaptation (LoRA) of LLMs [5] is a technique developed in 2021 to efficiently adapt a pre-trained language model to solve a specific task and allows to reduce the trainable parameters of the network, while keeping approximately the same performances compared to the whole fine-tuning of the model. The inspiration behind LoRA is built upon the observation that large-scale pre-trained models reside on a low intrinsic dimension, which means that their parameters often contain redundancies. Then, it was hypothesized that also the change in weights during fine-tuning process has a low intrinsic rank. This section explores LoRA and some of its variants, dealing with advantages and disadvantages of this technique.

Given an autoregressive large-scale pre-trained language model $\mathbb{P}_{\boldsymbol{W}_0}(y|x)$ with weights $\boldsymbol{W}_0$, we want to adapt it to solve several downstream tasks, each of them represented by a supervised dataset $\mathcal{Z} = \{(x_i, y_i)\}_{i=1}^{N}$. For instance, if our model has to be fine-tuned on text summarization, x_i represents the input text and y_i the output summarization. Full fine-tuning updates all the parameters of each layer of the model $\boldsymbol{W}_0$ by adding the weights $\Delta\boldsymbol{W}$ s.t.

$$\boldsymbol{W}_0 + \Delta\boldsymbol{W} \in \underset{\boldsymbol{W}}{\operatorname{argmax}} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log\left(\mathbb{P}_{\boldsymbol{W}}(y_t|x, y_1, \ldots, y_{t-1})\right). \tag{4.1}$$

The main drawback of full fine-tuning is that $\Delta\boldsymbol{W}$ and $\boldsymbol{W}_0$ share the same number of elements, which is computationally and memorically inefficient, especially when dealing with LLMs. Then, for each downstream task, a new $\Delta\boldsymbol{W}$ has to be created. For instance, a full fine-tuning of GPT-3 would require an update of

175B parameters. LoRA approach instead encodes $\Delta \boldsymbol{W} = \Delta \boldsymbol{W}(\Theta^*)$, where Θ^* is a small set of parameters, s.t.

$$\Theta^* \in \underset{\Theta}{\operatorname{argmax}} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log\left(\mathbb{P}_{\boldsymbol{W}_0+\Delta\boldsymbol{W}(\Theta)}(y_t|x, y_1, \ldots, y_{t-1})\right). \tag{4.2}$$

Instead of maximizing an objective function depending on $\boldsymbol{W}$ as in equation (4.1), we search the solution of the optimization problem over a lower-dimensional space containing Θ as in equation (4.2). In the following pages we explain the role of Θ by introducing a matrix decomposition of the weights of each layer and compare the results between full fine-tuning and LoRA.

An LLM architecture contains a sequence of layers, each of them described by a weight matrix, which is typically full rank. It was shown that pre-trained language models have a low intrinsic dimension, which means that there exists a low dimensional reparametrization which is effective as full fine-tuning [134]. This means that full fine-tuning may not be the best technique to adapt a model on a certain task, since pre-trained LLMs are yet able to solve a huge variety of tasks and do not need therefore a full update of weights, which would require also huge computational resources. Then, instead of fine-tuning on the whole parameter space, an LLM can achieve high performances also using a lower dimensional portion of that parameter space.
Given a layer l with frozen pre-trained weight matrix $\boldsymbol{W}_0^{(l)} \in \mathbb{R}^{d\times k}$, LoRA procedure adds a fine-tuning matrix $\Delta\boldsymbol{W}^{(l)}$, factorized as $\Delta\boldsymbol{W}^{(l)} = \boldsymbol{B}^{(l)}\boldsymbol{A}^{(l)}$, where $\boldsymbol{B}^{(l)} \in \mathbb{R}^{d\times r}$ and $\boldsymbol{A}^{(l)} \in \mathbb{R}^{r\times k}$. The number r is called rank. The trainable matrices $\boldsymbol{A}^{(l)}$ and $\boldsymbol{B}^{(l)}$ contain $dr + kr$ parameters: if $r \ll \min(d, k)$ then the trainable parameters are a few compared to $\boldsymbol{W}_0^{(l)}$, which contains dk weights. This procedure can be applied to any layer l of the model: as discussed in the original paper, LoRA is not applied to fully connected layers due to parameter efficiency. Indeed, from Table 3.2 we remark that the highest percentage of weights of an LLM is contained in the fully-connected modules. Since our goal is to reduce the number of trainable parameters while keeping similar performances to full fine-tuning, it is a reasonable choice to apply LoRA only to the attention weights, which is also confirmed by several experiments discussed later.
At the beginning of the training we initialize the trainable parameters $\boldsymbol{A}^{(l)} \sim \mathcal{N}(0, \sigma^2)$ and $\boldsymbol{B}^{(l)} = 0$ for the l-th layer, thus $\Delta\boldsymbol{W}^{(l)} = 0$ and we do not modify any parameter of the pre-trained network. Then, we scale $\Delta\boldsymbol{W}^{(l)}$ by $\frac{\alpha}{r}$, where α

is a tuning parameter constant in r and get

$$\boldsymbol{W}^{(l)} \leftarrow \boldsymbol{W}^{(l)} + \frac{\alpha}{r}\Delta\boldsymbol{W}^{(l)}.$$

Using Adam optimizer, tuning α is equivalent to tuning the learning rate, hence we set $\alpha = r$. Figure 4.1 shows LoRA reparametrization with trainable matrices $\boldsymbol{A}$ and $\boldsymbol{B}$.

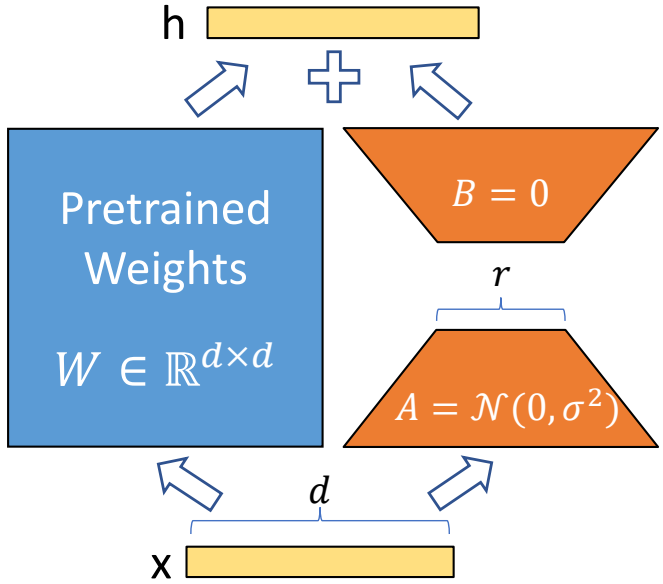

Figure 4.1: LoRA reparametrization with a frozen pre-trained weight matrix $\boldsymbol{W} \in \mathbb{R}^{d\times d}$ and two trainable matrices $\boldsymbol{B} \in \mathbb{R}^{d\times r}$ and $\boldsymbol{A} \in \mathbb{R}^{r\times d}$. Note that here the matrix is square, but the same procedure applies to rectangular matrices. Image from [5].

Remark. Once the trainable matrices $\boldsymbol{A}^{(l)}$ and $\boldsymbol{B}^{(l)}$ for each layer l have been computed and the new weight matrix $\boldsymbol{W}_0^{(l)} + \boldsymbol{B}^{(l)}\boldsymbol{A}^{(l)}$ has been stored, LoRA easily allows to recover the original frozen weights $\boldsymbol{W}_0^{(l)}$ by subtracting $\boldsymbol{B}^{(l)}\boldsymbol{A}^{(l)}$ and adding new trainable matrices $\boldsymbol{B}'^{(l)}\boldsymbol{A}'^{(l)}$ for a new downstream task. This implies that no additional inference latency is introduced.

LoRA technique using a small rank r is compute- and memory-efficient, but a deeper analysis on its performances based on practical experiments has to be discussed. Moreover, several challenges need to be addressed, such as the type of weight matrices to which LoRA should be applied to to achieve best performances and the optimal rank r.

According to the original paper [5] several experiments on RoBERTa, DeBERTa, GPT-2 and GPT-3 with different ranks r and adapting different weight matrices ($\mathbf{\Omega}_k, \mathbf{\Omega}_q, \mathbf{\Omega}_v$ and $\mathbf{\Omega}_d$) were performed on different datasets, which involve tasks such as language understanding and generation. MLP modules, LayerNorm layers and biases were not adapted and their weights were kept frozen, mostly due to parameter efficiency. This allows a great reduction in memory and storage usage, allowing to train the model with fewer GPUs: it was achieved a 25% speedup during training of GPT-3 compared to full fine-tuning due to the lower number of gradients' calculations. In general, LoRA factorization of the attention matrices achieves performances similar or even higher than full fine-tuning on all the considered models and datasets, showing empirically the huge potential of using a low-rank decomposition for fine-tuning, not only from a computational point of view. An example of the obtained accuracies on GPT-3 is shown in Figure 4.2.

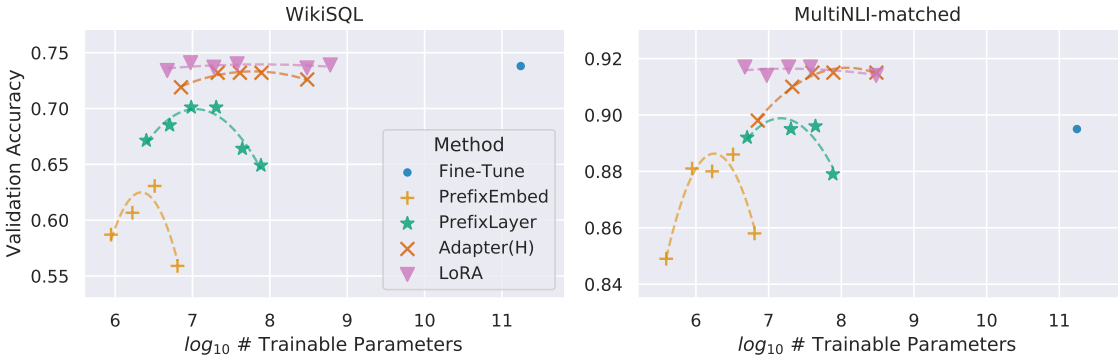

Figure 4.2: Validation accuracy of GPT-3 vs number of trainable parameters on the datasets WikiSQL and MultiNLI using several adaptation methods. We notice that LoRA exhibits similar or even higher accuracies than full fine-tuning. Image from [5].

Several empirical studies were performed to analyze the optimal performances of LoRA varying rank r and adapting different attention blocks. GPT-3 has been fine-tuned for 2 epochs with a batch size of 128. We consider two datasets, WikiSQL [135] [1] and MultiNLI from GLUE benchmark, on which fine-tuning is applied. Following results were obtained:

1. **Optimal subset of weight matrices**: given a parameter budget of 18M

[1]Supervised dataset containing 56.355 and 8.421 training and validation examples. The task is to generate SQL queries from natural language questions and table schemata.

(10.000 times less the parameters of GPT-3), LoRA is applied to adapt one or more type of attention matrices, scaling properly the rank r, in order to keep the number of trainable parameters constant. The results are shown in Table 4.1: optimal results where obtained with $r = 2$ adapting all four matrices $\boldsymbol{\Omega}_k, \boldsymbol{\Omega}_v, \boldsymbol{\Omega}_q$ and $\boldsymbol{\Omega}_d$. Lower performances were achieved by adapting only one of the four matrices even with higher rank $r = 8$. It follows that is preferable to adapt more weight matrices with smaller rank.

2. **Optimal rank r**: we apply LoRA to different attention matrices varying the rank r and see the model performance. The results are shown in Table 4.2. We notice that a small rank as $r = 1$ for adapting $\{\boldsymbol{\Omega}_k, \boldsymbol{\Omega}_v\}$ allows to achieve almost optimal accuracies, which suggests that the fine-tuning matrix $\boldsymbol{\Delta W}$ could have a small intrinsic rank. Moreover, increasing the rank r does not improve in a remarkable way the obtained accuracies, suggesting that a low-rank matrix is sufficient.

Weight Type	$\boldsymbol{\Omega}_q$	$\boldsymbol{\Omega}_k$	$\boldsymbol{\Omega}_v$	$\boldsymbol{\Omega}_d$	$\{\boldsymbol{\Omega}_q, \boldsymbol{\Omega}_k\}$	$\{\boldsymbol{\Omega}_q, \boldsymbol{\Omega}_v\}$	$\{\boldsymbol{\Omega}_q, \boldsymbol{\Omega}_k, \boldsymbol{\Omega}_v, \boldsymbol{\Omega}_d\}$
Rank r	8	8	8	8	4	4	2
WikiSQL	70.4	70.0	73.0	73.2	71.4	**73.7**	**73.7**
MultiNLI	91.0	90.8	91.0	91.3	91.3	91.3	**91.7**

Table 4.1: Validation accuracy on WikiSQL and MultiNLI applying LoRA to different weight matrices in GPT-3, keeping the number of trainable parameters constant. Table from [5].

