ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

Corso di Laurea magistrale in
Ingegneria e Scienze Informatiche

Deep Learning for Natural Language Processing: Novel State-of-the-art Solutions in Summarisation of Legal Case Reports

Tesi in
Text Mining

Relatore:
Chiar.mo Prof.
Gianluca Moro

Presentata da:
Nicola Piscaglia

Anno Accademico 2018-2019

Dedico questo lavoro, il quale chiude uno dei cerchi più importanti della mia vita, a chi la rende unica e mi permette di essere la persona di oggi,

Antonio, Daniela, Linda ed Elisa

Contents

List of Figures

List of Tables

Algorithms

Abstract

Deep neural networks are one of the major classification machines in machine learning. Several Deep Neural Networks (DNNs) have been developed and evaluated in recent years in recognition tasks, such as estimating a user base and estimating their interactivity. We present the best algorithms for extracting or summarising text using a deep neural network while allowing the workers to interpret the texts from the output speech.
In this work both extractive and abstractive summarisation approaches have been applied. In particular, BERT (Base, Multilingual Cased) and a deep neural network composed by CNN and GRU layers have been used in the extraction-based summarisation while the abstraction-based one has been performed by applying the GPT-2 Transformer model. We show our models achieve high scores in syntactical terms while a human evaluation is still needed to judge the coherence, consistency and unreferenced harmonicity of speech. Our proposed work outperform the state of the art results for extractive summarisation on the Australian Legal Case Report Dataset. Our paper can be viewed as further demonstrating that our model can outperform the state of the art on a variety of extractive and abstractive summarisation tasks.

Note: The abstract above was not written by the author, it was generated by providing a part of thesis introduction as input text to the pre-trained GPT-2 (Small) Transformer model [1] described in this work which has been previously fine-tuned for 4 epochs with the "NIPS 2015 Papers" dataset [2].

Chapter 1

Introduction

1.1 Background and goals

Reading and evaluating legal report texts are really time-consuming tasks for judges and lawyers, who usually base their choices on report abstracts, legal principles and common sense reasoning. Unluckily, legal report abstracts are not always available and text summarisation task requires law experts and also long time to be performed. This is the context where the Italian LAILA project starts. Thus, the challenge of this work is to build an automatic summarisation system of legal reports which is able to reduce the amount of text to be analyzed by humans. Two main approaches have been proposed in the literature for the text summarisation task: extractive and abstractive. Extractive summarisation aims to select the most important phrases in a text while the abstractive one is focused on rephrasing an input text like the human summarisation process (hence also using words not present in the input text). Even though NLP research has focused more on the first approach in past years, currently we are witnessing a positive trend for the second method.

1.2 State-of-art in summarisation

Advancements in neural architectures (Cho et al., 2014; Sutskever et al., 2014; Bahdanau et al., 2015; Vinyals et al., 2015; Vaswani et al.,2017), pre-training and transfer learning (Mc-Cann et al., 2017; Peters et al., 2018; Devlin et al., 2018), and availability of large-scale supervised datasets (Sandhaus, 2008; Nallapati et al., 2016; Grusky et al., 2018; Narayan et al., 2018; Sharma et al., 2019) allowed deep learning based approaches to dominate the field. State-of-the-art solutions utilize self-attentive blocks (Liu, 2019; Liu and Lapata, 2019; Zhang et al., 2019), attention and copying mechanisms (See et al., 2017; Cohan et al., 2018), and multi-objective training strategies (Guo et al., 2018; Pasunuru and Bansal, 2018), including reinforcement learning techniques (Kryscinski et al., 2018; Dong et al., 2018; Wu and Hu, 2018). [3]

1.2.1 Extractive models: state-of-art

Extractive summarisation state-of-art includes all BERT-based models which have been evaluated on the non-anonymized version of the dataset introduced by See et al. (2017) [4]. BertSumExt (Liu and Lapata, 2019) model [5] is the best performer (in ROUGE-1, ROUGE-2 and ROUGE-l terms) on that dataset while BERT-ext + RL (Bae et al., 2019) [6], PNBERT (Zhong et al., 2019) [7] and PNBERT (Zhong et al., 2019) [8] are $\approx 0.3 - 1$ points below in all considered metrics.

1.2.2 Abstractive models: state-of-art

Instead, abstractive summarisation state-of-art includes Transformer-based models for sequence-to-sequence learning. The first two models for performance when evaluated on the dataset mentioned above, are PEGASUS (Zhang et al., 2019) [9] and BART (Lewis et al., 2019) [10], respectively.

1.3 Related work

The summarisation task using Australian Legal Case Reports has been first tackled by Galbani et. al, 2010 [11]. In their paper they present a new approach towards legal citation classification using incremental knowledge acquisition, named LEXA. This formed a part of their more ambitious goal of automatic legal text summarization. They created a large training and test corpus from court decision reports in Australia. Then, they showed that, within less than a week, it is possible to develop a good quality knowledge base which considerably outperforms a baseline Machine Learning approach. In their work they also note that the problem of legal citation classification allows the use of Machine Learning as classified training data is available. For other subproblems of legal text summarization this is unlikely to be the case.
In their later work [12] they present the challenges and possibilities of a novel summarisation task: automatic generation of catchphrases for legal documents (catchphrases are meant to present the important legal points of a document with respect of identifying precedents). The authors developed a corpus of legal (human-generated) catchphrases (provided with the submission), which lets them compute statistics useful for automatic catchphrase extraction. They proposed a set of methods to generate legal catchphrases and evaluate them on our corpus. The evaluation shows a recall comparable to humans while still showing a competitive level of precision, which is very encouraging. Finally, they introduced a novel evaluation method for catchphrases for legal texts based on the known ROUGE [13] measure for evaluating summaries of general texts.
Afterwards, the same authors published a new paper [14] which describes their approach

to assign categories and generate catchphrases for legal case reports. They describe their knowledge acquisition framework which lets them quickly build classification rules, using a small number of features, to assign general labels to cases. They show how the resulting knowledge base outperforms machine learning models which use both the designed features or a traditional bag of word representation. The authors also describe how to extend this approach to extract from the full text a list of more specific catchphrases that describe the content of the case.
In the same year, Galbani et al. also published a paper named "Combining different summarization techniques for legal text" [15]where they describe a hybrid approach in which a number of different summarization techniques are combined in a rule-based system using manual knowledge acquisition, where human intuition, supported by data, specifies not only attributes and algorithms, but the contexts where these are best used. The authors applied this approach to automatic summarization of legal case reports, showing how a preliminary knowledge base, composed of only 23 rules, already outperformed competitive baselines.
Last Galbani et al. paper [16] on the Australian Legal Case Reports, presents an approach towards using both incoming and outgoing citation information for document summarisation. Their work aims at generating automatically catchphrases for legal case reports, using, beside the full text, also the text of cited cases and cases that cite the current case. The authors proposed methods to use catchphrases and sentences of cited/citing cases to extract catchphrases from the text of the target case. They created a corpus of cases, catchphrases and citations, and performed a ROUGE based evaluation, which showed the superiority of their citation-based methods over full-text-only methods.
One more recent paper on extractive summarisation of the same legal case reports is "Automatic Catchphrase Identification from Legal Court Case Documents" (Mandal et al., 2017) [17]. The authors proposed an unsupervised approach for extraction and ranking of catchphrases from the our same court case documents, by focusing on noun phrases. They compared the proposed approach with several unsupervised and supervised baselines, showing that the proposed methodology achieves statistically significantly better performance compared to all the baselines.

In the latest published work on Australian Legal Case Reports summarisation, named "Automatic Catchphrase Extraction from Legal Case Documents via Scoring using Deep Neural Networks" (Tran et al., 2018) [18], the authors present a method of automatic catchphrase extracting from legal case documents. They utilized deep neural networks for constructing scoring model of their extraction system. They achieved comparable performance with systems using corpus-wide and citation information which they do not use in their system.

Other recent works on automatic summarisation of legal case reports (which used different datasets) are "Machine Learning Approaches for Catchphrase Extraction in Legal Documents" (Koboyatshwene et al., 2017) [19], "Catchphrase Extraction from Legal Documents Using LSTM" (Bhargava et al., 2017) [20] and "Deep Learning Approach for Extracting Catch Phrases from Legal Documents" (Kayalvizhi et al., 2020) [21].

1.4 Adopted approaches

In this work both approaches have been applied. To achieve this, state-of-art NLP techniques have been applied. In particular, BERT [22] (Base, Multilingual Cased) and a deep neural network composed by CNN and GRU layers have been used in the extraction-based summarisation while the abstraction-based one has been performed by applying the GPT-2 Transformer model. The results will be evaluated by calculating the ROUGE score which is a common evaluation metric for text summarisation. We show our models achieve high scores in syntactical terms considering the difficulty of the tasks while a human evaluation is still needed to judge the coherence, consistency and fluidity of speech.

1.5 Analysis on translated reports

In order to evaluate our models on languages other than English, some translations of the dataset have been performed (via Google Translate API) first and then some in-domain (train and test reports are written in the same language) and cross-domain (test reports are written in a language other than English) experiments have been conducted on the translated reports. This latter approach let us overcome the problem of the absence of labelled data, i.e. in this case Italian legal case reports with an associated summary. Thus, the most interesting cross-domain scenario has English reports as source domain and reports translated into Italian as target domain. The relative results are encouraging and show the capability of our models to reach a certain level of language understanding.

1.6 Research contribution

The proposed work outperform the state of the art results for the extractive summarisation task on the Australian Legal Case Report Dataset [18].
Our abstractive summarisation experiments represent the first GPT-2 Transformer application to the summarisation task of legal case reports.
In addition, we set a performance baseline for the abstractive summarisation of the Australian Legal Case Report dataset.

Another contribution is related to the evaluation of our model effectiveness in the summarisation of legal case reports written in a language other than English: they have been tested both extractive and abstractive summarisation performance on translated legal case reports.
It has been also tested the cross-language performance of our Deep Learning models both on extractive and abstractive summarisation tasks where the source domain is represented by the Australian legal case reports (written in English) and the target domain is represented by the legal case reports which have been translated into Italian.

Chapter 2

Language representation models

2.1 Introduction

A statistical language model is a probability distribution over sequences of words. Given such a sequence, say of length m, it assigns a probability $P(w_1, \ldots, w_m))$ to the whole sequence.[23]

The language model provides context to distinguish between words and phrases that sound similar. For example, in American English, the phrases "recognize speech" and "wreck a nice beach" sound similar, but mean different things.

Data sparsity is a major problem in building language models. Most possible word sequences are not observed in training. One solution is to make the assumption that the probability of a word only depends on the previous n words. This is known as an n-gram model or unigram model when $n = 1$. The unigram model is also known as the bag of words model.

Estimating the relative likelihood of different phrases is useful in many natural language processing applications, especially those that generate text as an output. Language modeling is used in speech recognition, machine translation, part-of-speech tagging, parsing, Optical Character Recognition, handwriting recognition, information retrieval and other applications.

In speech recognition, sounds are matched with word sequences. Ambiguities are easier to resolve when evidence from the language model is integrated with a pronunciation model and an acoustic model.

Language models are used in information retrieval in the query likelihood model. There a separate language model is associated with each document in a collection. Documents are ranked based on the probability of the query Q in the document's language model M_d : $P(Q \mid M_d))$. Commonly, the unigram language model is used for this purpose.

2.1.1 Model Types

2.1.1.1 Unigram

A unigram model can be treated as the combination of several one-state finite automata.[1] It splits the probabilities of different terms in a context, e.g. from

$$P(t_1t_2t_3) = P(t_1)P(t_2 \mid t_1)P(t_3 \mid t_1t_2)P(t_1t_2t_3) = P(t_1)P(t_2 \mid t_1)P(t_3 \mid t_1t_2) \quad (2.1)$$

to

$$P_{\text{uni}}(t_1t_2t_3) = P(t_1)P(t_2)P(t_3)P_{\text{uni}}(t_1t_2t_3) = P(t_1)P(t_2)P(t_3) \quad (2.2)$$

In this model, the probability of each word only depends on that word's own probability in the document, so we only have one-state finite automata as units. The automaton itself has a probability distribution over the entire vocabulary of the model, summing to 1:

$$\sum_{\text{term in doc}} P(\text{term}) = 1 \quad (2.3)$$

The probability generated for a specific query is calculated as:

$$P(\text{query}) = \prod_{\text{term in query}} P(\text{term}) \quad (2.4)$$

Different documents have unigram models, with different hit probabilities of words in it. The probability distributions from different documents are used to generate hit probabilities for each query. Documents can be ranked for a query according to the probabilities.
In information retrieval contexts, unigram language models are often smoothed to avoid instances where $P(term) = 0$. A common approach is to generate a maximum-likelihood model for the entire collection and linearly interpolate the collection model with a maximum-likelihood model for each document to smooth the model.

2.1.1.2 n-gram

In an n-gram model, the probability $P(w_1, \ldots, w_m)P(w_1, \ldots, w_m)$ of observing the sentence $w_1, \ldots, w_mw_1, \ldots, w_m$ is approximated as

$$P(w_1, \ldots, w_m) = \prod_{i=1}^{m} P(w_i \mid w_1, \ldots, w_{i-1}) \approx \prod_{i=1}^{m} P(w_i \mid w_{i-(n-1)}, \ldots, w_{i-1}) \quad (2.5)$$

It is assumed that the probability of observing the i-th word w_i in the context history of the preceding i $-$ 1 words can be approximated by the probability of observing it in the shortened context history of the preceding n $-$ 1 words (nth order Markov property).
The conditional probability can be calculated from n-gram model frequency counts:

$$P(w_i \mid w_{i-(n-1)}, \ldots, w_{i-1}) = \frac{\text{count}(w_{i-(n-1)}, \ldots, w_{i-1}, w_i)}{\text{count}(w_{i-(n-1)}, \ldots, w_{i-1})} \quad (2.6)$$

The terms bigram and trigram language models denote n-gram models with $n = 2$ and $n = 3$, respectively.
Typically, the n-gram model probabilities are not derived directly from frequency counts, because models derived this way have severe problems when confronted with any n-grams that have not been explicitly seen before. Instead, some form of smoothing is necessary, assigning some of the total probability mass to unseen words or n-grams. Various methods are used, from simple "add-one" smoothing (assign a count of 1 to unseen n-grams, as an uninformative prior) to more sophisticated models, such as Good-Turing discounting [24] [25] or back-off models [26] [27].

2.1.1.3 Exponential

Maximum entropy language models encode the relationship between a word and the n-gram history using feature functions. The equation is

$$P(w_m|w_1,\ldots,w_{m-1}) = \frac{1}{Z(w_1,\ldots,w_{m-1})}\exp(a^T f(w_1,\ldots,w_m)) \tag{2.7}$$

where $Z(w_1,\ldots,w_{m-1})$ is the partition function, a is the parameter vector, and $f(w_1,\ldots,w_m)$ is the feature function. In the simplest case, the feature function is just an indicator of the presence of a certain n-gram. It is helpful to use a prior on a or some form of regularization.
The log-bilinear model is another example of an exponential language model.

2.1.1.4 Neural network

Neural language models (or continuous space language models) use continuous representations or embeddings of words to make their predictions. These models make use of Neural networks.
Continuous space embeddings help to alleviate the curse of dimensionality in language modeling: as language models are trained on larger and larger texts, the number of unique words (the vocabulary) increases. The number of possible sequences of words increases exponentially with the size of the vocabulary, causing a data sparsity problem because of the exponentially many sequences. Thus, statistics are needed to properly estimate probabilities. Neural networks avoid this problem by representing words in a distributed way, as non-linear combinations of weights in a neural net.
Typically, neural net language models are constructed and trained as probabilistic classifiers that learn to predict a probability distribution

$$P(w_t|\text{context})\,\forall t \in V \tag{2.8}$$

I.e., the network is trained to predict a probability distribution over the vocabulary, given some linguistic context. This is done using standard neural net training algorithms such as

stochastic gradient descent with backpropagation. The context might be a fixed-size window of previous words, so that the network predicts

$$P(w_t|w_{t-k},\ldots,w_{t-1}) \tag{2.9}$$

from a feature vector representing the previous k words.[6] Another option is to use "future" words as well as "past" words as features, so that the estimated probability is

$$P(w_t|w_{t-k},\ldots,w_{t-1},w_{t+1},\ldots,w_{t+k}) \tag{2.10}$$

A third option that allows faster training is to invert the previous problem and make a neural network learn the context, given a word. One then maximizes the log-probability

$$\sum_{-k\leq j-1,\, j\leq k} \log P(w_{t+j}|w_t)) \tag{2.11}$$

This is called a skip-gram language model, and is the basis of the Word2Vec program [28]. Instead of using neural net language models to produce actual probabilities, it is common to instead use the distributed representation encoded in the networks' "hidden" layers as representations of words; each word is then mapped onto an n-dimensional real vector called the word embedding, where n is the size of the layer just before the output layer. The representations in skip-gram models have the distinct characteristic that they model semantic relations between words as linear combinations, capturing a form of compositionality. For example, in some such models, if v is the function that maps a word w to its n-d vector representation, then

$$v(\text{king}) - v(\text{male}) + v(\text{female}) \approx v(\text{queen}) \tag{2.12}$$

where $\approx$ is made precise by stipulating that its right-hand side must be the nearest neighbor of the value of the left-hand side.

2.1.2 Word Embedding

Word Embedding is the collective name for a set of language modeling and feature learning techniques in natural language processing (NLP). Words or phrases from the vocabulary are mapped to vectors of real numbers. These techniques project a vector of numbers from a space with a size per word into a continuous vector space with far fewer dimensions.

2.1.2.1 Why generate Word Embeddings?

There are two main motivations to generate embeddings of words or sentences:

- when you analyze text using Data Mining techniques you need an input numeric representation;

- when used as the underlying input representation, they have been shown to boost the performance in NLP tasks.

2.1.2.2 How to generate Word Embeddings?

Methods to generate this mapping include neural networks, dimensionality reduction on the word co-occurrence matrix, probabilistic models, etc. In the subsequent sections we will go through the main embedding techniques proposed so far in the literature.

2.2 Word Embedding Techniques

2.2.1 Word2Vec (Mikolov et al., 2013)

Word2vec[28] is a group of related models that are used to produce word embeddings [29]. These models are shallow, two-layer neural networks that are trained to reconstruct linguistic contexts of words. Word2vec takes as its input a large corpus of text and produces a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space. Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located close to one another in the space.
Word2vec was created and published in 2013 by a team of researchers led by Tomas Mikolov at Google and patented. The algorithm has been subsequently analysed and explained by other researchers. Embedding vectors created using the Word2vec algorithm have many advantages compared to earlier algorithms such as latent semantic analysis.
As you can see in Figure 2.1, Word2vec can utilize either of two model architectures to produce a distributed representation of words: continuous bag-of-words (CBOW) or continuous skip-gram. In the continuous bag-of-words architecture, the model predicts the current word from a window of surrounding context words. The order of context words does not influence prediction (bag-of-words assumption). In the continuous skip-gram architecture, the model uses the current word to predict the surrounding window of context words. The skip-gram architecture weighs nearby context words more heavily than more distant context words. According to the authors' note, CBOW is faster while skip-gram is slower but does a better job for infrequent words.
To summarise, main Word2Vec features include:

- it uses two-layer neural networks that are trained to reconstruct linguistic contexts of words;

- takes as its input a large corpus of text and produces a vector space (typically of several hundred dimensions);

- each unique word in the corpus is assigned a corresponding vector in the space;
- word vectors are positioned in the vector space such that words that share common contexts in the corpus are located close to one another in the space.

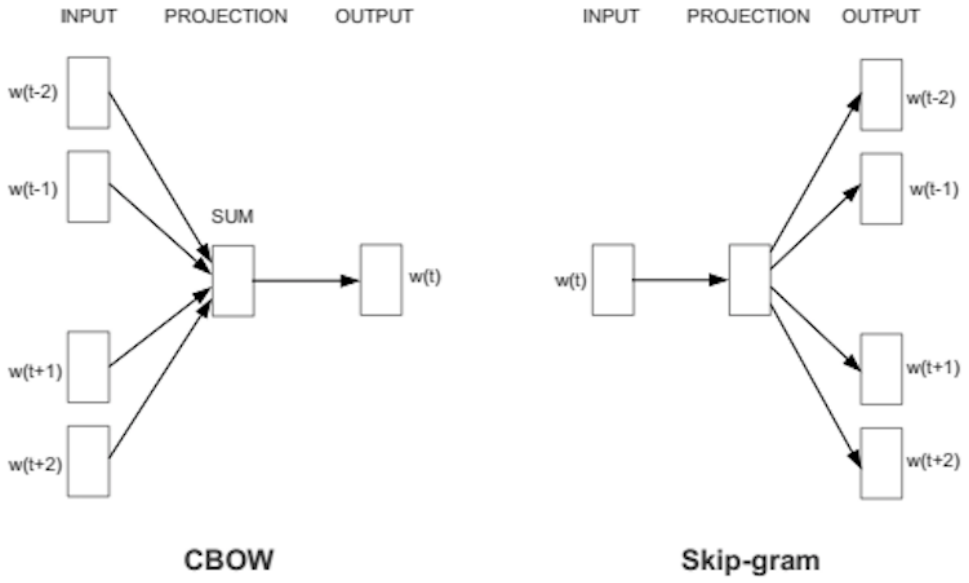


Figure 2.1: Word2Vec Training Models: CBOW and Skip-gram

The main Word2Vec strength is the **preservation of semantic and syntactic relationships**: semantic and syntactic patterns can be reproduced using vector arithmetic. This means that patterns such as “Man is to Woman as Brother is to Sister” can be generated through algebric operations on the vector representations of these words. The vector representation of “Brother” - ”Man” + ”Woman” produces a result which is closest to the vector representation of “Sister” in the model.
Main Word2Vec weaknesses include:

- it needs a large corpora and a high number of dimensions to achieve better performance than other algorithms such as LSA [30];
- Word Embeddings training can be computationally expensive especially if Skip-gram model is used;
- it does not have any explicit global information embedded in it by default. The training could not be able to learn global patterns on the entire corpora.

Below are some results details of Word2Vec model from the original paper[28]:

- a Google News corpus has been used for training the word vectors (this corpus contains about 6B tokens);

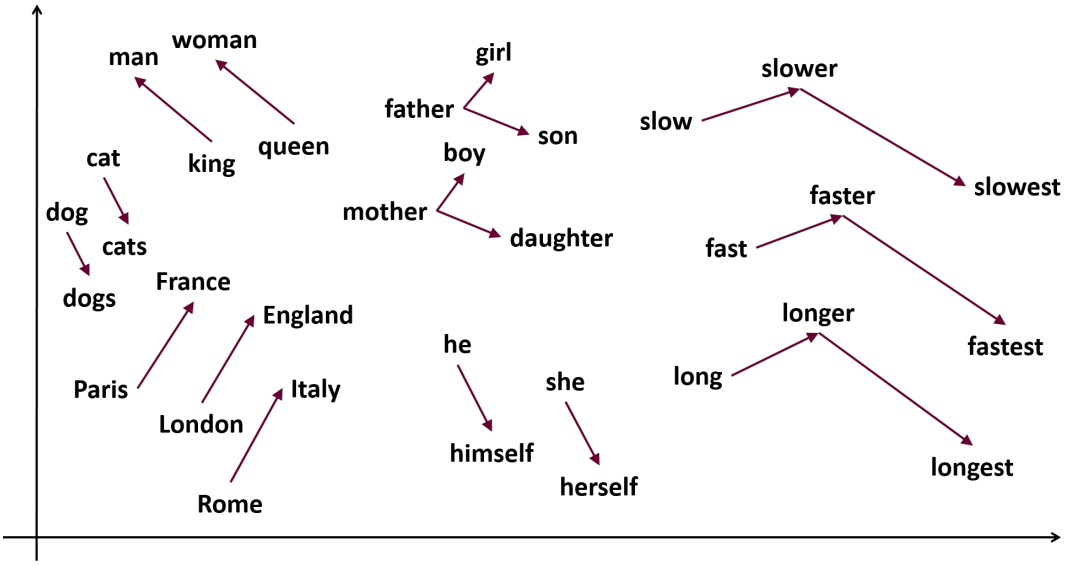


Figure 2.2: Word2Vec vector examples in a bidimensional space

- the vocabulary size has been restricted to 1 million most frequent words;
- comparison of architectures using models trained on the same data, with 640-dimensional word vectors:

Model Architecture	Semantic-Syntactic Word Relationship test set		MSR Word Relatedness Test Set [20]
	Semantic Accuracy [%]	Syntactic Accuracy [%]	
RNNLM	9	36	35
NNLM	23	53	47
CBOW	24	64	61
Skip-gram	55	59	56

Figure 2.3: Word2Vec results from the original paper on Semantic-Syntactic Word Relationship task. The model has been trained on a google news corpus of about 6B tokens and a vocabulary containing 1 million most frequent words.

2.2.2 GloVe (Pennington et al., 2014)

GloVe [31], coined from Global Vectors, is a model for distributed word representation. The model is an unsupervised learning algorithm for obtaining vector representations for words. [32] This is achieved by mapping words into a meaningful space where the distance between words is related to semantic similarity.

Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the

word vector space. It is developed as an open-source project at Stanford.
As log-bilinear regression model for unsupervised learning of word representations, it combines the features of two model families, namely the global matrix factorization and local context window methods.
GloVe main features include:

- it is a **count-based model** that learn their vectors by essentially doing dimensionality reduction on the global word-word co-occurrence counts matrix:

  1. it first constructs a large matrix of (words $\times$ context) co-occurrence information, i.e. for each "word" (the rows), you count how frequently we see this word in some "context" (the columns) in a large corpus;
  2. so then it factorizes this matrix to yield a lower-dimensional (word x features) matrix, where each row now yields a vector representation for each word.

- the resulting representations showcase interesting linear substructures of the word vector space as you can see in Figure 2.4.