	Weight Type	$r = 1$	$r = 2$	$r = 4$	$r = 8$	$r = 64$
WikiSQL	$\boldsymbol{\Omega}_q$	68.8	69.6	70.5	70.4	70.0
	$\{\boldsymbol{\Omega}_q, \boldsymbol{\Omega}_v\}$	73.4	73.3	73.7	73.8	73.5
	$\{\boldsymbol{\Omega}_q, \boldsymbol{\Omega}_k, \boldsymbol{\Omega}_v, \boldsymbol{\Omega}_d\}$	74.1	73.7	74.0	74.0	73.9
MultiNLI	$\boldsymbol{\Omega}_q$	90.7	90.9	91.1	90.7	90.7
	$\{\boldsymbol{\Omega}_q, \boldsymbol{\Omega}_v\}$	91.3	91.4	91.3	91.6	91.4
	$\{\boldsymbol{\Omega}_q, \boldsymbol{\Omega}_k, \boldsymbol{\Omega}_v, \boldsymbol{\Omega}_d\}$	91.2	91.7	91.7	91.5	91.4

Table 4.2: Validation accuracy on WikiSQL and MultiNLI applying LoRA to different weight matrices in GPT-3, varying the rank r. Table from [5].

Remark. All the experiments depend strongly on the model and the downstream task datasets: in general, smaller ranks r may not perform well for other tasks on other models. If we deal with more complex tasks, for instance text translation, a higher rank r may significantly achieve higher performances and outperform LoRA with smaller ranks. Moreover, as LLMs get scaled and increase their size, they become able to solve a wider range of tasks, since, for instance, capture more relationships inside the language. Thus, also model's size influences strongly PEFT performance.

A further analysis involves the comparison between the frozen pre-trained weights of $\boldsymbol{W}_0^{(l)}$ and the fine-tuning weights $\Delta\boldsymbol{W}^{(l)}$ at a network layer l. It was shown that these two matrices are strongly correlated, in particular $\Delta\boldsymbol{W}^{(l)}$ amplifies several features present in $\boldsymbol{W}_0^{(l)}$: this suggests that using a low-rank adaptation matrix for fine-tuning allows to emphasize certain features, which are related to the downstream task and were only learned in the pre-trained model. Several variations of LoRA reparametrization were proposed in the last years, such as LoRA+ and LoRA-drop, which are discussed in the next subsections.

4.1.1 LoRA+

LoRA+ [132] is a recent technique from 2024 which analyzes the role of the learning rate on the update of the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$. It was shown that LoRA on models with large embedding dimension D leads to suboptimal fine-tuning, due to the fact that the same learning rate for updating both $\boldsymbol{A}$ and $\boldsymbol{B}$ was used. In this subsection we discuss the advantages of using different learning rates, which allow to improve by 1-2% the performances of classical LoRA and doubling its fine-tuning speed, without increasing the computational cost. In this subsection we focus on the advantages of using different learning rates and discuss some experiments on different models.

During training process, LoRA updates the matrices as follows:

$$\begin{aligned} \boldsymbol{A} &\leftarrow \boldsymbol{A} - \eta\nabla\boldsymbol{A} \\ \boldsymbol{B} &\leftarrow \boldsymbol{B} - \eta\nabla\boldsymbol{B}, \end{aligned} \tag{4.3}$$

where η is the learning rate, $\nabla\boldsymbol{A}$ and $\nabla\boldsymbol{B}$ represent the gradients from Adam optimizer for $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively. We drop the dependence from layer l to make the notation easier. At the first training step, the matrices are initialized as $\boldsymbol{A} \sim \mathcal{N}(0, \sigma^2)$ and $\boldsymbol{B} = 0$. We want to show that using the same learning rate for

both $\boldsymbol{A}$ and $\boldsymbol{B}$ leads to suboptimal learning, especially when the embedding dimension D is large, which is the case for recent LLMs; moreover, the performances can be increased by setting different learning rates. Before discussing theoretical and empirical results on the use of different learning rate for LoRA fine-tuning, we introduce the concepts of stability and efficiency.

LoRA fine-tuning is stable if no quantity in the network explodes as D grows. In order to ensure stability, the hyperparameters as initialization of $\boldsymbol{A}$ and $\boldsymbol{B}$ and learning rate needs to be scaled properly: for instance, for $\boldsymbol{A}$ the variance σ^2 has to scale as D^{-1}. Instead, scaling arbitrarily the learning rate may lead to suboptimal fine-tuning. LoRA fine-tuning is efficient if it is stable and both weight matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ contribute to the final update. For instance, if one of the two matrices is not efficiently updated, we may achieve suboptimal learning. The following theorem shows the optimal scaling of learning rates in order to achieve an efficient fine-tuning for LoRA.

Theorem 1 (Efficient LoRA)**.** *If $\boldsymbol{A}$ and $\boldsymbol{B}$ are trained with Adam optimizer with learning rates $\eta_{\boldsymbol{A}}$ and $\eta_{\boldsymbol{B}}$, respectively, then efficiency is not achieved when $\eta_{\boldsymbol{A}} = \eta_{\boldsymbol{B}}$. However, LoRA fine-tuning is efficient if $\eta_{\boldsymbol{A}} = \Theta(D^{-1})$ and $\eta_{\boldsymbol{B}} = \Theta(1)$*[2]*.*

Proof. See [132], page 15. □

It follows that efficiency can only be reached when $\eta_{\boldsymbol{B}}/\eta_{\boldsymbol{A}} = \Theta(D)$, which means that the learning rate for updating $\boldsymbol{B}$ has to be much greater than the one for $\boldsymbol{A}$. However, the theorem does not provide a precise value for $\eta_{\boldsymbol{B}}/\eta_{\boldsymbol{A}}$: instead of implementing a 2D search tuning on both learning rates, which is highly inefficient from a computational point of view, we perform a 1D search by fixing a value for the ratio $\eta_{\boldsymbol{B}}/\eta_{\boldsymbol{A}}$ and tune only one of the two learning rates. In this way, LoRA+ and LoRA achieve same computational cost. We denote the ratio $\eta_{\boldsymbol{B}}/\eta_{\boldsymbol{A}}$ as λ: thus, $\eta_{\boldsymbol{B}} = \lambda\eta_{\boldsymbol{A}}$, where $\lambda \gg 1$ and we tune $\eta_{\boldsymbol{A}}$. The optimal ratio λ can be found through experiments, depending on the model and downstream task. Compared to equation (4.3) the training process of LoRA+ can be written as

$$\begin{aligned} \boldsymbol{A} &\leftarrow \boldsymbol{A} - \eta\nabla\boldsymbol{A} \\ \boldsymbol{B} &\leftarrow \boldsymbol{B} - \lambda\eta\nabla\boldsymbol{B},\ \lambda \gg 1. \end{aligned} \tag{4.4}$$

[2]Given two sequence $c_n \in \mathbb{R}$ and $d_n \in \mathbb{R}^+$, we write $c_n = \mathcal{O}(d_n)$, respectively $c_n = \Omega(d_n)$, if $c_n < kd_n$, resp. $c_n > kd_n$ for some positive constant k. We write $c_n = \Theta(d_n)$ if both $c_n = \mathcal{O}(d_n)$ and $c_n = \Theta(d_n)$ hold.

The theoretical results from Theorem 1 are confirmed by several experiments on different language models, as RoBERTa, GPT-2 and LLaMA. On RoBERTa and GPT-2 with $\alpha = r = 8$ the accuracy is maximal when $\eta_{\boldsymbol{B}} \gg \eta_{\boldsymbol{A}}$, outperforming the classical LoRA with equal learning rates. It was observed that the gap between the optimal choice of learning rates and the optimal choice when $\eta_{\boldsymbol{B}} = \eta_{\boldsymbol{A}}$ is more evident for harder tasks as MNLI and QQP from the GLUE benchmark. Considering LLaMA-7B with hyperparameters $\alpha = 16$ and $r = 64$ on MMLU benchmark achieved a 1.3% gain taking optimal $\eta_{\boldsymbol{B}} \gg \eta_{\boldsymbol{A}}$; on MNLI benchmark with hyperparameters $\alpha = 16$ and $r = 8$ nearly optimal performances were achieved using equal learning rates, since the task is easy for LLaMA but hard for smaller models as RoBERTa and GPT-2. A final analysis involves the optimal value of λ, which is model and task sensitive: for instance, a value of 2^2 or 2^3 is optimal for RoBERTa, while for LLaMA a value of 2^1 or 2^2 is preferred.

4.1.2 LoRA-drop

LoRA-drop [133] is a recent method from 2024 which optimizes fine-tuning by applying LoRA only to the most impactful layers by considering their outputs. On these layers LoRA technique is applied each with his own matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, while the remaining layers share the same reparametrization. Experiments showed that LoRA-drop can achieve performances similar to full fine-tuning and LoRA, while considering only half of the original LoRA parameters. In this subsection we explore LoRA-drop, focusing on its advantages and describing several results from the experiments.

While LoRA+ deals with the different learning rates for updating $\boldsymbol{A}$ and $\boldsymbol{B}$, LoRA-drop's main goal is to reduce the trainable parameters of LoRA without having a downgrade in the performances and achieving a faster training. The idea behind is to analyze how each output of each LoRA layer, which depends on the given data and trained parameters, impacts on the final result: for instance, given $\boldsymbol{x}^{(1)}$ the input at the first layer, then if $||\Delta \boldsymbol{W}^{(1)}\boldsymbol{x}^{(1)}||$ is large then $\boldsymbol{x}^{(1)}$ has a relevant impact on the first layer. In general, given input $\boldsymbol{x}^{(l)}$ at layer l the output is

$$\boldsymbol{h}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{x}^{(l)} + \Delta \boldsymbol{W}^{(l)}\boldsymbol{x}^{(l)}.$$

The value $\boldsymbol{B}^{(l)}\boldsymbol{A}^{(l)}\boldsymbol{x}^{(l)}$ depends clearly on the LoRA parameters and the hidden state, hence higher values correspond to more important LoRA. We remark that the hidden state or input at layer l is computed from the fine-tuning dataset

through the preceding layers. After several experiments, it was observed that $||\Delta \boldsymbol{W}^{(l)}\boldsymbol{x}^{(l)}||^2$ is highly concentrated for each layer, showing a peak Gaussian distribution. Moreover, for certain layers $||\Delta \boldsymbol{W}^{(l)}\boldsymbol{x}^{(l)}||^2$ consistently remains close to zero, showing that LoRA has no impact on the model: therefore, these layer are kept frozen and not trained. Figure 4.3 shows a diagram of LoRA-drop.

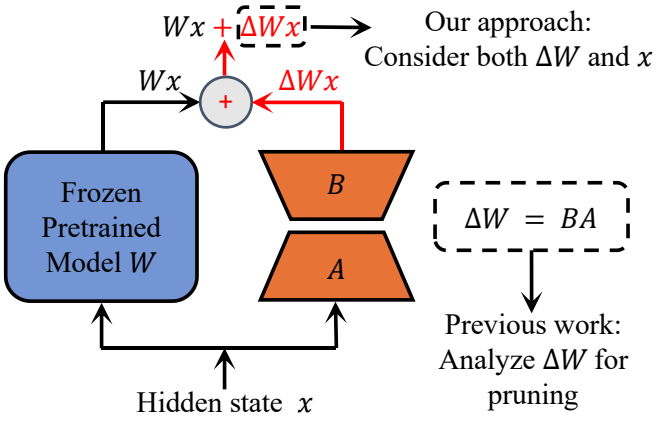

Figure 4.3: Diagram of LoRA-drop. LoRA influences the pre-trained model through its output $\Delta \boldsymbol{W}\boldsymbol{x}$. Image from [133].

Following these empirical observations, LoRA-drop technique was proposed, consisting in two steps:

1. **Importance Evaluation**: given a downstream task dataset $\mathcal{Z} = \{\boldsymbol{x}, \boldsymbol{y}\}$, where $\boldsymbol{x}$ are the inputs and $\boldsymbol{y}$ the outputs, we sample a subset $\mathcal{Z}_s$. In the original experiments, the sampling ratio α was set to 10%: even if the subset is much smaller than the full dataset, the LoRA importance distributions were similar, allowing to compute the importance for less data. We then compute
$$g_l = \sum_{\boldsymbol{x} \in \mathcal{Z}_s} ||\Delta \boldsymbol{W}^{(l)}\boldsymbol{x}^{(l)}||^2,$$
and, by normalizing,
$$I_l = \frac{g_l}{\sum_l g_l}.$$
The magnitude of g_l shows the impact of LoRA on layer l and the importance I_l for layer l is a value between 0 and 1.