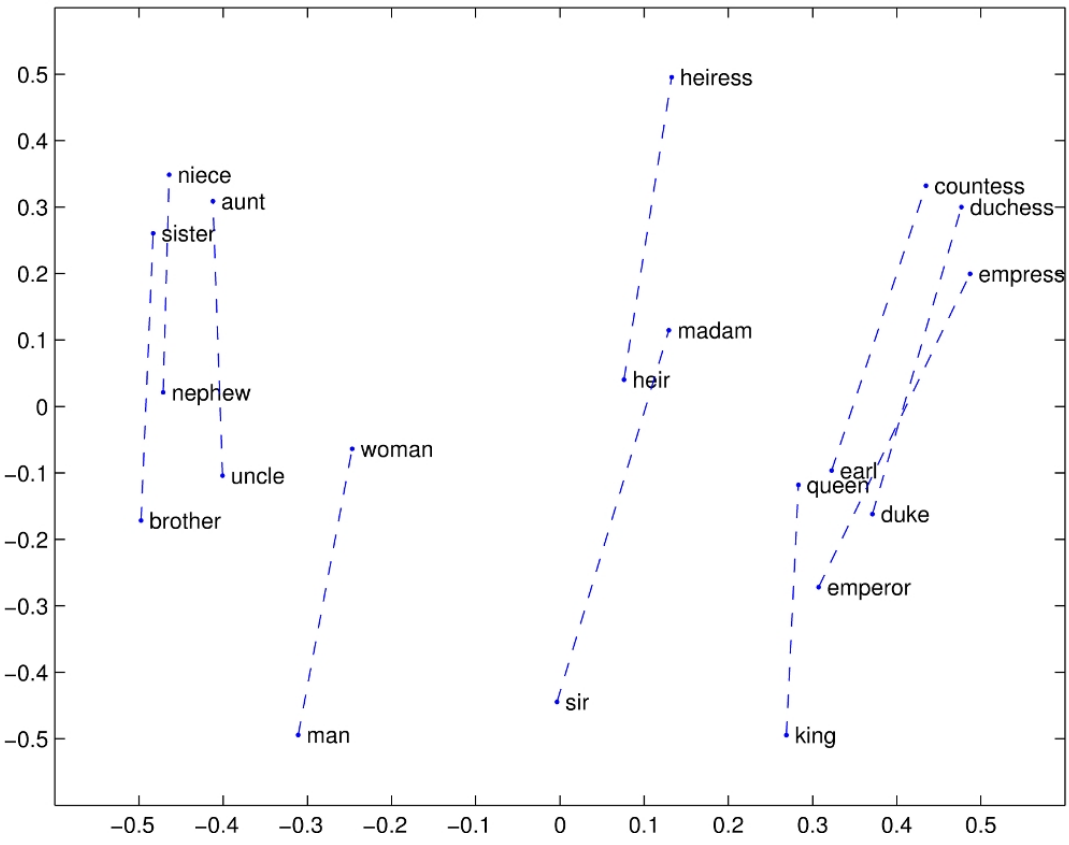


Figure 2.4: GloVe vector examples in a bidimensional space

GloVe main strengths include:

- the Euclidean distance (or cosine similarity) between two word vectors provides an effective method for measuring the linguistic or semantic similarity of the corresponding words;
- it is easier to parallelize the implementation than Word2Vec model, which means it is easier to train over more data;
- it achieves better performance than Word2Vec on some tasks and datasets (e.g. Sentiment Analysis on Amazon reviews).

GloVe main weaknesses include:

- populating the co-occurence matrix requires a single pass through the entire corpus to collect the statistics. For large corpora, this pass can be computationally expensive (but it is a one-time up-front cost);
- the model is trained on the co-occurrence matrix of words, which takes a lot of memory for storage;
- especially, if you change the hyper-parameters related to the co-occurrence matrix, you have to reconstruct the matrix again, which is very time-consuming as mentioned before.

Below are some GloVe results from the original paper [31]:

- Accuracy on the analogy task for 300-dimensional vectors trained on different corpora are shown in Figure 2.5;
- in Figure 2.6 we can compare GloVe and Word2vec performances. Overall accuracy on the word analogy task is plotted as a function of training time:
  - governed by the number of iterations for GloVe and by the number of negative samples for CBOW (a) and skip-gram (b);
  - in all cases, 300-dimensional vectors on the same 6B token corpus (Wikipedia 2014 + Gigaword 5) have been trained with the same 400,000 word vocabulary, and a symmetric context window of size 10.

2.2.3 Word2Vec & Glove limitations

Both Word2vec and Glove do not solve the problems like:

- how to learn the representation for out-of-vocabulary words;

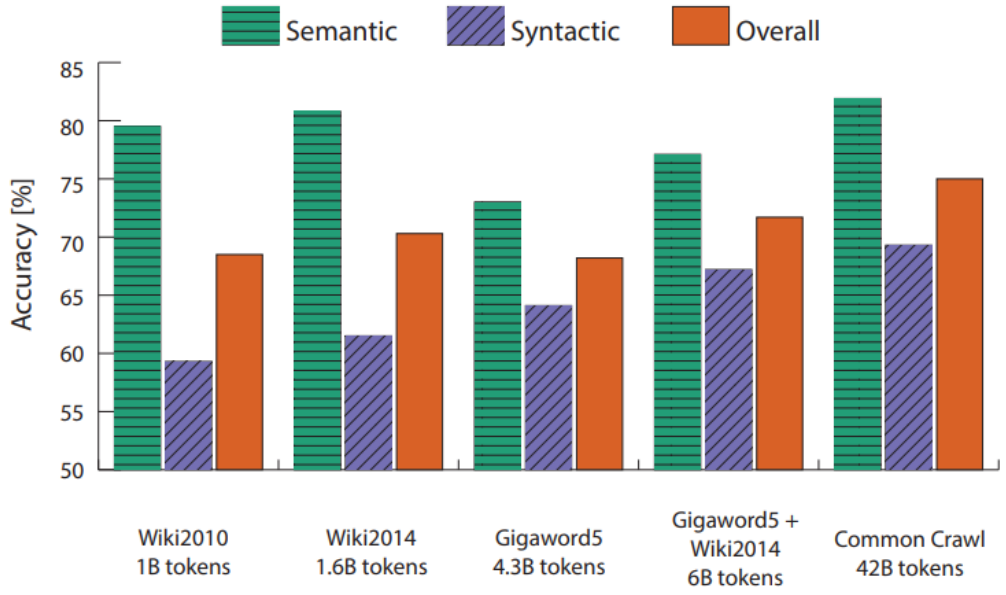


Figure 2.5: GloVe results on analogy task

- how to separate some opposite word pairs. For example, *"good"* and *"bad"* are usually located very close to each other in the vector space, which may limit the performance of word vectors in NLP tasks like sentiment analysis;

- possible meanings of a word are conflated into a single representation (a single vector in the semantic space);

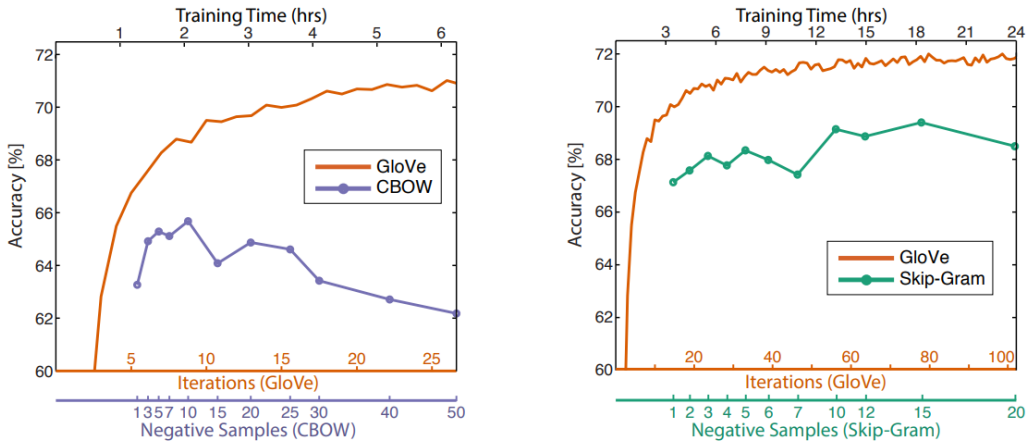
- they do not take into account word order in their training.

2.2.4 BERT (Devlin et al. 2018)

BERT [22] is a method of pre-training language representations, meaning that we train a general-purpose "language understanding" model on a large text corpus (like Wikipedia), and then use that model for downstream NLP tasks that we care about (like question answering). BERT outperforms previous methods because it is the first unsupervised, deeply bidirectional system for pre-training NLP.

Unsupervised means that BERT was trained using only a plain text corpus, which is important because an enormous amount of plain text data is publicly available on the web in many languages.

Pre-trained representations can also either be context-free or contextual, and contextual representations can further be unidirectional or bidirectional. Context-free models such as word2vec or GloVe generate a single "word embedding" representation for each word in the vocabulary, so bank would have the same representation in bank deposit and river bank.



(a) GloVe vs CBOW

(b) GloVe vs Skip-Gram

Figure 2.6: GloVe vs. Word2Vec

Contextual models instead generate a representation of each word that is based on the other words in the sentence.

BERT was built upon recent work in pre-training contextual representations — including Semi-supervised Sequence Learning, Generative Pre-Training, ELMo, and ULMFit — but crucially these models are all unidirectional or shallowly bidirectional. This means that each word is only contextualized using the words to its left (or right). For example, in the sentence I made a bank deposit the unidirectional representation of bank is only based on I made a but not deposit. Some previous work does combine the representations from separate left-context and right-context models, but only in a "shallow" manner. BERT represents "bank" using both its left and right context — I made a ... deposit — starting from the very bottom of a deep neural network, so it is deeply bidirectional.

BERT uses a simple approach for this:

1. mask out 15% of the words in the input;
2. run the entire sequence through a deep bidirectional Transformer encoder [33];
3. predict only the masked words (Figure 2.7).

For example:

```
Input: the man went to the [MASK1].
       He bought a [MASK2] of milk.
Labels: [MASK1] = store; [MASK2] = gallon
```

In order to learn relationships between sentences, we also train on a simple task which can be generated from any monolingual corpus: Given two sentences A and B, is B the actual next sentence that comes after A, or just a random sentence from the corpus?

```
Sentence A: the man went to the store .
```

```
Sentence B: he bought a gallon of milk .
Label: IsNextSentence

Sentence A: the man went to the store .
Sentence B: penguins are flightless .
Label: NotNextSentence
```

Then a large model (12-layer to 24-layer Transformer) is trained on a large corpus (Wikipedia + BookCorpus) for a long time (1M update steps), and that is BERT.
Using BERT has two stages: pre-training and fine-tuning.
Pre-training is fairly expensive (four days on 4 to 16 Cloud TPUs), but is a one-time procedure for each language (current models are English-only, but multilingual models will be released in the near future). We are releasing a number of pre-trained models from the paper which were pre-trained at Google. Most NLP researchers will never need to pre-train their own model from scratch.
Fine-tuning is inexpensive. All of the results in the paper can be replicated in at most 1 hour on a single Cloud TPU, or a few hours on a GPU, starting from the exact same pre-trained model. SQuAD, for example, can be trained in around 30 minutes on a single Cloud TPU to achieve a Dev F1 score of 91.0%, which is the single system state-of-the-art.
The other important aspect of BERT is that it can be adapted to many types of NLP tasks very easily. In the paper, we demonstrate state-of-the-art results on sentence-level (e.g., SST-2), sentence-pair-level (e.g., MultiNLI), word-level (e.g., NER), and span-level (e.g., SQuAD) tasks with almost no task-specific modifications.
To sum up, BERT:

- is a new method of pre-training language representations, meaning that we train a general-purpose "language understanding" model on a large text corpus (e.g. Wikipedia), and then use that model for downstream NLP tasks that we care about (e.g. question answering).
- outperforms previous methods because it is the first unsupervised, deeply bidirectional system for pre-training NLP.

Pre-trained representations can also either be **context-free** or **contextual**, and contextual representations can further be **unidirectional** or **bidirectional**.
Context-free models such as Word2Vec or GloVe generate a single "word embedding" representation for each word in the vocabulary, so bank would have the same representation in *"bank deposit"* and *"river bank"* (Figure 2.8). Contextual models instead generate a representation of each word that is based on the other words in the sentence (Figure 2.9).
Previous work in pre-training contextual representations (e.g. ELMo [34]) produced only unidirectional or shallowly bidirectional models: this means that each word is only

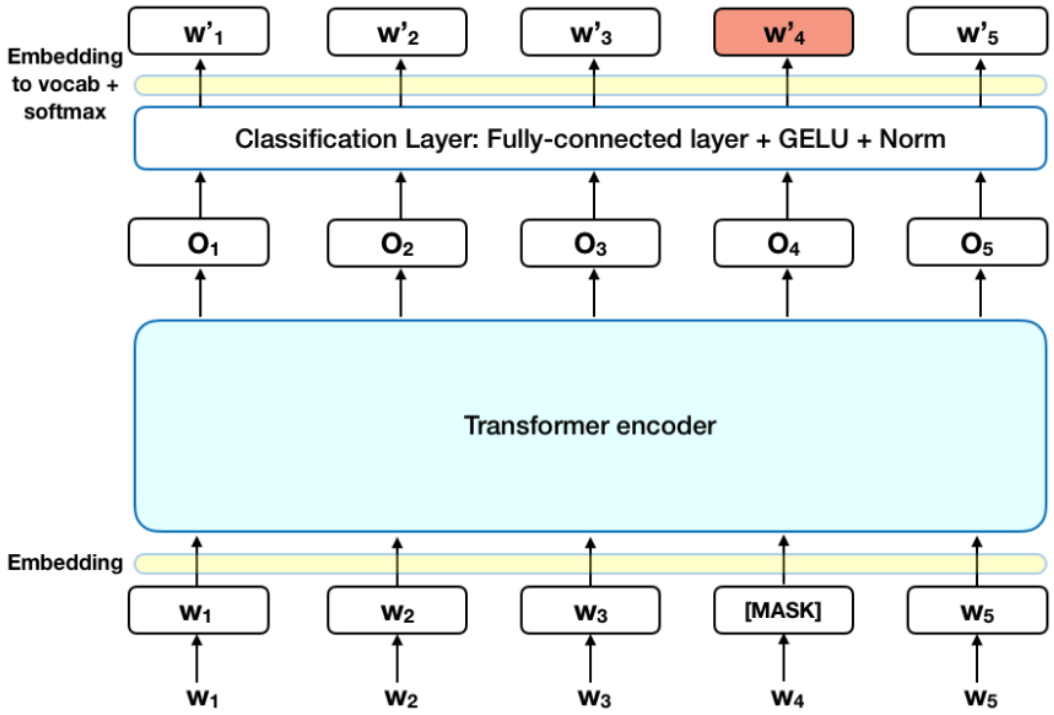


Figure 2.7: BERT - Training Neural Network architecture

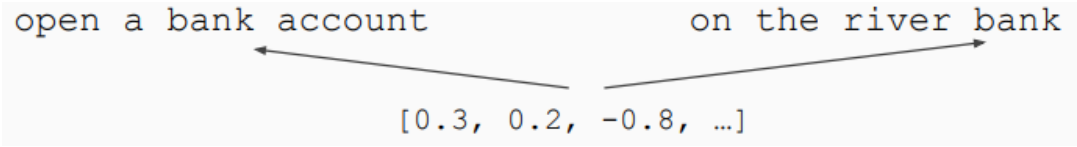


Figure 2.8: Examples of context-free Word Embeddings

contextualized using the words to its left (or right). For example, in the sentence *"I made a bank deposit"* the unidirectional representation of bank is only based on "I made a" but not "deposit". BERT represents "bank" using both its left and right context — *I made a ... deposit* — starting from the very bottom of a deep neural network, so it is deeply bidirectional.

What is the difference between BERT and other context produced by recent bidirectional methods like ELMo?

- Bidirectional LSTM based language models train a standard left-to-right language model and also train a right-to-left (reverse) language model that predicts previous words from subsequent words.
- The crucial difference is this: **neither LSTM takes both the previous and subsequent tokens into account at the same time**.
- In BERT The model is forced to use information from the entire sentence simultaneously – regardless of the position - to make good predictions

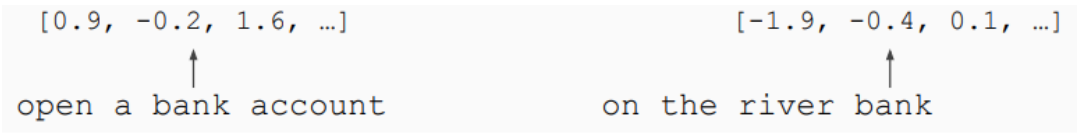


Figure 2.9: Examples of context-based Word Embeddings

Main BERT strengths include:

- it produces a contextual representation;
- based on a Bidirectional model;
- many pre-trained models are available for different languages. These ones can be latter fine tuned being tailored to a new specific task;
- it can handle OOV (Out Of Vocabulary) cases by learning subwords representations;
- it is unsupervised that means BERT was trained using only a plain text corpus, which is important because an enormous amount of plain text data is publicly available on the web in many languages.

BERT main weaknesses include:

- BERT is pre-trained on two unsupervised tasks: sentence reconstruction and next sentence prediction.
  The reconstruction task involves randomly masking tokens in a sentence, and reconstructing the original sentence from the masked one.
  The model reconstructs the masked tokens conditionally independently of one another. However, this is not really a valid assumption.
  An example from the paper uses the sentence, *"New York is a city"*, where *"New"* and *"York"* are masked and reconstructed during training. Clearly, if the first word is *"New"* then the next word is more likely to be *"York"*.
- BERT also uses a special [mask] token during pre-training, which creates some discrepancy between pre-train and fine-tune stages, where a special token such as [mask] is not used for the latter.
- BERT has a limitation of handling sentences of a maximum length (512). Smaller length sentences are padded.
  We can choose the maximum length in BERT but we are limited by the available memory on GPU (choice of sentence length impacts size of matrix to be loaded in GPU). This is not an issue if we are not using GPU but then the processing time is higher in CPU.

Below are some results from the original paper of BERT:

- The Stanford Question Answering Dataset (SQuAD v1.1) is a collection of 100k crowdsourced question/answer pairs (Rajpurkar et al., 2016) [35]. Given a question and a passage from Wikipedia containing the answer, the task is to predict the answer text span in the passage. In Figure 2.10 SQuAD v1.1 results of BERT are shown:

System	Dev		Test	
	EM	F1	EM	F1
Top Leaderboard Systems (Dec 10th, 2018)				
Human	-	-	82.3	91.2
#1 Ensemble - nlnet	-	-	86.0	91.7
#2 Ensemble - QANet	-	-	84.5	90.5
Published				
BiDAF+ELMo (Single)	-	85.6	-	85.8
R.M. Reader (Ensemble)	81.2	87.9	82.3	88.5
Ours				
$\text{BERT}_{\text{BASE}}$ (Single)	80.8	88.5	-	-
$\text{BERT}_{\text{LARGE}}$ (Single)	84.1	90.9	-	-
$\text{BERT}_{\text{LARGE}}$ (Ensemble)	85.8	91.8	-	-
$\text{BERT}_{\text{LARGE}}$ (Sgl.+TriviaQA)	**84.2**	**91.1**	**85.1**	**91.8**
$\text{BERT}_{\text{LARGE}}$ (Ens.+TriviaQA)	**86.2**	**92.2**	**87.4**	**93.2**

Figure 2.10: BERT - Results on SQuAD dataset

- The General Language Understanding Evaluation (GLUE) benchmark (Wang et al., 2018) [36] is a collection of diverse natural language understanding tasks. In Figure 2.11 GLUE test results of BERT are shown:
  - the number below each task denotes the number of training examples;
  - F1 scores are reported for QQP and MRPC;
  - Spearman correlations are reported for STS-B;
  - accuracy scores are reported for the other tasks.

System	MNLI-(m/mm) 392k	QQP 363k	QNLI 108k	SST-2 67k	CoLA 8.5k	STS-B 5.7k	MRPC 3.5k	RTE 2.5k	**Average** -
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
$\text{BERT}_{\text{BASE}}$	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
$\text{BERT}_{\text{LARGE}}$	**86.7/85.9**	**72.1**	**92.7**	**94.9**	**60.5**	**86.5**	**89.3**	**70.1**	**82.1**

Figure 2.11: BERT - Results on GLUE tasks

Below two recent BERT applications are briefly presented:

- Allen Institute for Artificial Intelligence (AI2) further study on BERT and released SciBERT which is based on BERT to address the performance on scientific data.
  - It uses a pre-trained model from BERT and fine-tune contextualized embeddings by using scientific publications which including 18% papers from computer science domain and 82% from the broad biomedical domain.
- On the other hand, Lee et al. work on biomedical domain. They also noticed that generic pretrained NLP model may not work very well in specific domain data.
  - Therefore, they fine-tuned BERT to be BioBERT and $0.51\% \sim 9.61\%$ absolute improvement in biomedical's NER, relation extraction and question answering NLP tasks.

All details can be found in these papers:

- SCIBERT: Pretrained Contextualized Embeddings for Scientific Text (Beltagy et al., 2019) [37]
- BioBERT: a pre-trained biomedical language representation model for biomedical text mining (Lee et al., 2019) [38]

2.2.5 ELMo (Peters et al., 2018)

ELMo [34] is a deep contextualized word representation that models both complex characteristics of word use (e.g., syntax and semantics), and how these uses vary across linguistic contexts (i.e., to model polysemy).
These word vectors are learned functions of the internal states of a deep bidirectional language model (biLM), which is pre-trained on a large text corpus and based on LSTM networks.
They can be easily added to existing models and significantly improve the state of the art across a broad range of challenging NLP problems, including question answering, textual entailment and sentiment analysis.
ELMo representations are:

- **contextual**: the representation for each word depends on the entire context in which it is used; Word embedding process is shown in Figure 2.13;
- **deep**: the word representations combine all layers of a deep pre-trained neural network;
- **character based**: ELMo representations are purely character based, allowing the network to use morphological clues to form robust representations for

out-of-vocabulary tokens unseen in training. Character embedding creation can be seen in Figure 2.12.

Main ELMo strengths are:

- it produces different representations of a word depending on the context it appears in;
- it can handle Out Of Vocabulary (OOV) words because its representations are character based;
- it is based on deep bidirectional LSTM networks which allow to learn more context-dependent aspects of word meanings in the higher layers along with syntax aspects in lower layers.

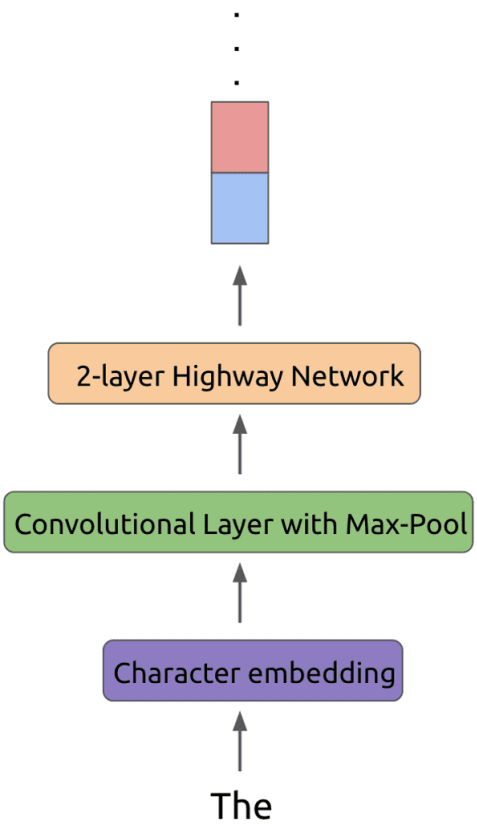


Figure 2.12: ELMo - Character Embedding creation

Main ELMo weaknesses are:

- it is a shallower bidirectional model than BERT;
- the representation cannot take advantage of both left and right contexts simultaneously.

In Figure 2.14 you can see a test set comparison of ELMo enhanced neural models with state-of-the-art single model baselines across six benchmark NLP tasks.

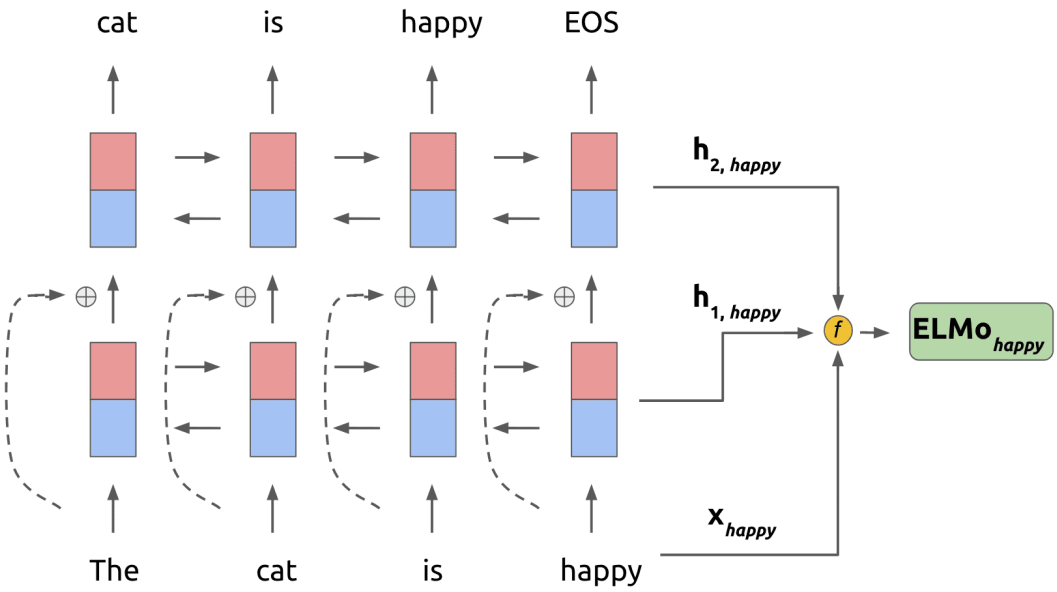


Figure 2.13: ELMo - Word Embedding creation process

TASK	PREVIOUS SOTA		OUR BASELINE	ELMO + BASELINE	INCREASE (ABSOLUTE/ RELATIVE)
SQuAD	Liu et al. (2017)	84.4	81.1	85.8	4.7 / 24.9%
SNLI	Chen et al. (2017)	88.6	88.0	88.7 ± 0.17	0.7 / 5.8%
SRL	He et al. (2017)	81.7	81.4	84.6	3.2 / 17.2%
Coref	Lee et al. (2017)	67.2	67.2	70.4	3.2 / 9.8%
NER	Peters et al. (2017)	91.93 ± 0.19	90.15	92.22 ± 0.10	2.06 / 21%
SST-5	McCann et al. (2017)	53.7	51.4	54.7 ± 0.5	3.3 / 6.8%

Figure 2.14: ELMo - Results on several tasks

2.2.6 Bidirectional NN based Language Models motivation

Here we want to make a brief reflection on the difference between the use of the context in new language models and the Word2Vec to give a useful intuition.
In Word2Vec both left and right context is rebuilt to produce word embeddings. Thus, what is the difference with recent contextual models based on bidirectional networks like ELMo? In models like ELMo information flows forward and backward thanks to bidirectional nets so the neural network sees data twice.