2. **Task Adaptation**: we sort the layers according to I_l and select the layers from most to least important important until the sum importance reaches a

threshold T. In the original paper, $T = 0.9$. The LoRA of the selected layers is re-trained on the whole dataset $\mathcal{Z}$, while a shared LoRA reparametrization replaces the LoRA of the other layers.

Experiments on RoBERTa and LLaMA-7B were performed to evaluate the models on GLUE benchmark for natural language understanding task. Moreover, LLaMA-7B was also tested on natural language generation tasks. Experiments on RoBERTa-base showed that by reducing of 50% the number of trainable parameters of LoRA, LoRA-drop achieves an average score of 86.2 on GLUE benchmark, comparable with full fine-tuning (86.4) and LoRA (86.1). On RoBERTa-large LoRA-drop with a score of 89.1 outperforms on average both full fine-tuning and LoRA by 0.2 points on the GLUE benchmark. Similar results were obtained on LLaMA-7B where LoRA-drop achieves a score of 89.3, while LoRA 89.2 and full fine-tuning of 87.3 on GLUE benchmark. Finally, for natural language generation (NLG) tasks using only 68% of the original LoRA parameters, which correspond to 0.09B parameters, LoRA-drop outperforms both full fine-tuning and LoRA methods on LLaMA-7B.

Finally, the choice of the sampling ratio of $\alpha = 10\%$ and of the threshold $T = 0.9$ were motivated through a series of results. As the training data increases, the importance order of each layer remains consistent: for instance, applying LoRA to the query matrices on RoBERTa-base model the 10th layer was always the most important, followed by layers 7,8,9. This shows that a small sampling ratio is sufficient for obtaining accurate values of importance. On the threshold T, if $T = 1$ then classical LoRA is applied. If $T < 0.9$ the model performance increases, if $T = 0.9$ approximately half of the layers were selected and if $T > 0.9$ the performance does not improve significantly: hence T was set to 0.9.

4.2 SMT

In the previous section we described LoRA and its variants which are PEFT techniques which aim to improve the performance of a language model by fine-tuning a lower dimensional parametrization of the model parameters. However, the gap between LoRA and full fine-tuning becomes relevant as the model's size increases. Other studies on modifying and locating memories in transformer based models as [136], [137] and [138], have shown that LLMs have memory sections located in distinct layers, which can be identified through fine-tuning. This means

that they contain domain-specific knowledge distributed separately and sparsely among the different layers: for instance, certain layers have higher impact on summarization tasks, while others on translation task. Therefore, a new technique called Sparse Matrix Tuning (SMT) [6] was recently proposed, which applies matrix sparsity and fine-tunes the most relevant memory sections of the model, outperforming LoRA with the same number of trainable parameters. A comparison between LoRA and SMT is shown in Figure 4.4.

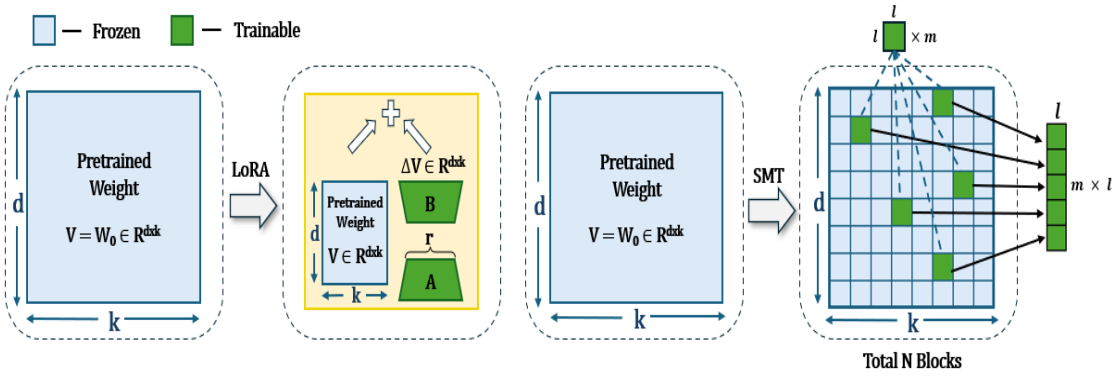

Figure 4.4: Comparison between LoRA (on the left) and SMT (on the right). For SMT the m green square matrices of dimension $l \times l$ indicate the selected sub-matrices to which fine-tuning is applied. The remaining light-blue sub-matrices are kept frozen. Image from [6].

From equation (4.2) SMT uses matrix sparsity as parameter-efficient approach, where Θ represents the sub-matrices within the sparse weight matrices. SMT slices a pre-trained weight matrix $\boldsymbol{W}_0 \in \mathbb{R}^{d \times k}$ into n_{SMT} sub-matrices and applies fine-tuning only on m_{SMT} of them, which dimension is $l \times l$. In the original paper $l = 256$: since the considered LLM is LLaMA, $l = 256$ is the largest common factor of the column and row sizes of all linear layers. The total number of sub-matrices is $n_{\text{SMT}} = \frac{d \times k}{l \times l}$. As in LoRA, SMT adds to $\boldsymbol{W}_0$ an update matrix $\Delta \boldsymbol{W}_{M_{\text{SMT}}}$ to get

$$\boldsymbol{W}_0 + \Delta \boldsymbol{W}_{m_{\text{SMT}}}, \; m_{\text{SMT}} \ll n_{\text{SMT}}.$$

A warm-up phase of 100 iterations allows to identify and select the most relevant sub-matrices, which exhibit maximal gradient changes for the specific fine-tuning task, as shown in Figure 4.5. A warm-up phase is a set of iterations before the training process in which the pre-trained model computes the gradients from a

given dataset. Thanks to a sparsity approach, SMT improves computational efficiency and reduces memory usage, allowing to achieve higher speedup than LoRA, since, for instance, unselected gradients are not calculated and updated. Moreover, since gradients are calculated and saved only for a subset of weights, SMT reduces the backward computation cost, the activation memory cost, the memory cost of the optimizer gradients and the gradient step computation. As in LoRA, only attention layers are updated, while MLP layers are kept frozen: several experiments discussed later have shown that attention layers are more relevant for achieving higher performances than MLP layers.

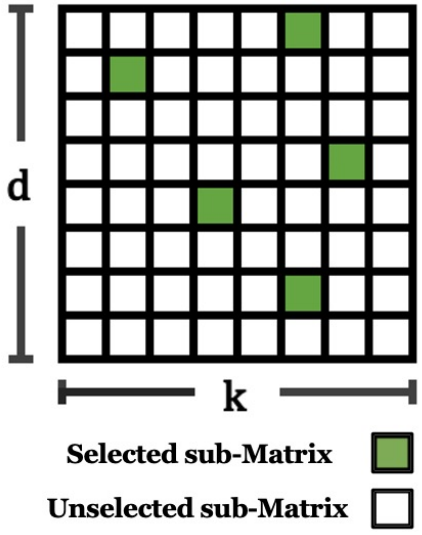

Figure 4.5: A $d \times k$ sparse weight matrix. The green sub-matrices of dimension $l \times l$ exhibit maximal gradient changes and are updated. Image from [6].

Experiments were run on open-weight LLaMA models, which were fine-tuned on Common Sense Reasoning with 8 sub-tasks for 3 epochs and the results compared with LoRA. Results are shown in Table 4.2: SMT outperforms LoRA given similar number of trainable parameters. Highest performances similar to full fine-tuning of SMT are achieved by training approximately 5% of the weights, except the largest model LLaMA-3-8B where only 3% are needed.

Another experiment involves the fine-tuning also of MLP layers: in particular, considering LLaMA-7B on the Commonsense dataset, SMT is applied and 0.84% of the parameters is updated. The highest accuracy is obtained when MLP layers

Model	PEFT	#Params (%)	Average performance
LLaMA-7B	LoRA (Best)	0.83	76.3
	SMT	0.84	**78.7**
	SMT (Best)	0.84	**78.7**
	Full fine-tuning	100	81.4
LLaMA-13B	LoRA (Best)	0.67	80.5
	SMT	0.68	**82.4**
	SMT (Best)	4.91	84.3
LLaMA-2-7B	LoRA (Best)	0.83	77.6
	SMT	0.84	**81.8**
	SMT (Best)	4.91	83.4
	Full fine-tuning	100	82.2
LLaMA-3-8B	LoRA (Best)	0.70	80.8
	SMT	0.71	**86.8**
	SMT (Best)	3.01	87.2

Table 4.3: Accuracy comparison on average of different LLaMA models on 8 Commonsense Reasoning datasets with LoRA, SMT and full fine-tuning. In bold the average accuracy of SMT under similar percentage of trainable parameters, where LoRA achieves best performance. Also SMT with highest accuracy is shown. Table from [6].

were kept frozen, while lowest accuracy is achieved when only MLP layers were updated, showing that attention mechanisms play a fundamental role for solving language tasks: the more trainable parameters are allocated to attention layers, the better the fine-tuned model performs, as shown in Table 4.4. Since attention layers are more impactful in the model fine-tuning, a final experiment involves the different role of queries, keys and values. It was observed that the majority of the weights is assigned to the values, showing that they contain the most of the memory. Figure 4.6 shows that all values in fine-tuning LLaMA-7B with SMT contain trainable parameters, while 22 query layers and 21 key layers are frozen. Moreover, if we apply SMT only to one among values, keys or queries, a significant gap is obtained when allocating all the trainable parameters to the values, suggesting again that values are the most impactful layers for SMT on this LLaMA-7B.

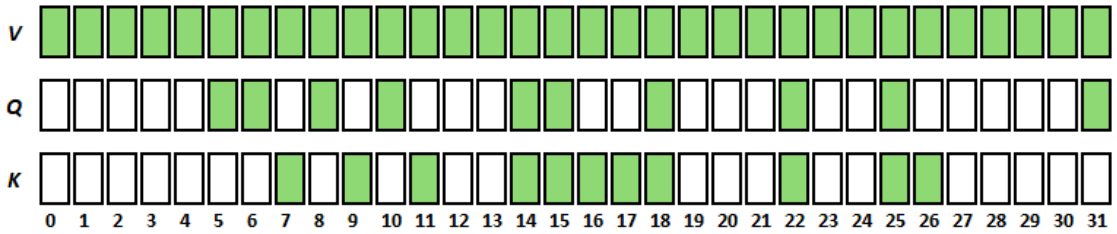

Figure 4.6: Fine-tuning of LLaMA-7B on Commonsense dataset with SMT and 0.86% trainable parameters. Values $\boldsymbol{V}$, queries $\boldsymbol{Q}$ and keys $\boldsymbol{K}$ are shown: white layers are frozen, green layers are updated with SMT. Image from [6].

Model	MLP%	Attention%	Average performance
LLaMA-7B	0.84	0	76.7
	0.42	0.42	77.3
	0.21	0.63	77.8
	0	0.84	78.7

Table 4.4: Fine-tuning of LLaMA-7B on Commonsense dataset with SMT. MLP% and Attention% show the percentage of trainable parameters for MLP and attention layers, respectively. Table from [6].

4.3 SVFT

Singular Vectors Fine Tuning (SVFT) [7] is a fine-tuning approach that updates the pre-trained frozen weight matrix by adding a sparse weighted combination of its own singular vectors. This technique achieves high downstream accuracy, while training only a small percentage of parameters. In this section, we explain SVFT, focusing in particular on the advantages of matrix factorizations through SVD for fine-tuning an LLM.

Let $\boldsymbol{W}_0 \in \mathbb{R}^{d\times k}$ be the usual frozen pre-trained matrix belonging to a certain layer of the pre-trained LLM. At first we computed its Singular Value Decomposition and get $\boldsymbol{W}_0 = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$, where $\boldsymbol{U} \in \mathbb{R}^{d\times d}$ is the square matrix of left singular vectors (orthonormal columns), $\boldsymbol{\Sigma} \in \mathbb{R}^{d\times k}$ is a diagonal matrix of singular values and $\boldsymbol{V}^T \in \mathbb{R}^{k\times k}$ is the square matrix of right singular vectors (orthonormal rows).

Then, we factorize the fine-tuning matrix as $\Delta \boldsymbol{W} = \boldsymbol{U}\boldsymbol{M}\boldsymbol{V}^T$, where $\boldsymbol{U}$ and $\boldsymbol{V}^T$ are computed and frozen, while $\boldsymbol{M} = (m_{i,j}) \in \mathbb{R}^{d \times k}$ is a sparse trainable matrix with fixed sparsity pattern Ω. Thus, we train only the non-zero elements of $\boldsymbol{M}$; $|\Omega|$ allows to control the number of trainable parameters. In particular,

$$\boldsymbol{W}_0 + \Delta \boldsymbol{W} = \sum_{i=1}^{\min(d,k)} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T + \sum_{(i,j)\in\Omega} m_{i,j} \boldsymbol{u}_i \boldsymbol{v}_j^T. \tag{4.5}$$

Given an input $\boldsymbol{x}$ the forward pass of SVFT is

$$\boldsymbol{h} = \boldsymbol{W}_0 \boldsymbol{x} + \Delta \boldsymbol{W} \boldsymbol{x} = \boldsymbol{U}(\boldsymbol{\Sigma} + \boldsymbol{M})\boldsymbol{V}^T \boldsymbol{x}.$$

Four given choices of Ω are:

1. **Plain**: $\boldsymbol{M}$ is diagonal, since $m_{i,j} = 0$ if $i \neq j$. Equation (4.5) reduces to

$$\boldsymbol{W}_0 + \Delta \boldsymbol{W} = \sum_{i=1}^{\min(d,k)} (\sigma_i + m_{i,i}) \boldsymbol{u}_i \boldsymbol{v}_i^T,$$

   which means that only the diagonal elements are trained. It can be interpreted as adapting the singular values which becomes $\sigma_i + m_{i,i}$ and reweighting the frozen singular vectors. This is the most parameter-efficient SVFT.

2. **Banded**: $\boldsymbol{M}$ is a banded matrix, since $m_{i,j} = 0$ if $j < i - b_1$ or $j > i + b_2$ for $b_1, b_2 \geq 0$. It extends the plain case to more than one diagonal, allowing to capture more interactions between the singular values.

3. **Random**: $\boldsymbol{M}$ is a randomly chosen sparse matrix.

4. **Top-j**: it works only when $d = k$, hence the pre-trained frozen matrix is square. We select and only train the top-j elements, which means that only the top-j strong interactions between singular vector directions are learnable.

Figure 4.7 shows SVFT approach with the four described sparsity patterns.

Several remarkable properties are highlighted in the following proposition, which shows interesting improvements of LoRA technique.

Proposition 1. *Let $\boldsymbol{W}_0 \in \mathbb{R}^{d \times k}$ with SVD given by $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$. Consider the matrix $\boldsymbol{W}_0 + \boldsymbol{U}\boldsymbol{M}\boldsymbol{V}^T$, where $\boldsymbol{M}$ has the same size of $\boldsymbol{\Sigma}$. It holds:*

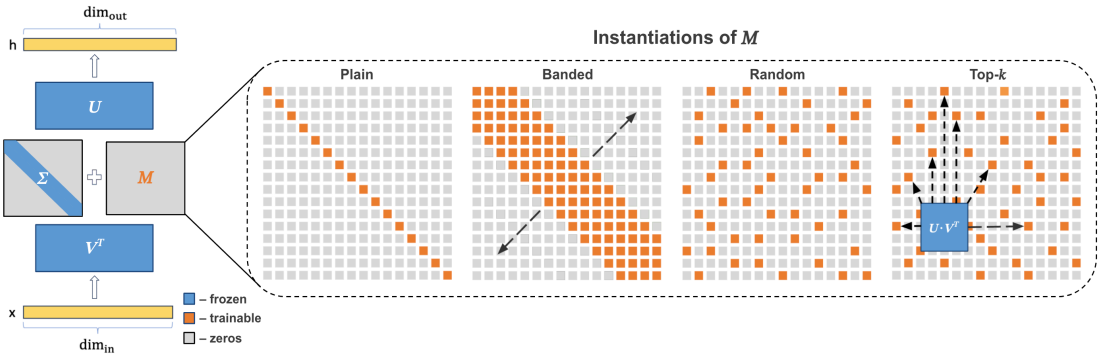

Figure 4.7: Overview of SVFT. The pre-trained frozen weight matrix is factorized through SVD to get $\boldsymbol{W}_0 = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$. The fine-tuning matrix is factorized as $\Delta\boldsymbol{W} = \boldsymbol{U}\boldsymbol{M}\boldsymbol{V}^T$, where $\boldsymbol{M}$ is a sparse trainable matrix (the orange elements are adapted, the gray ones are zero). Image from [7].

1. ***Structure**: if $\boldsymbol{M}$ is diagonal (plain SVFT) then $\boldsymbol{W}_0 + \boldsymbol{U}\boldsymbol{M}\boldsymbol{V}^T$ has $\boldsymbol{U}$ as left singular vectors and sign$(\boldsymbol{\Sigma} + \boldsymbol{M})\boldsymbol{V}^T$ as right singular vectors. Hence, singular vectors do not change except for possible sign flips.*

2. ***Expressivity**: given a target matrix $\boldsymbol{P}$ of the same size of $\boldsymbol{W}_0$, there exists $\boldsymbol{M}$ such that $\boldsymbol{P} = \boldsymbol{W}_0 + \boldsymbol{U}\boldsymbol{M}\boldsymbol{V}^T$. This implies that $\boldsymbol{M}$ is full trainable and any target matrix can be realized.*

3. ***Rank**: if $\boldsymbol{M}$ has j non-zero elements, the rank of $\boldsymbol{U}\boldsymbol{M}\boldsymbol{V}^T$ is at most $min\{j, min(d, k)\}$. Thus, SVFT can produce a much higher rank perturbation than LoRA.*

Proof. See [7], page 13. □

We compare now the fine-tuning matrix of LoRA and SVFT:

$$\begin{aligned} \text{LoRA: } \Delta\boldsymbol{W} &= \sum_{i=1}^{r} \boldsymbol{a}_i \boldsymbol{b}_i^T \\ \text{SVFT: } \Delta\boldsymbol{W} &= \sum_{(i,j)\in\Omega} m_{i,j} \boldsymbol{u}_i \boldsymbol{v}_j^T. \end{aligned} \tag{4.6}$$

We remark that LoRA reparametrization can be expressed a sum of r rank-one matrices, while SVFT factorization as a sum of $|\Omega|$ rank-one matrices. LoRA requires $d + k$ parameters per rank-one matrix, while SVFT only one. However,

SVFT has to compute and store the SVD of $\boldsymbol{W}_0$ and roughly doubles the memory usage with respect to LoRA.

SVFT experiments on encoder-only model DeBERTaV3-base and two decoder-only model Gemma-2B/7B and LLaMA-3-8B were performed for solving both natural language generation and understanding tasks. These showed that SVFT is an excellent trade-off between parameter efficiency and performance, allowing to achieve similar performance of LoRA with much fewer parameters and outperforming in several cases full fine-tuning, making this technique an attractive PEFT choice. For NLG tasks Gemma-2B/7B and LLaMA-3-8B are evaluated on GSM-8K, MATH dataset and Commonsense reasoning benchmark: on average on the first two datasets, SVFT plain achieves 18% relative improvement over techniques that use 6× more trainable parameters and 15.5% relative improvement on overage over techniques with comparable sizes. On GSM-8K, by training only 0.25% of the pre-trained model's parameters, SVFT random is able to achieve 96% of full fine-tuning, while other PEFT methods recover at most 86% with 2× more parameters. Instead, on Commonsense reasoning benchmark, SVFT plain shows competitive performances in comparison with LoRA with $r = 1$ which as 1.9× more parameters. For NLU tasks on GLUE benchmark SVFT matches LoRA with $r = 8$ using 12× more trainable parameters. As in LoRA and SMT, adapting more types of attention blocks enhances performance.
Finally, a result on the impact of $\boldsymbol{M}$'s sparsity is discussed. Both random and Top-k variants outperform SVFT banded on GSM-8K dataset. However, SVFT banded has highest performances on MATH dataset. This suggests that the sparsity pattern may depend strongly on the downstream task, hence the choice of parametrization has a relevant influence on fine-tuned model's performance.

4.4 LLM in HPC

Due to the large and increasing size of LLMs, the training process, which needs to learn billions or even trillions of weights, is computationally expensive and requires significant memory resources. Laptops do not have have enough memory to store all the weights and gradients and the training process would require several years, hence more powerful computers are required to train such models. Supercomputers as Leonardo are equipped with a large number of compute nodes and, thanks to High Performance Computing (HPC), are able to store huge quantity of data and train LLMs faster. Thus, supercomputers play a crucial role to signifi-

cantly reduce pre-training, fine-tuning and inference time if compared to classical computers. In this section, we discuss the role of HPC for training an LLM and illustrate a library called *Accelerate* which is useful for accelerating our experiments.

Laptops have a limited number of CPUs and low memory storage, and the majority of them does not have GPUs, which would allow to perform several calculations simultaneously. Thus, it is extremely hard to train an LLM on these computers. For instance, GPT-3 training would require 355 years if done on a single GPU, while the parameters would need 700GB memory space if stored as float32 (4 bytes per weight), which drastically exceeds the memory available in a GPU [139]. However, this problem may be solved by using supercomputers through parallel computing: a given problem can be split into smaller subproblems, which can be executed simultaneously, reducing training time and memory storage. For instance, using 1.024 GPUs, GPT-3 training may be reduced to approximately one month [140]. Thus, HPC accelerates considerably training process, enabling to train large models in a significantly smaller fraction of time than that required by traditional computing methods.

The main idea behind parallel computing is to split a given problem into smaller problems, which can be executed at the same time: this allows to reduce training cost, make an efficient use of GPUs and CPUs, and therefore solve more complex problems [141]. Thanks to the huge number of computing resources, supercomputers rely strongly on parallel computing, since a lot of operations inside a neural network can be split into smaller independent parts. For instance, a product between two matrices can be performed serially, multiplying the first row with the first column, and so on, or in parallel, where each row is multiplied with each column simultaneously, allowing for a much more efficient implementation, both in training and inference process. In our experiments, fine-tuning LLaMA-2-7B using serial computing would have taken approximately 10× more time than using parallel computing with 16 GPUs, showing the high performances achieved by supercomputers.

There exists several libraries which apply parallel computing efficiently. We used Accelerate [142], a library that enables the same Pytorch code to be run across any distributed configuration, for instance in terms of number of nodes and GPUs on a supercomputer, by adding just few lines of code for preparing and distributing the model on the given configuration. We integrated Accelerate

with DeepSpeed [143], a Deep Learning optimization library designed to efficiently scale models across distributed systems. In particular, from DeepSpeed we applied *Zero Redundancy Optimizer* (ZeRO) - Stage 3 [144], which makes use both of data parallelism and model sharding. Data parallelism distributes data across different nodes, while model sharding splits the model into smaller pieces. ZeRO is implemented in three progressive stages:

1. ZeRO-Stage 1 (Optimizer State Partitioning): shards the optimizer states, as variance and momentum, across GPUs. In this way we eliminate redundant memory allocations.

2. ZeRO-Stage 2 (Gradient Partitioning): it extends the previous stage by sharding also gradients across GPUs. In this way each process contains only a subset of the gradients, corresponding to its portion of the optimizer states.

3. ZeRO-Stage 3 (Parameter Partitioning): it extends the previous stages by sharding also model parameters across GPUs. In this way each process loads only the specific parameters needed for the current computational step during training.

In our experiments, we parallelized the code on more GPUs and on one or more compute nodes making use of Accelerate and DeepSpeed ZeRO-Stage 3. We also allow the GPUs to communicate with each other inside the different compute nodes, and the compute nodes to communicate with each other.

4.5 Experiments on Leonardo

In this final section, we provide several results on the experiments on Leonardo supercomputer. We applied two PEFT techniques, namely LoRA and SMT to the open-weight LLaMA-2-7B, varying the number of GPUs and trainable parameters. In particular, we updated at most 160M parameters, against the 7B required by full fine-tuning, thus, more than 97.7% of the weights was kept frozen. Then, we analyzed the results in terms of speedup and evaluated the fine-tuned model on the Big Bench Hard (BBH) dataset [145], comparing the performances and inference times achieved by the two techniques. The Pytorch code for our experiments was adapted from [146].

We downloaded the LLaMA-2-7B model and stored it on Leonardo. We fine-tuned this model on the instruction tuning dataset `tulu-v2-sft-mixture` [147], suitable for question answering and text summarization tasks and containing 326K samples. The dataset is fully in English and is a mixture of samples from different datasets, as:

- **FLAN** [148]: 100K samples to emphasize Chain-of-Thought reasoning.
- **ShareGPT** [149]: 114K samples containing conversations scraped via the ShareGPT API, including both user prompts and responses from OpenAI's ChatGPT.
- **Open Assistant 1** [150]: 7.7K human-generated, human-annotated assistant-style conversation messages.
- **Code-Alpaca** [151]: 20K samples for code generation.
- **Science examples**: 7.5K samples from a mixture of scientific document for understand tasks, including question answering, fact-checking, summarization, and information extraction.