2.2.6.1 Why making the neural network read backwards?

In a sentence statistically the significance of the first words is very important but the net sees them first and then forgets them because there are other successive words that it must process. Scrolling in two directions rebalances the forgetfulness of the first words over time.

2.2.7 Word Embeddings Techniques Differences

Main differences among word embeddings are related to learnt representation type and contextualized feature. Based on these two characteristics, we can classify the embedding techniques described so far as in Figure 2.15.

Model Name	Context Sensitive embeddings	Learnt representations
Word2vec	No	Words
Glove	No	Words
ELMo	Yes	Words
BERT	Yes	Subwords

Figure 2.15: Differences between GloVe, BERT, Word2Vec and ELMo

2.2.8 Flair (Akbik et al., 2018)

Flair [39] is a novel type of contextualized character-level word embedding. These latter embeddings are pre-trained on large unlabeled corpora and they are able to capture word meaning in context. Therefore, this approach produces different embeddings for polysemous words depending on their usage.
Flair method models words and context fundamentally as sequences of characters, to both better handle rare and misspelled words as well as model subword structures such as prefixes and endings. Thus, the embedding process:

- passes sentences as sequences of characters into a character-level language model to form word-level embeddings.
- uses the LSTM variant of recurrent neural networks as language modeling architecture

The embedding creation process can be visualised in Figure 2.16 and it is performed as follows:

1. from the forward language model (shown in red), we extract the output hidden state after the last character in the word. This hidden state thus contains information propagated from the beginning of the sentence up to this point;
2. from the backward language model (shown in blue), we extract the output hidden state before the first character in the word. It thus contains information propagated from the end of the sentence to this point;

3. both output hidden states are concatenated to form the final embedding.

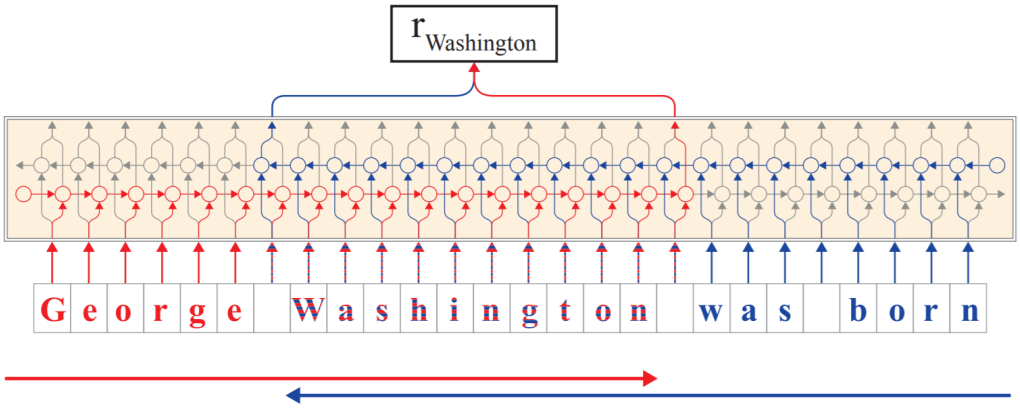


Figure 2.16: Flair Context Embedding generation

As one can see in Figure 2.17, Flair approach:

- significantly outperforms all previous works on Named Entity Recognition (NER)
- slightly outperform the previous state-of-the-art in Part-of-Speech (PoS) tagging and chunking

Approach	NER-English F1-score	NER-German F1-score	Chunking F1-score	POS Accuracy
proposed				
PROPOSED	91.97±0.04	85.78 ± 0.18	96.68±0.03	97.73±0.02
PROPOSED$_{+WORD}$	93.07±0.10	88.20 ± 0.21	96.70±0.04	97.82±0.02
PROPOSED$_{+CHAR}$	91.92±0.03	85.88 ± 0.20	**96.72**±0.05	97.8±0.01
PROPOSED$_{+WORD+CHAR}$	**93.09**±0.12	**88.32** ± 0.20	96.71±0.07	97.76±0.01
PROPOSED$_{+ALL}$	92.72±0.09	n/a	96.65±0.05	**97.85**±0.01
baselines				
HUANG	88.54±0.08	82.32 ± 0.35	95.4±0.08	96.94±0.02
LAMPLE	89.3±0.23	83.78 ± 0.39	95.34±0.06	97.02±0.03
PETERS	92.34±0.09	n/a	96.69±0.05	97.81± 0.02
best published				
	92.22±0.10	78.76	96.37±0.05	97.64
	(Peters et al., 2018)	(Lample et al., 2016)	(Peters et al., 2017)	(Choi, 2016)
	91.93±0.19	77.20	95.96±0.08	97.55
	(Peters et al., 2017)	(Seyler et al., 2017)	(Liu et al., 2017)	(Ma and Hovy, 2016)
	91.71±0.10	76.22	95.77	97.53±0.03
	(Liu et al., 2017)	(Gillick et al., 2015)	(Hashimoto et al., 2016)	(Liu et al., 2017)
	91.21	75.72	95.56	97.30
	(Ma and Hovy, 2016)	(Qi et al., 2009)	Søgaard et al. (2016)	(Lample et al., 2016)

Figure 2.17: Flair - Results on several tasks

2.2.9 ALBERT (Lan et al., 2019)

ALBERT[40] is "A Lite" version of BERT, the popular unsupervised language representation learning algorithm. ALBERT uses parameter-reduction techniques that allow

for large-scale configurations, overcome previous memory limitations, and achieve better behavior with respect to model degradation.
Increasing model size when pretraining natural language representations often results in improved performance on downstream tasks. However, at some point further model increases become harder due to GPU/TPU memory limitations, longer training times, and unexpected model degradation. To address these problems, Google AI researchers present two parameter-reduction techniques to lower memory consumption and increase the training speed of BERT (Devlin et al., 2019)[22]. Comprehensive empirical evidence shows that their proposed methods lead to models that scale much better compared to the original BERT. They also use a selfsupervised loss that focuses on modeling inter-sentence coherence, and show it consistently helps downstream tasks with multi-sentence inputs. As a result, their best model establishes new state-of-the-art results on the GLUE, RACE, and SQuAD benchmarks while having fewer parameters compared to BERT-large.
The backbone of the ALBERT architecture is similar to BERT in that it uses a transformer encoder (Vaswani et al., 2017)[33] with GELU nonlinearities (Hendrycks & Gimpel, 2016)[41]. They follow the BERT notation conventions and denote the vocabulary embedding size as E, the number of encoder layers as L, and the hidden size as H. Following Devlin et al. (2019)[22], they set the feed-forward/filter size to be $4H$ and the number of attention heads to be $H/64$.
There are three main contributions that ALBERT makes over the design choices of BERT:

- **Factorized embedding parameterization**: in BERT, as well as subsequent modeling improvements such as XLNet (Yang et al., 2019)[42] and RoBERTa (Liu et al., 2019)[43], the WordPiece embedding size E is tied with the hidden layer size H, i.e., $E \equiv H$. This decision appears suboptimal for both modeling and practical reasons, as follows:

  - from a modeling perspective, WordPiece embeddings are meant to learn context-independent representations, whereas hidden-layer embeddings are meant to learn context-dependent representations. As the power of BERT-like representations comes from the use of context to provide the signal for learning such context-dependent representations, untying the WordPiece embedding size E from the hidden layer size H allows us to make a more efficient usage of the total model parameters as informed by modeling needs, which dictate that $H >> E$.
  - from a practical perspective, natural language processing usually require the vocabulary size V to be large. If $E \equiv H$, then increasing H increases the size of the embedding matrix, which has size $V \times E$. This can easily result in a model

with billions of parameters, most of which are only updated sparsely during training. Therefore, ALBERT approach uses a factorization of the embedding parameters, decomposing them into two smaller matrices.
Instead of projecting the one-hot vectors directly into the hidden space of size H, ALBERT technique first projects them into a lower dimensional embedding space of size E, and then projects it to the hidden space. By using this decomposition, it reduces the embedding parameters from $O(V \times H)$ to $O(V \times E + E \times H)$. This parameter reduction is significant when $H >> E$.

- **Cross-layer parameter sharing**: for ALBERT, cross-layer parameter sharing has been proposed as another way to improve parameter efficiency. There are multiple ways to share parameters, e.g., only sharing feed-forward network (FFN) parameters across layers, or only sharing attention parameters.
The default decision for ALBERT is to share all parameters across layers.

- **Inter-sentence coherence loss**: in addition to the masked language modeling (MLM) loss (Devlin et al., 2019)[22], BERT uses an additional loss called next-sentence prediction (NSP). NSP is a binary classification loss for predicting whether two segments appear consecutively in the original text, as follows: positive examples are created by taking consecutive segments from the training corpus; negative examples are created by pairing segments from different documents; positive and negative examples are sampled with equal probability. The NSP objective was designed to improve performance on downstream tasks, such as natural language inference, that require reasoning about the relationship between sentence pairs. However, subsequent studies (Yang et al., 2019[42]; Liu et al., 2019[43]) found NSP's impact unreliable and decided to eliminate it, a decision supported by an improvement in downstream task performance across several tasks.
The ALBERT paper's authors conjecture that the main reason behind NSP's ineffectiveness is its lack of difficulty as a task, as compared to MLM. As formulated, NSP conflates topic prediction and coherence prediction in a single task. However, topic prediction is easier to learn compared to coherence prediction, and also overlaps more with what is learned using the MLM loss.
They maintain that inter-sentence modeling is an important aspect of language understanding, but the authors propose a loss based primarily on coherence. That is, for ALBERT, they use a sentence-order prediction (SOP) loss, which avoids topic prediction and instead focuses on modeling inter-sentence coherence. The SOP loss uses as positive examples the same technique as BERT (two consecutive segments from the same document), and as negative examples the same two consecutive

segments but with their order swapped. This forces the model to learn finer-grained distinctions about discourse-level coherence properties.
As it is shown in Sec. 4.6 of the original paper, it turns out that NSP cannot solve the SOP task at all (i.e., it ends up learning the easier topic-prediction signal, and performs at randombaseline level on the SOP task), while SOP can solve the NSP task to a reasonable degree, presumably based on analyzing misaligned coherence cues.
As a result, ALBERT models consistently improve downstream task performance for multi-sentence encoding tasks.

2.3 Document Embedding Techniques

2.3.1 Aggregating word embeddings

Word Embedding aggregation is a very intuitive way to construct document embeddings from meaningful word embeddings.
Given a document, you perform some vector arithmetics on all the vectors corresponding to the words of the document to summarize them into a single vector in the same embedding space.
Two such common summarisation operators are average and sum.

2.3.2 Doc2Vec (Mikolov et al., 2014)

Doc2Vec [44] is a set of approaches to represent documents as fixed length low dimensional vectors (also known as document embeddings).
The goal of Doc2Vec is to create a numeric representation of a document, regardless of its length. But unlike words, documents do not come in logical structures such as words, so the another method has to be found.
Hence, Doc2Vec which aims at learning how to project a document into a latent d-dimensional space.

2.3.2.1 Paragraph Vector Distributed Memory (PV-DM)

Similarly to W2V-CBOW model, the central idea is: randomly sample consecutive words from a paragraph and predict a center word from the randomly sampled set of words by taking as input — the **context words** and a **paragraph id**.
In Figure 2.18 and 2.19 one can see the embedding training elements which are:

1. **Paragraph matrix**: it is the matrix where each row represents the vector of a paragraph;

2. **Average/Concatenate**: it means whether the word vectors and paragraph vector are averaged or concatenated;

3. **Classifier**: this part takes the hidden layer vector (the one that was concatenated/averaged) as input and predicts the center word.