We applied both PEFT techniques only to attention matrices, hence we kept frozen the fully connected layers. In particular, we applied LoRA with ranks 8, 16, 32 and 64, which correspond to 0.30%, 0.59%, 1.17% and 2.32% of the original 7 billion parameters of pre-trained LLaMA-2; thus, the maximum number of trainable parameters was approximately 160 millions. Similarly, we applied SMT with a similar percentage of trainable parameters to make the comparison of the two PEFT techniques more consistent.
Finally, we evaluated the fine-tuned model in terms of inference time and performances on the Big Bench Hard dataset, a collection of 23 tasks which require multi-step reasoning and can be collected into four macro-categories [145]:

- **Algorithmic and Multi-Step Arithmetic Reasoning**: these tasks require a varying level of arithmetical, logical, hierarchical, spatial and temporal reasoning capabilities.
- **Natural Language Understanding**: these tasks focus on semantic understanding, name disambiguation, entity resolution, grammar rules, or sarcasm detection.

- **Use of World Knowledge**: these tasks require factual and general knowledge about the world as well as the common practices and presuppositions in the Western society.
- **Multilingual Knowledge and Reasoning**: these tasks are based on translation quality estimation and cross-lingual natural-language inference.

Table 4.5 lists all the hyperparameters used for both techniques.

Hyperparameter	**LoRA**	**SMT**
Epochs	2	2
Global batch size	128	128
Batch size per GPU	1	1
Sequence length	2.048	2.048
Dropout rate	0.1	
α	16	
Weight decay	0	30
Warm-up ratio	0.03	0.03
Initial learning rate[3]	1e-5	1e-5

Table 4.5: Hyperparameters used for fine-tuning LLaMA-2-7B applying PEFT techniques LoRA and SMT.

We varied the number of GPUs per node, from 1 to 4, which is the highest possible on Leonardo for each node, and the number of nodes, from 1 to 4, thus the maximum number of GPUs used was 16. This was useful to study the speedup of the two PEFT techniques. We collect the results obtained in several tables. Results on the fine-tuning time needed on average per iteration using several numbers of GPUs for both PEFT techniques are reported in Table 4.6. The obtained speedups are shown in Figure 4.5: in the optimal theoretical case, the speedup using 1 processor and n processors is equal to n. Hence, by doubling the number of processors, then the training time should halve. However, empirically, this optimal speedup can not be achieved, since the time that GPUs use to communicate each other is much larger than the time needed for calculations. Results on the fine-tuning time needed on average per iteration using 16 GPUs and varying

[3]A linear learning rate decay following warm-up over the first 3% of the training steps is applied.

percentage of trainable parameters for both PEFT techniques are reported in Table 4.7. Results on the inference time needed on average per iteration using 16 processors, i.e. 4 nodes with 4 GPUs each, and varying percentage of trainable parameters for both PEFT techniques are reported in Table 4.8. Average evaluation results on BBH dataset varying percentage of trainable parameters for both techniques are reported in Table 4.9. The evaluation metric is exact match for all 23 tasks.

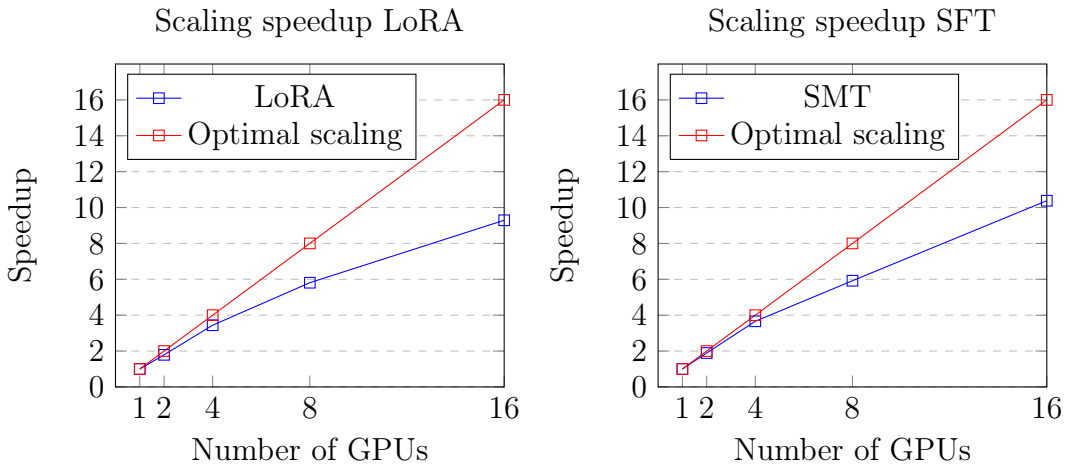

Figure 4.8: Speedup plots of LoRA and SMT related to Table 4.6.

	Number of GPUs				
	1	2	4	8	16
LoRA	45.35	25.29	13.17	7.81	4.88
SMT	46.91	24.88	12.84	7.92	4.52

Table 4.6: Average fine-tuning time per iteration (in seconds) of LLaMA-2-7B with LoRA and SMT, using 2.32% of trainable parameters and varying number of GPUs. Here only 100 training iterations were performed on a larger batch size, otherwise using 1 GPU with the given choice of hyperparameters would have required too much time.

	Trainable parameters (%)			
	0.30	0.59	1.17	2.32
LoRA	2.89	2.89	2.99	3.09
SMT	3.41	3.39	3.05	3.22

Table 4.7: Average fine-tuning time per iteration (in seconds) of LLaMA-2-7B with LoRA and SMT, using 16 GPUs and varying percentage of trainable parameters.

	Trainable parameters (%)			
	0.30	0.59	1.17	2.32
LoRA	198.72	199.85	201.08	227.39
SMT	225.88	211.67	208.13	196.23

Table 4.8: Average inference time per iteration (in seconds) of the fine-tuned LLaMA-2-7B with LoRA and SMT, using different percentages of trainable parameters.

	Trainable parameters (%)			
	0.30	0.59	1.17	2.32
LoRA	42.6	42.9	41.9	43.1
SMT	42.8	42.3	43.7	44.9

Table 4.9: Evaluation results in terms of exact match percentage (%) of the fine-tuned model LLaMA-2-7B with LoRA and SMT, using different percentages of trainable parameters.

As we can notice from Figure 4.5 both LoRA and SMT achieve similar speedups by updating 160 million parameters: by using 2 or 4 GPUs on the same compute node it is almost optimal, while with 8 or 16 GPUs the speedups significantly decreases, since the time required for communication between GPUs was much higher than the time required for computation. The final speedup with 16 GPUs was 9.3× and 10.4× with LoRA and SMT, respectively, against the optimal speedup of 16×, due to the high communication time between GPUs. However, by increasing number of processors and parallelizing the code, the training time diminishes, showing

a higher scalability of SMT than LoRA. Similar speedups were achieved using different percentage of trainable parameters.
Moreover, LoRA increases the fine-tuning time per iteration as increasing the number of trainable parameters: it is a reasonable result, since LoRA adds to each attention block two low-rank matrices, whose rank increases. On the other hand, SMT needs most time with few parameters, due to the different nature of the code. In general, both PEFT techniques are comparable, with LoRA a little faster than SMT during training but with lower speedup.
On the inference time, we observe that LoRA increases it gradually since the computation slows down when adding higher ranks matrices to the attention layers. On the other hand, SMT does the opposite and decreases the inference time by increasing the number of trainable parameters, due to the sparse approach. Evaluation results have shown that without fine-tuning the model obtains an average performance of 32.6%, hence both techniques improve significantly the performance. SMT achieves best performance of 44.9% with 2.32% trainable parameters, which is 2.1%, 2.6% and 1.2% points higher than applying SMT with only 0.30%, 0.59% and 1.17% parameters, respectively. Similarly for LoRA, where fine-tuning with 2.32% parameters achieves better results if compared to other smaller ranks. However, we can not state that SMT is better than LoRA, since, as remarked in the previous sections, fine-tuning relies strongly on the model's size, fine-tuning dataset and evaluation dataset. Our experiments had the only goal to compare the results of these two PEFT techniques in terms of number of GPUs used, training and inference time needed and evaluation results. We showed that matrix decomposition techniques are compute- and memory-efficient, allowing to significantly improve the performance of the pre-trained LLaMA-2-7B on the given downstream task.

Conclusions

The purpose of this thesis was to discuss several matrix factorization techniques for adapting LLMs to a downstream task. In particular, among all possible decompositions, we explored low-rank through LoRA and two of its variants, sparse through SMT and SVD through SVFT. In contrast to full fine-tuning, which requires an update of all model's weights, parameter-efficient techniques exploit the fact that recent LLMs are able to solve a wide variety of tasks yet and need only to modify a smaller percentage of their weights to improve the performances on a specific task. Moreover, due to the huge amount of computational resources required for solving these problems, parallel computing allows for a faster training time and a larger memory storage than serial computing.

In this thesis we studied the structure of an LLM, focusing on its key components and training process. At first, we introduced tokenization and embedding: the former allows to segment any input sequence into tokens, while the latter converts each token into a high-dimensional embedding space. We presented several tokenization techniques, both rule-based and trained, discussing the advantages and disadvantages of them. In particular, we focused on subword tokenization techniques, as BPE, Unigram and SentencePiece, which allow to segment each word into meaningful subwords and create a vocabulary representative for a certain language. On the other hand, since machines deal with vectors and not words, we mapped each token through a numerical embedding. In detail, if two tokens were semantically similar then also their embedding vectors achieved high cosine similarity. After that a given corpus text has been tokenized and embedded, it is passed through a series of transformer layers, where self-attention mechanisms play a crucial role in understanding relationships inside a text sequence, and, thus, understanding human language. Depending on the task, we can apply different transformer architectures as encoder, decoder and encoder-decoder. In order to evaluate the performance of an LLM, several evaluation metrics and benchmarks

were reported, as BLEU, ROUGE, PPL and GLUE. Due to the increasing size and capabilities of LLMs, the role of benchmarking is increasing its relevance: indeed, we listed the components of GLUE benchmark, which allows to evaluate an LLM on a great variety natural language understanding tasks. After that an LLM has been pre-trained, it can be fine-tuned in order to improve the performance on a given task or it can be adapted on a specific domain: since full fine-tuning is highly inefficient computationally, we illustrated different parameter-efficient fine-tuning techniques, as LoRA, SMT and SVFT. We underlined the role of matrix factorization techniques to apply fine-tuning in a compute- and memory-efficient way, in order to reduce the computational resources, speed up the training time and obtain comparable or higher performances than full fine-tuning. Moreover, HPC plays a crucial role for developing an LLM, whose training would require years on a classical computer and whose memory storage would be impossible. Different types of parallelism, as data and model parallelism, were the basis for achieving high performances. Experiments performed on the open-weight model LLaMA-2-7B with LoRA and SMT have confirmed the great importance of applying matrix factorization techniques for fine-tuning LLMs efficiently.
Further investigations on different models, datasets and tasks can be carried out to prove the importance of matrix factorization for fine-tuning an LLM on a wider range of contexts.

Appendix A

High Performance Computing

In this chapter, we give an overview of HPC clusters and why do we need them to solve modern problems; moreover, we see in detail the structure of a cluster and focus on the concept of parallelism. A supercomputer, also called *cluster*, is a collection of hundreds or thousands servers, called *nodes*, connected through a high speed and low latency network. The use of these computers allows us to study very complex physical system in different scientific fields at all scales, from a macroscopic point of view (like weather forecasting or evolution of the Universe) to a microscopic one (like chip or drug design). Sometimes, experimenting a real simulation of a system would be too dangerous or expensive, such as a fault simulation or a crash analysis: a virtual simulation would be profitable. All these problems require a huge amount of data and computational power: supercomputers can execute such tasks thanks to a high degree of parallelization, providing fast and reliable answers.

A.1 Structure and workflow of a supercomputer

The main structural components of a supercomputer are the following:

- **Login Nodes**: the user can access the cluster via ssh to perform operations like moving files and data or running a *jobscript*, which is a series of instructions indicating the desired resources (CPUs, GPUs, time, memory). The main function of these nodes is to interact with the scheduler; however, they are not designed for running tasks. Login nodes have access to the Internet, hence they are useful to download data.

- **Scheduler and Master Node**: the master node is the "management" node for the cluster and runs the scheduler; the scheduler distributes the computational resources among all users, monitoring the jobs currently running on the supercomputer and assigning pending jobs to the compute nodes. The most frequently used (Nov. 2021) workload manager is Simple Linux Utility for Resource Management, or simply SLURM, which can handle up to 1000 job submissions and 600 job executions per second.

- **Compute Nodes**: these nodes provide computing power and run user's algorithms; each compute node is a server equipped with CPUs, memory and sometimes GPUs, and is specialized for computation.

- **Parallel Filesystem**: a filesystem manages how files are stored on disks; in a parallel architecture it performs I/O operations, allowing simultaneous access from multiple nodes to the filesystem. It uses technology like GPFS, LUSTRE and BeeGFS.

- **High performance network**: the nodes are interconnected via a high-speed network, usually InfiniBand or Ethernet. It is able to transfer large quantities of data in small time due to its large bandwidth and low latency.

Figure A.1 sums up a general HPC cluster structure.

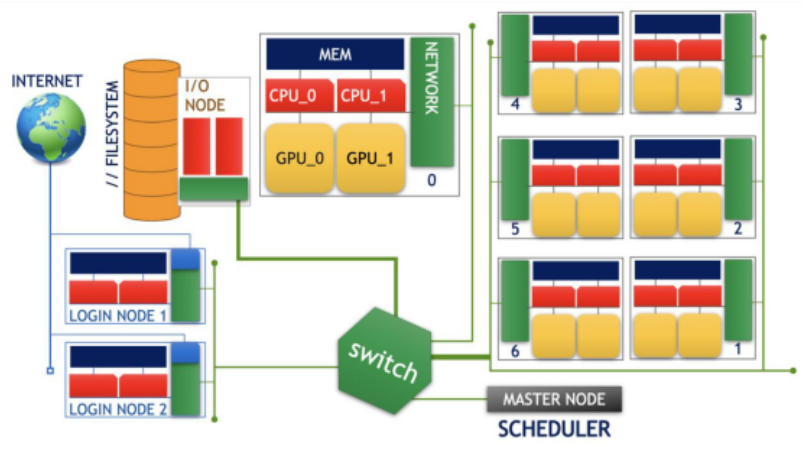

Figure A.1: General supercomputer's architecture with the main components described above. Image from [152].

The computational power of an HPC cluster relies on the compute nodes, which are equipped with two or more CPUs, a great amount of memory and sometimes with one or more GPUs:

- **Central Processing Unit** (CPU): its main tasks are to perform computations and manage the data necessary for running applications; every CPU has from 32 to 56 cores and is connected to the server memory and to I/O devices.
- **Graphics Processing Unit** (GPU): it was originally designed for accelerating computer graphics and for tasks, for which the same set of mathematical calculations has to be repeated on every pixel. Now, GPUs are widely used since they can compute a much higher number of FLOPS[1] than what CPUs can do, therefore it can achieve a high level of parallelism. On the other hand, a GPU has low memory, hence the data have to pass through the CPU: this process introduces high latency and can lower the computational performance.

Now, we briefly describe the most important steps to access the cluster. The user logs in through Internet to the cluster and gets access to one of the login nodes, where he can interact with the SLURM scheduler. Here, he can submit jobs to the compute nodes in an *interactive* or *batch* way: in the first one, the resources are required directly to the compute node, while in the second way, the resources are demanded in a file called *jobscript*. Finally, the scheduler adds in a queuing line every submitted job together with the jobs of other users; as soon as the resources are available, the jobscript will be executed.

A jobscript is a series of commands containing SLURM directives: the following jobscript asks 200GB and 1 GPU of a compute node for 30 minutes to run the program "myexecutable". The standard output and error will be directed in the files job.out and job.err respectively. The number of CPUs per node is expressed through the SLURM directive ntasks-per-node.

```
#!/bin/bash
#SBATCH --job-name=myjob
#SBATCH --output=job.out
#SBATCH --error=job.err
```

[1]Floating Point Operation Per Second

```
#SBATCH --time=00:30:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=24
#SBATCH --mem=200GB
#SBATCH --gres=gpu:1
#SBATCH --partition=partition_name
#SBATCH --account=account_name
#SBATCH --mail-type=ALL
#SBATCH --mail-user=user@email.com

mpirun ./myexecutable
```

A.2 Parallelism

Traditionally, softwares have been written for *serial* computing: briefly, a problem is divided into a sequence of instructions which are executed one after the other on a single processor. Using serial computing to solve modern complex problems with a huge amount of data would be practically impossible both in terms of time and cost: a solution is exploiting *parallel* computing. In this way, a problem is divided in different parts which can be executed simultaneously: each part is broken into a series of instructions. Instructions from different parts run at the same time on different processors, allowing an improvement in the performance.

Not every problem can be parallelized, as it depends on the nature of the problem itself. It is possible to implement different types of parallelism:

- **Shared memory parallelism** (threading with library OpenMP): a work is divided between multiple processes which run on the same machine and share the same memory. This allows an efficient and fast communication between processing units; although, any change made by one processing unit to a shared variable has effects to all the other processing units that access that variable, which could create overlapping problems.

- **Distributed memory parallelism** (tasking with library MPI[2]): each processor has its own memory and communicates with others processors through

[2]Message Passing Interface

MPI. Although we have a large memory at disposal, the main disadvantage of MPI is the slow communication, which could cause a performance loss.

- **Hybrid parallelism**: it is a combination of the previous two models, using both threading within each node and MPI tasking between nodes. It is the most used in real applications and achieves high performances.

To establish the efficiency of a parallel program we introduce the concepts of *speedup* S and *efficiency* E, defined as follows:

$$\mathrm{S} = \frac{\text{Time for serial}}{\text{Time for parallel}}$$
$$\mathrm{E} = \frac{\mathrm{S}}{\text{Number of cores}}.$$

The speedup compares the wall-clock time for a serial program with a parallel that fulfills the same job and is limited by a couple of factors. First, speedup is generally limited by the speed of the slowest processor, thus we need that the whole system is load balanced. Secondly, if the communication and computation can not be overlapped, also partially, then the communication will reduce the speed of the application. A final limitation is given by Amdahl's Law (1967): program's serial parts limit the potential speedup from parallelizing code, therefore not only the communication time can slow down the performance, but also the execution time of serial parts which can not be parallelized. For an ideal system with n processors the speedup is equal to n: however this does not happen because, for instance, we have to take into account the time for communication. Dividing the speedup by the number of processors we get a number between 0 and 1 (in the ideal case we get 1), hence we can interpret the efficiency as a measure of the percentage of time for which a processor is used effectively.

In general, we have also to consider the problem's dimension: the bigger it is, the higher will be the computing time. The *scalability* of a program refers to its ability to handle an increasing amount of data without a significant increase in computational requirements or decrease in performance. In HPC we have two ways to measure it:

- **Strong scaling**: the problem size is kept unchanged, while the number of tasks assigned to solve the problem increases.
- **Weak scaling**: the problem size increases together with the number of tasks, hence the work per task is constant.

A.3 Supercomputing in Cineca

Cineca is a non profit Consortium, made up of 118 members: the Italian Ministry of Education and Merit, the Italian Ministry of Universities and Research, 70 Italian universities and 46 Italian National Institutions and agencies. Today it is the largest Italian computing centre, one of the most important worldwide. It operates in the technological transfer sector through high performance scientific computing, the management and development of web based services to the Italian academic system, and the development of complex information systems for treating large amounts of data for public administrations and health care institutions [8].

1. **GALILEO100** [153]: It was co-funded by the European ICEI (Interactive Computing e-Infrastructure) for scientific research and installed in August 2021 to replace GALILEO. Engineered by DELL, it consists of 636 computing nodes each with 2 CPUs Intel CascadeLake 8260 with 24 cores each, 2.4 GHz and 384GB RAM. The compute nodes are divided in 422 standard nodes (called *thin* nodes), 180 data processing nodes (called *fat* nodes) and 34 GPU nodes. The internal network is a Mellanox Infiniband 100GbE. It can achieve a theoretical peak performance of about 2 PFLOPS.

2. **LEONARDO** [154]: It is currently hosted by Cineca and built in the Bologna Technopole; it was classified in 4th position among the most powerful supercomputers in the Top500 List of November 2022. LEONARDO's main goal is to support European academic and industrial researchers to develop applications able to face with urgent challenges, such as climate change, pandemics and prediction of extreme events. It has a storage of about 110 PetaBytes and consists of two partitions:

    - a *booster* partition with 3456 nodes, 4 GPUs NVIDIA A100 SXM6 64GB and a single CPU Intel Ice Lake with 32 cores. It achieves a theoretical peak performance of about 240 PFLOPS; it is mostly used to maximize the computational capacity.
    - a *Data Centric General Purpose* (DCGP) partition with 1536 nodes, 2 CPUs Intel Sapphire Rapids each with 56 cores. It achieves a theoretical peak performance of about 9 PFLOPS.

Bibliography

[1] "LLaMA." [Online]. Available: https://www.llama.com/

[2] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "PaLM: Scaling Language Modeling with Pathways," 2022. [Online]. Available: https://arxiv.org/abs/2204.02311

[3] OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, R. Avila, I. Babuschkin, S. Balaji, V. Balcom, P. Baltescu, H. Bao, M. Bavarian, J. Belgum, I. Bello, J. Berdine, G. Bernadett-Shapiro, C. Berner, L. Bogdonoff, O. Boiko, M. Boyd, A.-L. Brakman, G. Brockman, T. Brooks, M. Brundage, K. Button, T. Cai, R. Campbell, A. Cann, B. Carey, C. Carlson, R. Carmichael, B. Chan, C. Chang, F. Chantzis, D. Chen, S. Chen, R. Chen, J. Chen, M. Chen, B. Chess, C. Cho, C. Chu, H. W. Chung, D. Cummings, J. Currier, Y. Dai, C. Decareaux, T. Degry, N. Deutsch, D. Deville, A. Dhar, D. Dohan, S. Dowling, S. Dunning, A. Ecoffet, A. Eleti, T. Eloundou, D. Farhi, L. Fedus, N. Felix, S. P. Fishman, J. Forte, I. Fulford, L. Gao, E. Georges, C. Gibson, V. Goel, T. Gogineni, G. Goh, R. Gontijo-Lopes, J. Gordon, M. Grafstein, S. Gray,

R. Greene, J. Gross, S. S. Gu, Y. Guo, C. Hallacy, J. Han, J. Harris, Y. He, M. Heaton, J. Heidecke, C. Hesse, A. Hickey, W. Hickey, P. Hoeschele, B. Houghton, K. Hsu, S. Hu, X. Hu, J. Huizinga, S. Jain, S. Jain, J. Jang, A. Jiang, R. Jiang, H. Jin, D. Jin, S. Jomoto, B. Jonn, H. Jun, T. Kaftan, Łukasz Kaiser, A. Kamali, I. Kanitscheider, N. S. Keskar, T. Khan, L. Kilpatrick, J. W. Kim, C. Kim, Y. Kim, J. H. Kirchner, J. Kiros, M. Knight, D. Kokotajlo, Łukasz Kondraciuk, A. Kondrich, A. Konstantinidis, K. Kosic, G. Krueger, V. Kuo, M. Lampe, I. Lan, T. Lee, J. Leike, J. Leung, D. Levy, C. M. Li, R. Lim, M. Lin, S. Lin, M. Litwin, T. Lopez, R. Lowe, P. Lue, A. Makanju, K. Malfacini, S. Manning, T. Markov, Y. Markovski, B. Martin, K. Mayer, A. Mayne, B. McGrew, S. M. McKinney, C. McLeavey, P. McMillan, J. McNeil, D. Medina, A. Mehta, J. Menick, L. Metz, A. Mishchenko, P. Mishkin, V. Monaco, E. Morikawa, D. Mossing, T. Mu, M. Murati, O. Murk, D. Mély, A. Nair, R. Nakano, R. Nayak, A. Neelakantan, R. Ngo, H. Noh, L. Ouyang, C. O'Keefe, J. Pachocki, A. Paino, J. Palermo, A. Pantuliano, G. Parascandolo, J. Parish, E. Parparita, A. Passos, M. Pavlov, A. Peng, A. Perelman, F. de Avila Belbute Peres, M. Petrov, H. P. de Oliveira Pinto, Michael, Pokorny, M. Pokrass, V. H. Pong, T. Powell, A. Power, B. Power, E. Proehl, R. Puri, A. Radford, J. Rae, A. Ramesh, C. Raymond, F. Real, K. Rimbach, C. Ross, B. Rotsted, H. Roussez, N. Ryder, M. Saltarelli, T. Sanders, S. Santurkar, G. Sastry, H. Schmidt, D. Schnurr, J. Schulman, D. Selsam, K. Sheppard, T. Sherbakov, J. Shieh, S. Shoker, P. Shyam, S. Sidor, E. Sigler, M. Simens, J. Sitkin, K. Slama, I. Sohl, B. Sokolowsky, Y. Song, N. Staudacher, F. P. Such, N. Summers, I. Sutskever, J. Tang, N. Tezak, M. B. Thompson, P. Tillet, A. Tootoonchian, E. Tseng, P. Tuggle, N. Turley, J. Tworek, J. F. C. Uribe, A. Vallone, A. Vijayvergiya, C. Voss, C. Wainwright, J. J. Wang, A. Wang, B. Wang, J. Ward, J. Wei, C. Weinmann, A. Welihinda, P. Welinder, J. Weng, L. Weng, M. Wiethoff, D. Willner, C. Winter, S. Wolrich, H. Wong, L. Workman, S. Wu, J. Wu, M. Wu, K. Xiao, T. Xu, S. Yoo, K. Yu, Q. Yuan, W. Zaremba, R. Zellers, C. Zhang, M. Zhang, S. Zhao, T. Zheng, J. Zhuang, W. Zhuk, and B. Zoph, "GPT-4 Technical Report," 2024. [Online]. Available: https://arxiv.org/abs/2303.08774