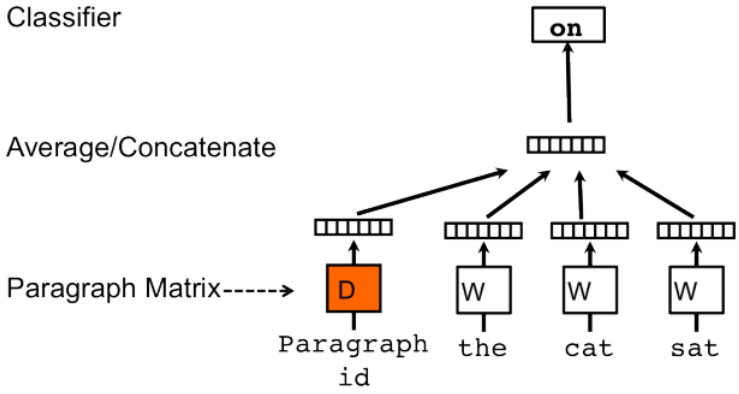


Figure 2.18: Paragraph Vector - Distributed Memory model (PV-DM).

In Figure 2.19 one can go deeper in the training neural network architecture. Below are some architectural details:

- when training the word vectors *W*, the document vector *D* is trained as well, and in the end of training, it holds a numeric representation of the paragraph;
- the word vector is a one-hot vector with a dimension *1 x V*;
- the paragraph ID vector has a dimension of *1 x C*, where *C* is the number of paragraphs;
- *N* represents the dimension of the hidden layer;
- the dimension of the weight matrix *W* of the hidden layer is *V x N*;
- the dimension of the weight matrix *D* of the hidden layer is *C x N*.

2.3.2.2 Paragraph Vector Distributed Bag Of Words (PV-DBOW)

The DBOW model ignores the context words in the input, but force the model to predict words randomly sampled from the paragraph in the output. So, in order to learn the document vector, 4 words are sampled from {the, cat, sat, on, the, sofa}, as shown in Figure 2.20.

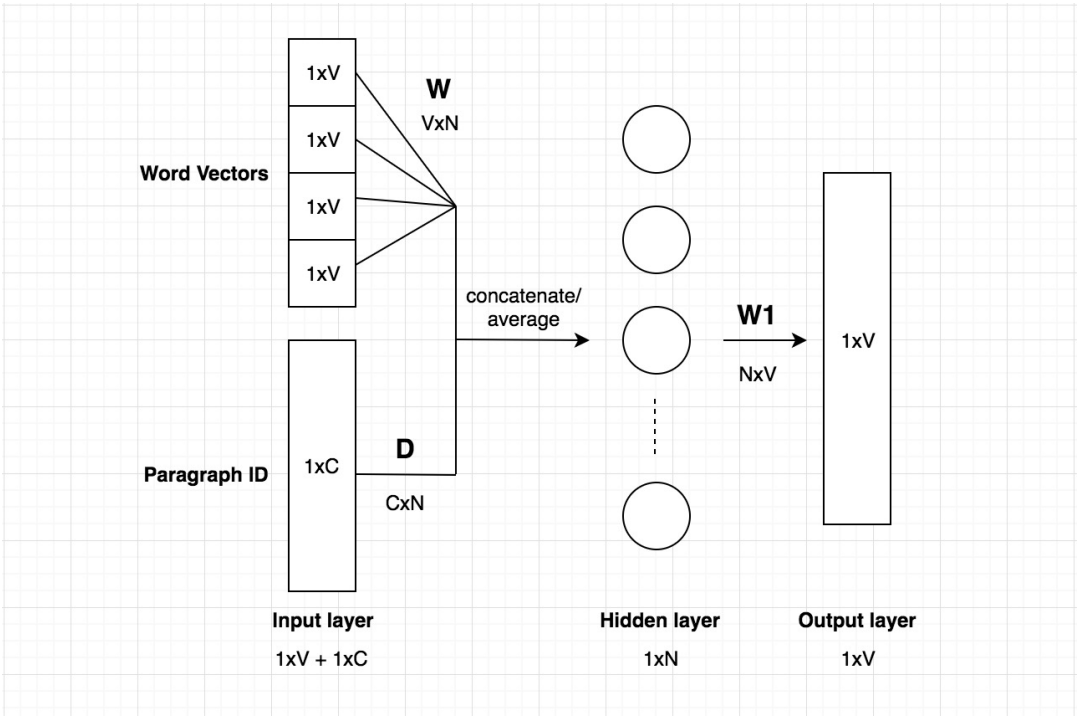


Figure 2.19: Paragraph Vector - Neural Network Architecture of the Paragraph Vector Distributed Memory model (PV-DM).

2.4 Embedding techniques in LAILA project

As we want to analyze legal sentences, we will need to represent the text in a way that:

- a Machine Learning algorithm could treat them;
- the semantic was preserved by the obtained representation;
- the text encoding let out algorithm perform some kind of language reasoning.

All the presented techniques are applicable but we do not know which of them will lead to good performance because it also depends on the specific considered dataset. Thus, we will have to compare them and choose the right one in order to represent legal sentences texts and obtain good results.
Flair, ELMo, BERT and ALBERT embeddings could let us achieve a good performance since:

- they reached State-Of-Art results in several NLP tasks;
- the authors released pre-trained models, which they have only to be fine-tuned
- these approaches provide the model with some kind of language understanding capacity which is really important when it comes to perform tasks like Sentences Summarisation.

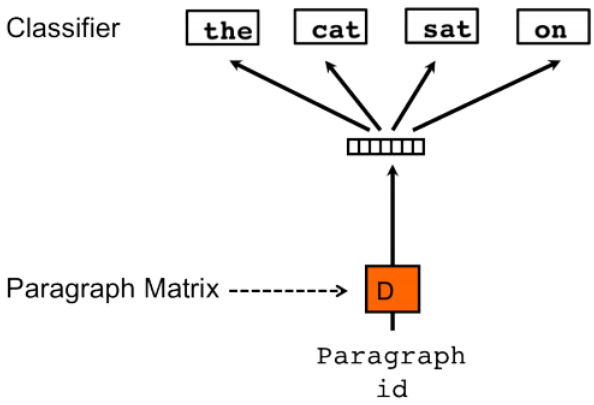


Figure 2.20: Paragraph Vector - Distributed Bag Of Words model (PV-DBOW)

2.5 Final considerations on language representation models

2.5.1 Embedding techniques variety

Several embeddings generation techniques have been proposed over recent years.
Latest methods like BERT and Flair seem to overcome the performance of the previous ones speaking in efficiency and accuracy terms.

2.5.2 The importance of Word Embedding method choice

As mentioned before, you have to choose the right word embedding technique in order to achieve a good training efficiency and a high accuracy of your built language model.
The choice of the right word embedding techinque in LAILA project will be a core step in the analysis process.
Some algorithms can perform better on some dataset while others could lead to worse results.
Luckily, the techniques presented are implemented in many NLP libraries (e.g. Flair NLP Library [45]) so we will be able to apply them and compare the results.

Chapter 3

Automatic Summarisation

3.1 Introduction

Automatic summarization [46] is the process of shortening a text document with software, in order to create a summary with the major points of the original document. Technologies that can make a coherent summary take into account variables such as length, writing style and syntax.

Automatic data summarization is part of machine learning and data mining. The main idea of summarization is to find a subset of data which contains the "information" of the entire set. Such techniques are widely used in industry today. Search engines are an example; others include summarization of documents, image collections and videos. Document summarization tries to create a representative summary or abstract of the entire document, by finding the most informative sentences, while in image summarization the system finds the most representative and important (i.e. salient) images. For surveillance videos, one might want to extract the important events from the uneventful context.

There are two general approaches to automatic summarization: extraction and abstraction. Extractive methods work by selecting a subset of existing words, phrases, or sentences in the original text to form the summary. In contrast, abstractive methods build an internal semantic representation and then use natural language generation techniques to create a summary that is closer to what a human might express. Such a summary might include verbal innovations. Research to date has focused primarily on extractive methods, which are appropriate for image collection summarization and video summarization.

An example of a summarization problem is document summarization, which attempts to automatically produce an abstract from a given document. Sometimes one might be interested in generating a summary from a single source document, while others can use multiple source documents (for example, a cluster of articles on the same topic). This problem is called multi-document summarization. A related application is summarizing news articles. Imagine a system, which automatically pulls together news articles on a given

topic (from the web), and concisely represents the latest news as a summary.

3.2 Summarisation Types: Extractive and Abstractive

There are broadly two different approaches that are used for text summarization [47]:

- **Extractive Summarization**
- **Abstractive Summarization**

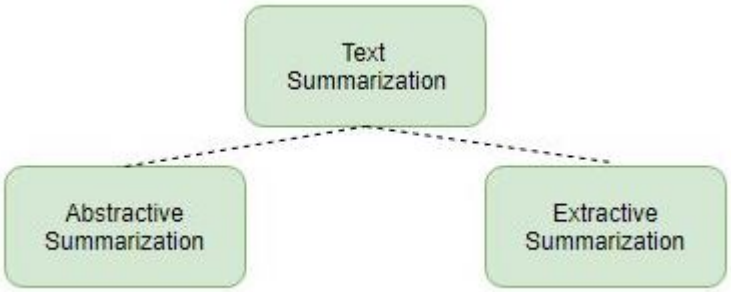


Figure 3.1: Text Summarisation classification

The extractive approach selects passages from the source text and then arranges it to form a summary. One way of thinking about this is like a highlighter underlining the important sections. The main idea is that the summarized text is a sub portion of the source text. [48]

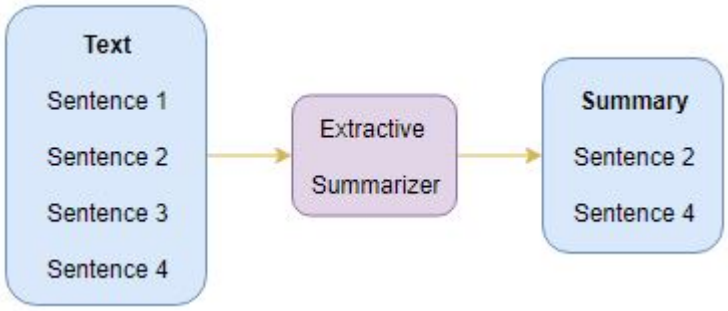


Figure 3.2: Extractive Summarisation high-level diagram

In contrast, abstractive approach involves understanding the intent and writes the summary in your own words. Naturally abstractive summarization is the more challenging problem here. This is one domain where machine learning has made slow progress. It is a difficult problem since creating abstractive summaries requires good command of the subject and of natural language which can both be difficult tasks for a machine. Also historically we didn't have a good and big data set for this problem. Data set here meaning source text with its abstractive summary. Since humans need to write the summaries, getting a lot of them is a problem (except in the News domain).

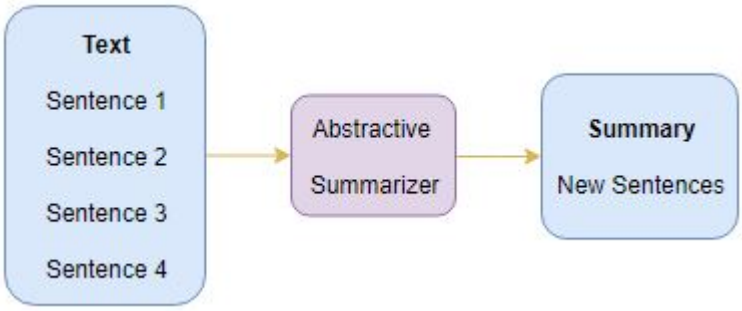


Figure 3.3: Abstractive Summarisation high-level diagram

3.3 Main Text Summarisation Applications: Keyphrase extraction and Document summarization

The keyphrase extraction task is the following. You are given a piece of text, such as a journal article, and you must produce a list of keywords or key[phrase]s that capture the primary topics discussed in the text. In the case of research articles, many authors provide manually assigned keywords, but most text lacks pre-existing keyphrases. For example, news articles rarely have keyphrases attached, but it would be useful to be able to automatically do so for a number of applications discussed below. Consider the example text from a news article:
"The Army Corps of Engineers, rushing to meet President Bush's promise to protect New Orleans by the start of the 2006 hurricane season, installed defective flood-control pumps last year despite warnings from its own expert that the equipment would fail during a storm, according to documents obtained by The Associated Press".
A keyphrase extractor might select "Army Corps of Engineers", "President Bush", "New Orleans", and "defective flood-control pumps" as keyphrases. These are pulled directly from the text. In contrast, an abstractive keyphrase system would somehow internalize the content and generate keyphrases that do not appear in the text, but more closely resemble what a human might produce, such as "political negligence" or "inadequate protection from floods". Abstraction requires a deep understanding of the text, which makes it difficult for a computer system. Keyphrases have many applications. They can enable document browsing by providing a short summary, improve information retrieval (if documents have keyphrases assigned, a user could search by keyphrase to produce more reliable hits than a full-text search), and be employed in generating index entries for a large text corpus.
Like keyphrase extraction, document summarization aims to identify the essence of a text. The only real difference is that now we are dealing with larger text units—whole sentences instead of words and phrases.

3.3.1 Multi-document summarization

Multi-document summarization is an automatic procedure aimed at extraction of information from multiple texts written about the same topic. Resulting summary report allows individual users, such as professional information consumers, to quickly familiarize themselves with information contained in a large cluster of documents. In such a way, multi-document summarization systems are complementing the news aggregators performing the next step down the road of coping with information overload.
Multi-document summarization may also be done in response to a question.
Multi-document summarization creates information reports that are both concise and comprehensive. With different opinions being put together and outlined, every topic is described from multiple perspectives within a single document. While the goal of a brief summary is to simplify information search and cut the time by pointing to the most relevant source documents, comprehensive multi-document summary should itself contain the required information, hence limiting the need for accessing original files to cases when refinement is required. Automatic summaries present information extracted from multiple sources algorithmically, without any editorial touch or subjective human intervention, thus making it completely unbiased.