[4] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-

Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," *CoRR*, vol. abs/2005.14165, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[5] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, and W. Chen, "LoRA: Low-Rank Adaptation of Large Language Models," *CoRR*, vol. abs/2106.09685, 2021. [Online]. Available: https://arxiv.org/abs/2106.09685

[6] H. He, J. B. Li, X. Jiang, and H. Miller, "Sparse Matrix in Large Language Model Fine-tuning," 2024. [Online]. Available: https://arxiv.org/abs/2405.15525

[7] V. Lingam, A. Tejaswi, A. Vavre, A. Shetty, G. K. Gudur, J. Ghosh, A. Dimakis, E. Choi, A. Bojchevski, and S. Sanghavi, "SVFT: Parameter-Efficient Fine-Tuning with Singular Vectors," 2024. [Online]. Available: https://arxiv.org/abs/2405.19597

[8] "Cineca organization." [Online]. Available: https://www.cineca.it/en/about-us/organization

[9] "Top500 list." [Online]. Available: https://www.top500.org/

[10] "Tokenization in Natural Language Processing: Methods, Types, and Challenges." [Online]. Available: https://www.solulab.com/tokenization-nlp/

[11] "What are some challenges faced by tokenizers in natural language processing, including issues like ambiguity, word contractions, named entities, and out-of-vocabulary words?" [Online]. Available: https://medium.com/@sujathamudadla1213/what-are-some-challenges-faced-by-tokenizers-in-natural-language-processing-including-issues-like-a065d09445f1

[12] "What is Tokenization?" [Online]. Available: https://www.datacamp.com/blog/what-is-tokenization

[13] "Tokenization in NLP: Types, Challenges, Examples, Tools." [Online]. Available: https://neptune.ai/blog/tokenization-in-nlp

[14] "The Art of Tokenization in Text Preprocessing." [Online]. Available: https://cognitivecreator.medium.com/the-art-of-tokenization-in-text-preprocessing-day2-0b8185bab488

[15] "Word, Subword, and Character-Based Tokenization: Know the Difference." [Online]. Available: https://towardsdatascience.com/word-subword-and-character-based-tokenization-know-the-difference-ea0976b64e17

[16] "What is Out-of-Vocabulary (OOV) Words: LLMs Explained." [Online]. Available: https://www.chatgptguide.ai/2024/03/01/what-is-out-of-vocabulary-oov-words-llms-explained/

[17] "Regex Tutorial – How to write Regular Expressions?" [Online]. Available: https://www.geeksforgeeks.org/write-regular-expressions/

[18] "Regular Expressions (Regex)." [Online]. Available: https://www3.ntu.edu.sg/home/ehchua/programming/howto/Regexe.html

[19] "NLP | How tokenizing text, sentence, words works." [Online]. Available: https://www.geeksforgeeks.org/nlp-how-tokenizing-text-sentence-words-works/

[20] "Byte-Pair Encoding: Subword-based tokenization algorithm." [Online]. Available: https://towardsdatascience.com/byte-pair-encoding-subword-based-tokenization-algorithm-77828a70bee0

[21] "Summary of the tokenizers." [Online]. Available: https://huggingface.co/docs/transformers/tokenizer_summary

[22] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[23] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language Models are Unsupervised Multitask Learners," 2019. [Online]. Available: https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[24] "Byte-Pair Encoding For Beginners." [Online]. Available: https://towardsdatascience.com/byte-pair-encoding-for-beginners-708d4472c0c7

[25] P. Gage, "A New Algorithm for Data Compression," 1994. [Online]. Available: http://www.pennelynn.com/Documents/CUJ/HTML/94HTML/19940045.HTM

[26] "Byte-Pair Encoding (BPE) in NLP." [Online]. Available: https://www.geeksforgeeks.org/byte-pair-encoding-bpe-in-nlp/

[27] "Normalization and pre-tokenization." [Online]. Available: https://huggingface.co/learn/nlp-course/en/chapter6/4

[28] "Byte-Pair Encoding." [Online]. Available: https://www.youtube.com/watch?v=tOMjTCO0htA

[29] "Byte-Pair Encoding tokenization." [Online]. Available: https://huggingface.co/learn/nlp-course/chapter6/5

[30] "Byte-Pair Encoding." [Online]. Available: https://en.wikipedia.org/wiki/Byte_pair_encoding

[31] K. Bostrom and G. Durrett, "Byte Pair Encoding is Suboptimal for Language Model Pretraining," *CoRR*, vol. abs/2004.03720, 2020. [Online]. Available: https://arxiv.org/abs/2004.03720

[32] M. Schuster and K. Nakajima, "Japanese and Korean Voice Search," in *International Conference on Acoustics, Speech and Signal Processing*, 2012, pp. 5149–5152. [Online]. Available: https://static.googleusercontent.com/media/research.google.com/it//pubs/archive/37842.pdf

[33] "WordPiece: Subword-based tokenization algorithm." [Online]. Available: https://towardsdatascience.com/wordpiece-subword-based-tokenization-algorithm-1fbd14394ed7

[34] "WordPiece." [Online]. Available: https://h2o.ai/wiki/wordpiece/

[35] "WordPiece tokenization." [Online]. Available: https://huggingface.co/learn/nlp-course/chapter6/6

[36] T. Kudo, "Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates," *CoRR*, vol. abs/1804.10959, 2018. [Online]. Available: http://arxiv.org/abs/1804.10959

[37] "Unigram language based subword segmentation." [Online]. Available: https://machinelearnit.com/2018/03/01/unigram-language-based-subword-segmentation/

[38] "N-gram language models." [Online]. Available: https://medium.com/mti-technology/n-gram-language-model-b7c2fc322799

[39] "Unigram tokenization." [Online]. Available: https://huggingface.co/learn/nlp-course/chapter6/7

[40] "On Subword Units." [Online]. Available: https://everdark.github.io/k9/notebooks/ml/natural_language_understanding/subword_units/subword_units.nb.html#123_em_with_viterbi

[41] T. Kudo and J. Richardson, "SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing," *CoRR*, vol. abs/1808.06226, 2018. [Online]. Available: http://arxiv.org/abs/1808.06226

[42] "SentencePiece." [Online]. Available: https://github.com/google/sentencepiece

[43] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A Robustly Optimized BERT Pretraining Approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[44] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension," *CoRR*, vol. abs/1910.13461, 2019. [Online]. Available: http://arxiv.org/abs/1910.13461

[45] P. He, X. Liu, J. Gao, and W. Chen, "DeBERTa: Decoding-enhanced BERT with Disentangled Attention," *CoRR*, vol. abs/2006.03654, 2020. [Online]. Available: https://arxiv.org/abs/2006.03654

[46] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter," *CoRR*, vol. abs/1910.01108, 2019. [Online]. Available: http://arxiv.org/abs/1910.01108

[47] Z. Sun, H. Yu, X. Song, R. Liu, Y. Yang, and D. Zhou, "MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices," *CoRR*, vol. abs/2004.02984, 2020. [Online]. Available: https://arxiv.org/abs/2004.02984

[48] Z. Dai, G. Lai, Y. Yang, and Q. V. Le, "Funnel-Transformer: Filtering out Sequential Redundancy for Efficient Language Processing," *CoRR*, vol. abs/2006.03236, 2020. [Online]. Available: https://arxiv.org/abs/2006.03236

[49] K. Song, X. Tan, T. Qin, J. Lu, and T. Liu, "MPNet: Masked and Permuted Pre-training for Language Understanding," *CoRR*, vol. abs/2004.09297, 2020. [Online]. Available: https://arxiv.org/abs/2004.09297

[50] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "ALBERT: A Lite BERT for Self-supervised Learning of Language Representations," *CoRR*, vol. abs/1909.11942, 2019. [Online]. Available: http://arxiv.org/abs/1909.11942

[51] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer," *CoRR*, vol. abs/1910.10683, 2019. [Online]. Available: http://arxiv.org/abs/1910.10683

[52] Y. Liu, J. Gu, N. Goyal, X. Li, S. Edunov, M. Ghazvininejad, M. Lewis, and L. Zettlemoyer, "Multilingual Denoising Pre-training for Neural Machine Translation," *CoRR*, vol. abs/2001.08210, 2020. [Online]. Available: https://arxiv.org/abs/2001.08210

[53] M. Zaheer, G. Guruganesh, A. Dubey, J. Ainslie, C. Alberti, S. Ontañón, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed, "Big Bird: Transformers for Longer Sequences," *CoRR*, vol. abs/2007.14062, 2020. [Online]. Available: https://arxiv.org/abs/2007.14062

[54] Z. Yang, Z. Dai, Y. Yang, J. G. Carbonell, R. Salakhutdinov, and Q. V. Le, "XLNet: Generalized Autoregressive Pretraining for Language Understanding," *CoRR*, vol. abs/1906.08237, 2019. [Online]. Available: http://arxiv.org/abs/1906.08237

[55] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "LLaMA: Open and Efficient Foundation

Language Models," 2023. [Online]. Available: https://arxiv.org/abs/2302.13971

[56] "Mistral AI." [Online]. Available: https://mistral.ai/

[57] E. Federici, M. Ferraretto, and N. Landro. (2024) Gazzetta Ufficiale: A Dataset of Legislative Texts, Public and Private Acts. [Online]. Available: https://huggingface.co/datasets/mii-llm/gazzetta-ufficiale

[58] "Better language models and their implications." [Online]. Available: https://openai.com/index/better-language-models/

[59] "Italian BioBERT." [Online]. Available: https://huggingface.co/datasets/IVN-RIN/BioBERT_Italian

[60] "Training a new tokenizer from an old one." [Online]. Available: https://huggingface.co/learn/nlp-course/chapter6/2

[61] S. J. Prince, *Understanding Deep Learning*. The MIT Press, 2023. [Online]. Available: http://udlbook.com

[62] "A Beginner's Guide to Tokens, Vectors, and Embeddings in NLP." [Online]. Available: https://saschametzger.com/blog/what-are-tokens-vectors-and-embeddings-how-do-you-create-them

[63] "Word embeddings: the (very) basics." [Online]. Available: https://corpling.hypotheses.org/495

[64] "Understanding Cosine Similarity and Word Embeddings." [Online]. Available: https://spencerporter2.medium.com/understanding-cosine-similarity-and-word-embeddings-dbf19362a3c

[65] "Word Embedding." [Online]. Available: https://lilianweng.github.io/posts/2017-10-15-word-embedding/

[66] "CBOW vs Skip-Gram." [Online]. Available: https://www.baeldung.com/cs/word-embeddings-cbow-vs-skip-gram

[67] "Introduction to Word Embedding." [Online]. Available: https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa

[68] "Word2Vec Tutorial — The Skip-Gram Model." [Online]. Available: https://medium.com/nearist-ai/word2vec-tutorial-the-skip-gram-model-c7926e1fdc09

[69] "word2vec tensorflow." [Online]. Available: https://www.tensorflow.org/text/tutorials/word2vec

[70] X. Rong, "word2vec Parameter Learning Explained," *CoRR*, vol. abs/1411.2738, 2014. [Online]. Available: http://arxiv.org/abs/1411.2738

[71] "Skip Gram and CBOW architectures." [Online]. Available: https://it.wikipedia.org/wiki/Word2vec

[72] "Why in Word2Vec model, the hidden layer has no activation?" [Online]. Available: https://www.reddit.com/r/MachineLearning/comments/cgw7ab/d_why_in_word2vec_model_the_hidden_layer_has_no/

[73] "Demystifying Neural Network in Skip-Gram Language Modeling." [Online]. Available: https://aegis4048.github.io/demystifying_neural_network_in_skip_gram_language_modeling

[74] "Understanding the Continuous Bag of Words (CBOW) Model: Architecture, Working Mechanism and Math Behind It | Natural language processing." [Online]. Available: https://linkedin.com/pulse/understanding-continuous-bag-words-cbow-model-working-anjil-adhikari-zueff

[75] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and their Compositionality," *CoRR*, vol. abs/1310.4546, 2013. [Online]. Available: http://arxiv.org/abs/1310.4546

[76] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[77] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional Sequence to Sequence Learning," *CoRR*, vol. abs/1705.03122, 2017. [Online]. Available: http://arxiv.org/abs/1705.03122

[78] "Self-attention." [Online]. Available: https://h2o.ai/wiki/self-attention/

[79] "Attention Mechanism in LLMs: An Intuitive Explanation." [Online]. Available: https://www.datacamp.com/blog/attention-mechanism-in-llms-intuition

[80] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[81] M. A. Georgios Nanos, "Why Are Residual Connections Important in Transformer Architectures?" [Online]. Available: https://www.baeldung.com/cs/transformer-networks-residual-connections

[82] "Introduction to ResNets." [Online]. Available: https://towardsdatascience.com/introduction-to-resnets-c0a830a288a4

[83] F. June, "BatchNorm and LayerNorm." [Online]. Available: https://medium.com/@florian_algo/batchnorm-and-layernorm-2637f46a998b

[84] M. Hoque, "Demystifying Neural Network Normalization Techniques." [Online]. Available: https://medium.com/@minh.hoque/demystifying-neural-network-normalization-techniques-4a21d35b14f8

[85] "Layer Normalization." [Online]. Available: https://nn.labml.ai/normalization/layer_norm/index.html

[86] "Batch Normalization." [Online]. Available: https://nn.labml.ai/normalization/batch_norm/index.html

[87] S. Shen, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "Rethinking Batch Normalization in Transformers," *CoRR*, vol. abs/2003.07845, 2020. [Online]. Available: https://arxiv.org/abs/2003.07845

[88] S. Islam, H. Elmekki, A. Elsebai, J. Bentahar, N. Drawel, G. Rjoub, and W. Pedrycz, "A Comprehensive Survey on Applications of Transformers for Deep Learning Tasks," 2023. [Online]. Available: https://arxiv.org/abs/2306.07303

[89] "What Transformers can do." [Online]. Available: https://huggingface.co/docs/transformers/task_summary

[90] "Encoder models." [Online]. Available: https://huggingface.co/learn/nlp-course/en/chapter1/5

[91] "BERT Explained: State of the art language model for NLP." [Online]. Available: https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270

[92] "BERT MLM - 80% [MASK], 10% random words and 10% same word - how does this work?" [Online]. Available: https://stats.stackexchange.com/questions/575002/bert-mlm-80-mask-10-random-words-and-10-same-word-how-does-this-work

[93] "BERT For Next Sentence Prediction." [Online]. Available: https://towardsdatascience.com/bert-for-next-sentence-prediction-466b67f8226f

[94] Zain ul Abideen, "Autoregressive Models for Natural Language Processing." [Online]. Available: https://medium.com/@zaiinn440/autoregressive-models-for-natural-language-processing-b95e5f933e1f

[95] "How does GPT-3 spend its 175B parameters?" [Online]. Available: https://www.lesswrong.com/posts/3duR8CrvcHywrnhLo/how-does-gpt-3-spend-its-175b-parameters

[96] "Hallucination in Large Language Models." [Online]. Available: https://medium.com/@asheshnathmishra/hallucination-in-large-language-models-2023-f7b4e77855ae

[97] S. Raschka, "Developing an LLM: Building, Training, Finetuning." [Online]. Available: https://github.com/rasbt/LLMs-from-scratch

[98] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: a Method for Automatic Evaluation of Machine Translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, P. Isabelle, E. Charniak, and D. Lin, Eds. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040

[99] Ketan Doshi, "Foundations of NLP Explained — Bleu Score and WER Metrics." [Online]. Available: https://towardsdatascience.com/foundations-of-nlp-explained-bleu-score-and-wer-metrics-1a5ba06d812b

[100] C.-Y. Lin, "ROUGE: A Package for Automatic Evaluation of Summaries," in *Text Summarization Branches Out*. Barcelona, Spain: Association

for Computational Linguistics, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013

[101] S. M. Walker, "What is the ROUGE Score (Recall-Oriented Understudy for Gisting Evaluation)?" [Online]. Available: https://klu.ai/glossary/rouge-score

[102] F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker, "Perplexity—a measure of the difficulty of speech recognition tasks," *The Journal of the Acoustical Society of America*, vol. 62, no. S1, pp. S63–S63, 1977.

[103] "Perplexity of fixed-length models." [Online]. Available: https://huggingface.co/docs/transformers/perplexity

[104] "Perplexity." [Online]. Available: https://docs.kolena.com/metrics/perplexity/

[105] "Perplexity." [Online]. Available: https://www.larksuite.com/en_us/topics/ai-glossary/perplexity

[106] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding," 2019, in the Proceedings of ICLR.

[107] S. Mudadla, "What is Glue Benchmark in Natural language processing?" [Online]. Available: https://medium.com/@sujathamudadla1213/what-is-glue-benchmark-in-natural-language-processing-8ec14dd54db0

[108] "GLUE: A multi-task benchmark and analysis platform for natural language understanding." [Online]. Available: https://openreview.net/pdf?id=rJ4km2R5t7

[109] "CoLA." [Online]. Available: https://nyu-mll.github.io/CoLA/

[110] "SST-2." [Online]. Available: https://github.com/YJiangcm/SST-2-sentiment-analysis

[111] "MRPC." [Online]. Available: ttps://www.microsoft.com/en-us/download/details.aspx?id=52398

[112] "STSB." [Online]. Available: https://ixa2.si.ehu.eus/stswiki/index.php/STSbenchmark

[113] "First Quora Dataset Release: Question Pairs." [Online]. Available: https://quoradata.quora.com/First-Quora-Dataset-Release-Question-Pairs

[114] "MultiNLI." [Online]. Available: https://cims.nyu.edu/~sbowman/multinli/

[115] "SQuAD2.0. The Stanford Question Answering Dataset." [Online]. Available: https://rajpurkar.github.io/SQuAD-explorer/

[116] "Recognizing Textual Entailment." [Online]. Available: https://aclweb.org/aclwiki/Recognizing_Textual_Entailment

[117] "The Winograd Schema Challenge." [Online]. Available: https://cs.nyu.edu/~davise/papers/WinogradSchemas/WS.html

[118] A. Wang, Y. Pruksachatkun, N. Nangia, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems," *CoRR*, 2019. [Online]. Available: http://arxiv.org/abs/1905.00537

[119] "LLaMA." [Online]. Available: https://github.com/meta-llama/llama-models/blob/main/models/llama3_1/MODEL_CARD.md

[120] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt, "Measuring Massive Multitask Language Understanding," *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

[121] A. Talmor, J. Herzig, N. Lourie, and J. Berant, "CommonsenseQA: A Question Answering Challenge Targeting Commonsense Knowledge," 2019. [Online]. Available: https://arxiv.org/abs/1811.00937

[122] M. Joshi, E. Choi, D. S. Weld, and L. Zettlemoyer, "TriviaQA: A Large Scale Distantly Supervised Challenge Dataset for Reading Comprehension," *CoRR*, vol. abs/1705.03551, 2017. [Online]. Available: http://arxiv.org/abs/1705.03551

[123] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ Questions for Machine Comprehension of Text," *CoRR*, vol. abs/1606.05250, 2016. [Online]. Available: http://arxiv.org/abs/1606.05250

[124] D. Dua, Y. Wang, P. Dasigi, G. Stanovsky, S. Singh, and M. Gardner, "DROP: A Reading Comprehension Benchmark Requiring Discrete Reasoning Over Paragraphs," *CoRR*, vol. abs/1903.00161, 2019. [Online]. Available: http://arxiv.org/abs/1903.00161

[125] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[126] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, C. Hesse, and J. Schulman, "Training Verifiers to Solve Math Word Problems," *arXiv preprint arXiv:2110.14168*, 2021.

[127] D. Paperno, G. Kruszewski, A. Lazaridou, Q. N. Pham, R. Bernardi, S. Pezzelle, M. Baroni, G. Boleda, and R. Fernández, "The LAMBADA dataset: Word prediction requiring a broad discourse context," *CoRR*, vol. abs/1606.06031, 2016. [Online]. Available: http://arxiv.org/abs/1606.06031

[128] "Statistical and Neural Machine Translation." [Online]. Available: https://www.statmt.org/

[129] Y. Bisk, R. Zellers, R. L. Bras, J. Gao, and Y. Choi, "PIQA: Reasoning about Physical Commonsense in Natural Language," in *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.

[130] P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord, "Think you have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge," *CoRR*, vol. abs/1803.05457, 2018. [Online]. Available: http://arxiv.org/abs/1803.05457

[131] "The Ultimate Guide to Parameter-efficient Fine-tuning (PEFT)." [Online]. Available: https://kanerika.com/blogs/parameter-efficient-fine-tuning/

[132] S. Hayou, N. Ghosh, and B. Yu, "LoRA+: Efficient Low Rank Adaptation of Large Models," 2024. [Online]. Available: https://arxiv.org/abs/2402.12354

[133] H. Zhou, X. Lu, W. Xu, C. Zhu, T. Zhao, and M. Yang, "LoRA-drop: Efficient LoRA Parameter Pruning based on Output Evaluation," 2024. [Online]. Available: https://arxiv.org/abs/2402.07721

[134] A. Aghajanyan, L. Zettlemoyer, and S. Gupta, "Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning," *CoRR*, vol. abs/2012.13255, 2020. [Online]. Available: https://arxiv.org/abs/2012.13255

[135] V. Zhong, C. Xiong, and R. Socher, "Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning," *CoRR*, vol. abs/1709.00103, 2017.

[136] C. Zhu, A. S. Rawat, M. Zaheer, S. Bhojanapalli, D. Li, F. X. Yu, and S. Kumar, "Modifying Memories in Transformer Models," *CoRR*, vol. abs/2012.00363, 2020. [Online]. Available: https://arxiv.org/abs/2012.00363

[137] K. Meng, D. Bau, A. Andonian, and Y. Belinkov, "Locating and Editing Factual Associations in GPT," 2023. [Online]. Available: https://arxiv.org/abs/2202.05262

[138] K. Meng, A. S. Sharma, A. Andonian, Y. Belinkov, and D. Bau, "Mass-Editing Memory in a Transformer," 2023. [Online]. Available: https://arxiv.org/abs/2210.07229

[139] "OpenAI's GPT-3 Language Model: A Technical Overview." [Online]. Available: https://lambdalabs.com/blog/demystifying-gpt-3?srsltid=AfmBOoqBIfOt1Ktjlk2BJqUcZH_8ZcXQ1t67iG2D1XzocXOKQT2jGfb-

[140] "How long will it take to train an LLM model like GPT-3?" [Online]. Available: https://karvai.medium.com/how-long-will-it-take-to-train-an-llm-model-like-gpt-3-d48407198077

[141] "What is Parallel Computing." [Online]. Available: https://www.ibm.com/think/topics/parallel-computing

[142] S. Gugger, L. Debut, T. Wolf, P. Schmid, Z. Mueller, S. Mangrulkar, M. Sun, and B. Bossan, "Accelerate: Training and inference at scale made simple, efficient and adaptable," https://github.com/huggingface/accelerate, 2022.

[143] "DeepSpeed." [Online]. Available: https://www.deepspeed.ai/

[144] "Zero Redundancy Optimizer (ZeRO)." [Online]. Available: https://deepspeed.readthedocs.io/en/latest/zero3.html

[145] M. Suzgun, N. Scales, N. Schärli, S. Gehrmann, Y. Tay, H. W. Chung, A. Chowdhery, Q. V. Le, E. H. Chi, D. Zhou, and J. Wei, "Challenging BIG-Bench Tasks and Whether Chain-of-Thought Can Solve Them," 2022. [Online]. Available: https://arxiv.org/abs/2210.09261

[146] A. Ansell, I. Vulić, H. Sterz, A. Korhonen, and E. M. Ponti, "Scaling Sparse Fine-Tuning to Large Language Models," 2024.

[147] "Dataset: allenai/tulu-v2-sft-mixture." [Online]. Available: https://huggingface.co/datasets/allenai/tulu-v2-sft-mixture

[148] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned Language Models Are Zero-Shot Learners," 2022. [Online]. Available: https://arxiv.org/abs/2109.01652

[149] "ShareGPT." [Online]. Available: https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered

[150] A. Köpf, Y. Kilcher, D. von Rütte, S. Anagnostidis, Z.-R. Tam, K. Stevens, A. Barhoum, N. M. Duc, O. Stanley, R. Nagyfi, S. ES, S. Suri, D. Glushkov, A. Dantuluri, A. Maguire, C. Schuhmann, H. Nguyen, and A. Mattick, "OpenAssistant Conversations – Democratizing Large Language Model Alignment," 2023. [Online]. Available: https://arxiv.org/abs/2304.07327

[151] S. Chaudhary, "Code Alpaca: An Instruction-following LLaMA model for code generation," https://github.com/sahil280114/codealpaca, 2023.

[152] G. Guiduzzi, "Development of AI benchmarks to monitor the performances of a Supercomputer." Master's thesis, University of Modena and Reggio Emilia, 2021.

[153] "GALILEO100 User Guide." [Online]. Available: https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.3%3A+GALILEO100+UserGuide

[154] "LEONARDO HPC System." [Online]. Available: https://leonardo-supercomputer.cineca.eu/it/leonardo-hpc-system/