3.4 Text Summarization Techniques

3.4.1 Extractive-based Summarisation

3.4.1.1 Supervised learning approaches: Neural Networks for Text Classification

Beginning with the work of Turney [49], many researchers have approached keyphrase extraction as a supervised machine learning problem. Given a document, we construct an example for each unigram, bigram, and trigram found in the text (though other text units are also possible, as discussed below). We then compute various features describing each example (e.g., does the phrase begin with an upper-case letter?). We assume there are known keyphrases available for a set of training documents. Using the known keyphrases, we can assign positive or negative labels to the examples. Then we learn a classifier that can discriminate between positive and negative examples as a function of the features. Some classifiers make a binary classification for a test example, while others assign a probability of being a keyphrase. For instance, in the above text, we might learn a rule that says phrases with initial capital letters are likely to be keyphrases. After training a learner, we can select keyphrases for test documents in the following manner. We apply the same example-generation strategy to the test documents, then run each example through the learner. We can determine the keyphrases by looking at binary classification decisions or probabilities returned from our learned model. If probabilities are given, a threshold is used

to select the keyphrases. Keyphrase extractors are generally evaluated using precision and recall. Precision measures how many of the proposed keyphrases are actually correct. Recall measures how many of the true keyphrases your system proposed. The two measures can be combined in an F-score, which is the harmonic mean of the two ($F = \frac{2 \times P \times R}{P+R}$). Matches between the proposed keyphrases and the known keyphrases can be checked after stemming or applying some other text normalization.

Designing a supervised keyphrase extraction system involves deciding on several choices (some of these apply to unsupervised, too). The first choice is exactly how to generate examples. Turney and others have used all possible unigrams, bigrams, and trigrams without intervening punctuation and after removing stopwords. Hulth showed that you can get some improvement by selecting examples to be sequences of tokens that match certain patterns of part-of-speech tags. Ideally, the mechanism for generating examples produces all the known labeled keyphrases as candidates, though this is often not the case. For example, if we use only unigrams, bigrams, and trigrams, then we will never be able to extract a known keyphrase containing four words. Thus, recall may suffer. However, generating too many examples can also lead to low precision.

We also need to create features that describe the examples and are informative enough to allow a learning algorithm to discriminate keyphrases from non- keyphrases. Typically features involve various term frequencies (how many times a phrase appears in the current text or in a larger corpus), the length of the example, relative position of the first occurrence, various boolean syntactic features (e.g., contains all caps), etc. The Turney paper used about 12 such features. Hulth uses a reduced set of features, which were found most successful in the KEA (Keyphrase Extraction Algorithm) work derived from Turney's seminal paper.

In the end, the system will need to return a list of keyphrases for a test document, so we need to have a way to limit the number. Ensemble methods (i.e., using votes from several classifiers) have been used to produce numeric scores that can be thresholded to provide a user-provided number of keyphrases. This is the technique used by Turney with C4.5 decision trees. Hulth used a single binary classifier so the learning algorithm implicitly determines the appropriate number.

Once examples and features are created, we need a way to learn to predict keyphrases. Virtually any supervised learning algorithm could be used, such as neural networks, decision trees, Naive Bayes, and rule induction.

Neural Networks for Text Classification.

Neural networks for text classification task include:

- **Feedforward Networks**. A feedforward neural network [50] is an artificial neural network wherein connections between the nodes do not form a cycle. As such, it is

different from recurrent neural networks.
The feedforward neural network was the first and simplest type of artificial neural network devised. [51] [52] [53] In this network, the information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) and to the output nodes. There are no cycles or loops in the network.

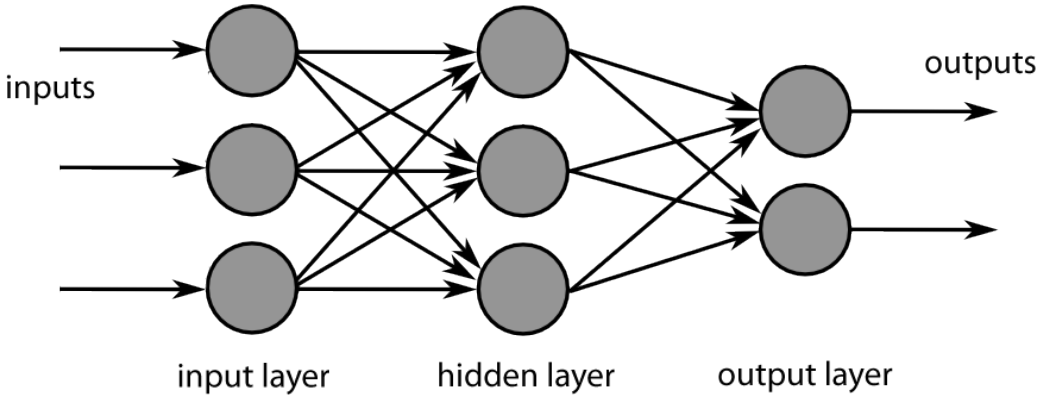


Figure 3.4: An example of Feed Forward Neural Network architecture.

- **LSTM Networks**. Long short-term memory (LSTM)[54][55] is an artificial recurrent neural network (RNN) architecture used in the field of deep learning. Unlike standard feedforward neural networks, LSTM has feedback connections. It can not only process single data points (such as images), but also entire sequences of data (such as speech or video).
  A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell.
  LSTM networks are well-suited to classifying, processing and making predictions based on time series data, since there can be lags of unknown duration between important events in a time series. LSTMs were developed to deal with the exploding and vanishing gradient problems that can be encountered when training traditional RNNs.
  A LSTM architecture to mention is GRU [56]: Gated Recurrent Units (GRUs) are a gating mechanism in recurrent neural networks, introduced in 2014 by Kyunghyun Cho et al. [57] The GRU is like a long short-term memory (LSTM) with forget gate but has fewer parameters than LSTM, as it lacks an output gate. GRU's performance on certain tasks of polyphonic music modeling and speech signal modeling was found to be similar to that of LSTM. GRUs have been shown to exhibit even better performance on certain smaller datasets.

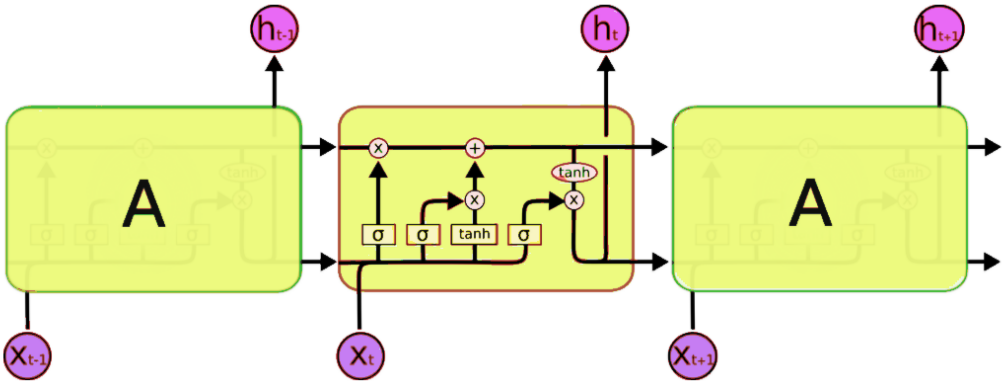


Figure 3.5: LSTM Neural Network architecture

- **CNNs** [58]. In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, most commonly applied to analyzing visual imagery. They are also known as shift invariant or space invariant artificial neural networks (SIANN), based on their shared-weights architecture and translation invariance characteristics. They have applications in image and video recognition, recommender systems, image classification, medical image analysis, and natural language processing.
  CNNs are regularized versions of multilayer perceptrons. Multilayer perceptrons usually mean fully connected networks, that is, each neuron in one layer is connected to all neurons in the next layer. The "fully-connectedness" of these networks makes them prone to overfitting data. Typical ways of regularization include adding some form of magnitude measurement of weights to the loss function. However, CNNs take a different approach towards regularization: they take advantage of the hierarchical pattern in data and assemble more complex patterns using smaller and simpler patterns. Therefore, on the scale of connectedness and complexity, CNNs are on the lower extreme.
  Convolutional networks were inspired by biological processes in that the connectivity pattern between neurons resembles the organization of the animal visual cortex. [59] Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field.
  CNNs use relatively little pre-processing compared to other image classification algorithms. This means that the network learns the filters that in traditional algorithms were hand-engineered. This independence from prior knowledge and human effort in feature design is a major advantage.
  Most recently, however, Convolutional Neural Networks have also found prevalence

in tackling problems associated with NLP tasks like Sentence Classification [60], Text Classification, Sentiment Analysis, Text Summarization, Machine Translation and Answer Relations. [61]

So, how does any of this apply to NLP?

Instead of image pixels, the input to most NLP tasks are sentences or documents represented as a matrix. Each row of the matrix corresponds to one token, typically a word, but it could be a character. That is, each row is vector that represents a word. Typically, these vectors are word embeddings (low-dimensional representations) like word2vec or GloVe, but they could also be one-hot vectors that index the word into a vocabulary. For a 10 word sentence using a 100-dimensional embedding we would have a 10×100 matrix as our input. That's our "image". [62] In Figure 3.6 an example of a CNN for text classification is shown.

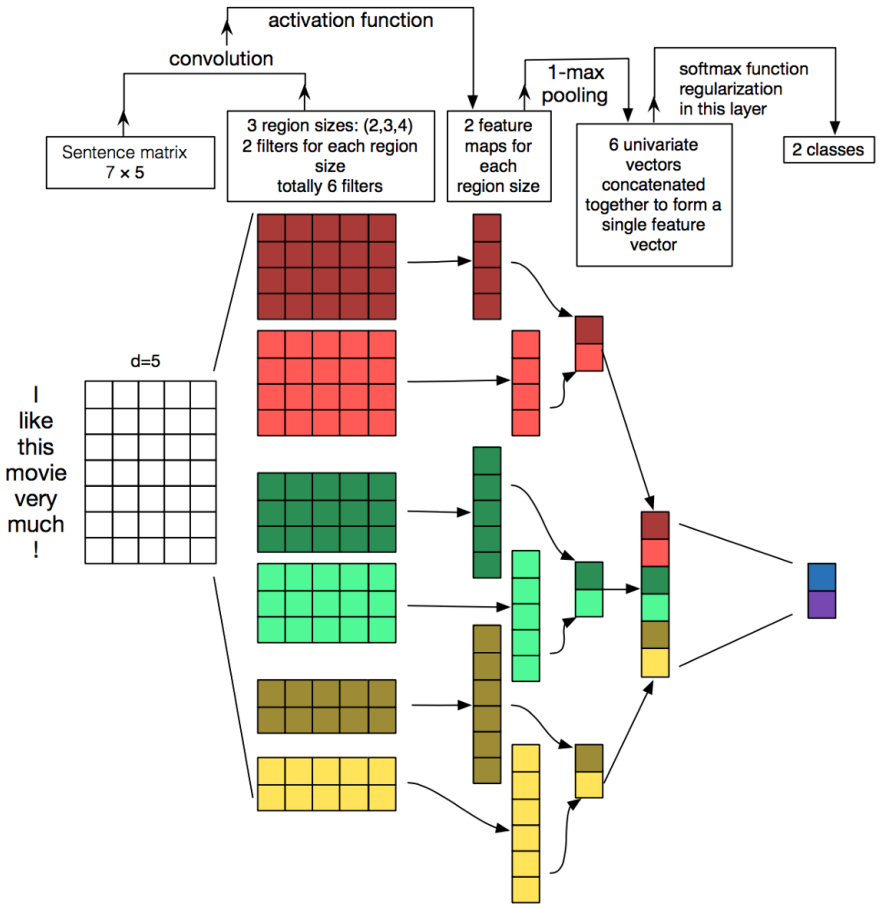


Figure 3.6: An example of CNN for text classification.

Examples of application of Neural Networks to Text Classification.

One of the most interesting and successful application of neural networks models to the text

classification task is represented by the sentiment analysis field. Sentiment Analysis is a field of Natural Language Processing responsible for systems that can extract opinions from natural language. [63]
Examples of related works in the literature which apply Deep Learning models to this task are: "Cross-domain & In-domain Sentiment Analysis with Memory-based Deep Neural Networks" (Moro et al., 2018) [64], "Transfer Learning in Sentiment Classification with Deep Neural Networks" (Pagliarani et al., 2017) [65] and "On Deep Learning in Cross-Domain Sentiment Classification" (Domeniconi et al., 2017) [66].
Another application of NNs to text classification is message thread identification. Example of related works are "Identifying Conversational Message Threads by Integrating Classification and Data Clustering" (Domeniconi et al., 2016) [67] and "A Novel Method for Unsupervised and Supervised Conversational Message Thread Detection" (Domeniconi et al., 2016) [68]. These works show how Deep Learning has given a boost to sentiment classification due to its intrinsic ability in mining hidden relationships in text. Indeed, Deep Learning approaches are usually more robust and efficient than those based on classical text mining techniques (e.g. bag-of-words) used in previous works [69] [70] [71] [72] [73] [74], because their performance typically scales better with dataset size both in terms of accuracy and from the computational point of view.

This is not to say that previous techniques such as bag-of-words should no longer be considered. In fact, the latter can be useful in the following cases:

- Building a baseline model. By using libraries such as scikit-learn, only a few lines of code are needed to build model. Later on, one can use Deep Learning to beat it.

- If your dataset is small and context is domain specific (you cannot find corresponding vector from pre-trained word embedding models such as GloVe and fastText), BoW may work better than Word Embedding.

For instance, "Personalized Web Search via Query Expansion Based on User's Local Hierarchically-Organized Files" (Moro et al., 2017) [75] is an example of work that improves personalized web searches relevance applying the bag-of-words model.

Another successful application of the Bag-Of-Words model is the prediction of Stock Market Dow Jones via Tweet text mining. Examples of such works are "Learning to Predict the Stock Market Dow Jones Index Detecting and Mining Relevant Tweets" (Domeniconi et al, 2017) [76] and "Prediction and Trading of Dow Jones from Twitter: A Boosting Text Mining Method with Relevant Tweets Identification" (Moro et al., 2017) [77].

3.4.1.2 Unsupervised approaches: TextRank and LexRank

A popular unsupervised keyphrase extraction algorithm is TextRank [78]. While supervised methods have some nice properties, like being able to produce interpretable rules for what features characterize a keyphrase, they also require a large amount of training data. Many documents with known keyphrases are needed. Furthermore, training on a specific domain tends to customize the extraction process to that domain, so the resulting classifier is not necessarily portable, as some of Turney's results demonstrate. Unsupervised keyphrase extraction removes the need for training data. It approaches the problem from a different angle. Instead of trying to learn explicit features that characterize keyphrases, the TextRank algorithm exploits the structure of the text itself to determine keyphrases that appear "central" to the text in the same way that PageRank [79] selects important Web pages. Recall this is based on the notion of "prestige" or "recommendation" from social networks. In this way, TextRank does not rely on any previous training data at all, but rather can be run on any arbitrary piece of text, and it can produce output simply based on the text's intrinsic properties. Thus the algorithm is easily portable to new domains and languages. TextRank is a general purpose graph-based ranking algorithm for NLP. Essentially, it runs PageRank on a graph specially designed for a particular NLP task. For keyphrase extraction, it builds a graph using some set of text units as vertices. Edges are based on some measure of semantic or lexical similarity between the text unit vertices. Unlike PageRank, the edges are typically undirected and can be weighted to reflect a degree of similarity. Once the graph is constructed, it is used to form a stochastic matrix, combined with a damping factor (as in the "random surfer model"), and the ranking over vertices is obtained by finding the eigenvector corresponding to eigenvalue 1 (i.e., the stationary distribution of the random walk on the graph).
The vertices should correspond to what we want to rank. Potentially, we could do something similar to the supervised methods and create a vertex for each unigram, bigram, trigram, etc. However, to keep the graph small, the authors decide to rank individual unigrams in a first step, and then include a second step that merges highly ranked adjacent unigrams to form multi-word phrases. This has a nice side effect of allowing us to produce keyphrases of arbitrary length. For example, if we rank unigrams and find that "advanced", "natural", "language", and "processing" all get high ranks, then we would look at the original text and see that these words appear consecutively and create a final keyphrase using all four together. Note that the unigrams placed in the graph can be filtered by part of speech. The authors found that adjectives and nouns were the best to include. Thus, some linguistic knowledge comes into play in this step.
Edges are created based on word co-occurrence in this application of TextRank. Two vertices are connected by an edge if the unigrams appear within a window of size N in the

original text. N is typically around 2–10. Thus, "natural" and "language" might be linked in a text about NLP. "Natural" and "processing" would also be linked because they would both appear in the same string of N words. These edges build on the notion of "text cohesion" and the idea that words that appear near each other are likely related in a meaningful way and "recommend" each other to the reader.

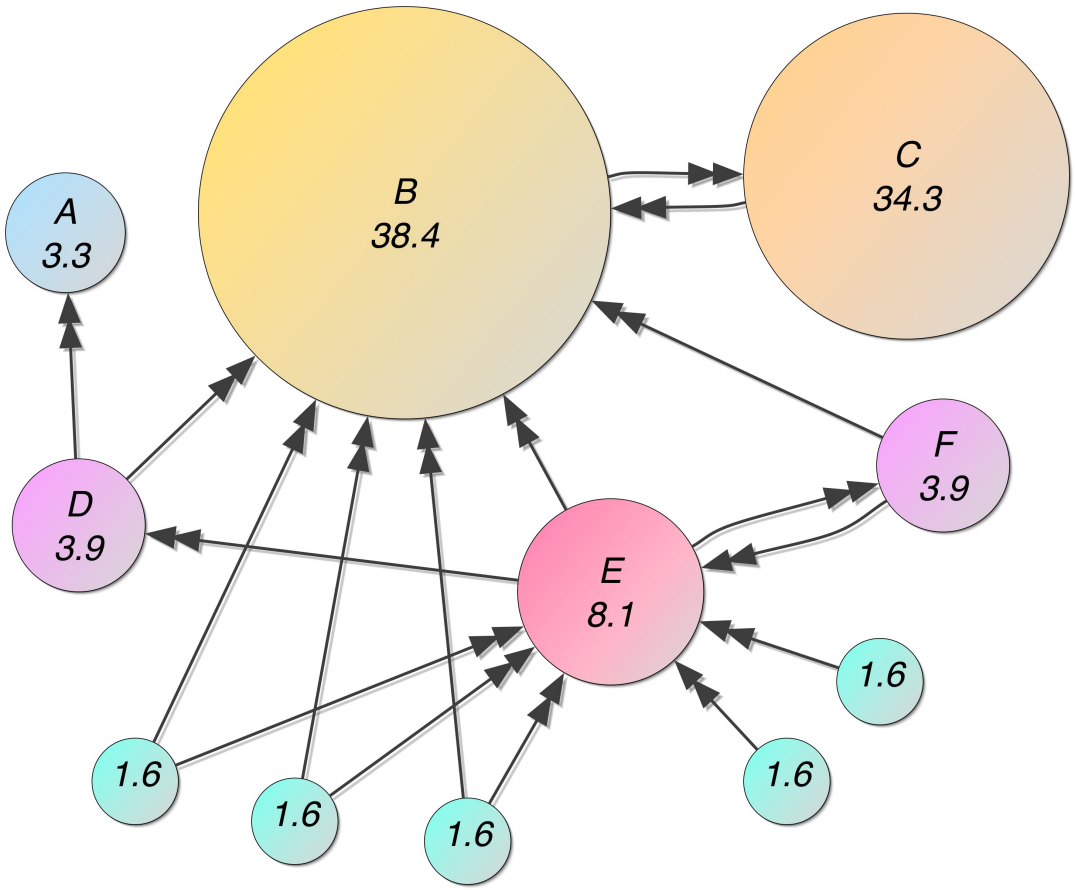


Figure 3.7: An example of PageRank graph.

Since this method simply ranks the individual vertices, we need a way to threshold or produce a limited number of keyphrases. The technique chosen is to set a count T to be a user-specified fraction of the total number of vertices in the graph. Then the top T vertices/unigrams are selected based on their stationary probabilities. A post- processing step is then applied to merge adjacent instances of these T unigrams. As a result, potentially more or less than T final keyphrases will be produced, but the number should be roughly proportional to the length of the original text.

It is not initially clear why applying PageRank to a co-occurrence graph would produce useful keyphrases. One way to think about it is the following. A word that appears multiple times throughout a text may have many different co-occurring neighbors. For example, in a text about machine learning, the unigram "learning" might co-occur with "machine",

"supervised", "un-supervised", and "semi-supervised" in four different sentences. Thus, the "learning" vertex would be a central "hub" that connects to these other modifying words. Running PageRank/TextRank on the graph is likely to rank "learning" highly. Similarly, if the text contains the phrase "supervised classification", then there would be an edge between "supervised" and "classification". If "classification" appears several other places and thus has many neighbors, its importance would contribute to the importance of "supervised". If it ends up with a high rank, it will be selected as one of the top T unigrams, along with "learning" and probably "classification". In the final post-processing step, we would then end up with keyphrases "supervised learning" and "supervised classification".
In short, the co-occurrence graph will contain densely connected regions for terms that appear often and in different contexts. A random walk on this graph will have a stationary distribution that assigns large probabilities to the terms in the centers of the clusters. This is similar to densely connected Web pages getting ranked highly by PageRank.

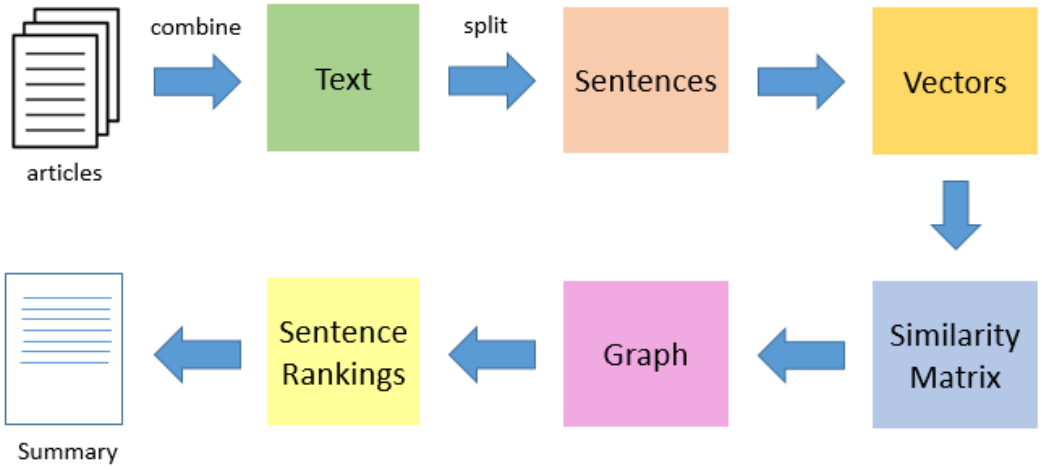


Figure 3.8: Text Summarisation process using TextRank algorithm.

LexRank [80] is an algorithm essentially identical to TextRank, and both use this approach for document summarization. The two methods were developed by different groups at the same time, and LexRank simply focused on summarization, but could just as easily be used for keyphrase extraction or any other NLP ranking task.
In both LexRank and TextRank, a graph is constructed by creating a vertex for each sentence in the document.
The edges between sentences are based on some form of semantic similarity or content overlap. While LexRank uses cosine similarity of TF-IDF vectors, TextRank uses a very similar measure based on the number of words two sentences have in common (normalized by the sentences' lengths). The LexRank paper explored using unweighted edges after applying a threshold to the cosine values, but also experimented with using edges with

weights equal to the similarity score. TextRank uses continuous similarity scores as weights.
In both algorithms, the sentences are ranked by applying PageRank to the resulting graph. A summary is formed by combining the top ranking sentences, using a threshold or length cutoff to limit the size of the summary.
It is worth noting that TextRank was applied to summarization exactly as described here, while LexRank was used as part of a larger summarization system (MEAD) that combines the LexRank score (stationary probability) with other features like sentence position and length using a linear combination with either user-specified or automatically tuned weights. In this case, some training documents might be needed, though the TextRank results show the additional features are not absolutely necessary.
Another important distinction is that TextRank was used for single document summarization, while LexRank has been applied to multi-document summarization. The task remains the same in both cases—only the number of sentences to choose from has grown. However, when summarizing multiple documents, there is a greater risk of selecting duplicate or highly redundant sentences to place in the same summary. Imagine you have a cluster of news articles on a particular event, and you want to produce one summary. Each article is likely to have many similar sentences, and you would only want to include distinct ideas in the summary. To address this issue, LexRank applies a heuristic post-processing step that builds up a summary by adding sentences in rank order, but discards any sentences that are too similar to ones already placed in the summary. The method used is called Cross-Sentence Information Subsumption (CSIS).

3.4.2 Abstractive-based Summarisation

3.4.2.1 Introduction to Sequence-to-Sequence (Seq2Seq) Modeling

We can build a Seq2Seq model on any problem which involves sequential information. This includes Sentiment classification, Neural Machine Translation, and Named Entity Recognition — some very common applications of sequential information. In the case of Neural Machine Translation, the input is a text in one language and the output is also a text in another language.

Figure 3.9: An example of language translation which could be tackled by a Seq2Seq model.

In the Named Entity Recognition, the input is a sequence of words and the output is a sequence of tags for every word in the input sequence.

Figure 3.10: Named Entity Recognition as a Sequence-to-Sequence model

Our objective is to build a text summarizer where the input is a long sequence of words (in a text body), and the output is a short summary (which is a sequence as well). So, we can model this as a Many-to-Many Seq2Seq problem. In Figure 4.0.2 a typical Seq2Seq model architecture is shown.

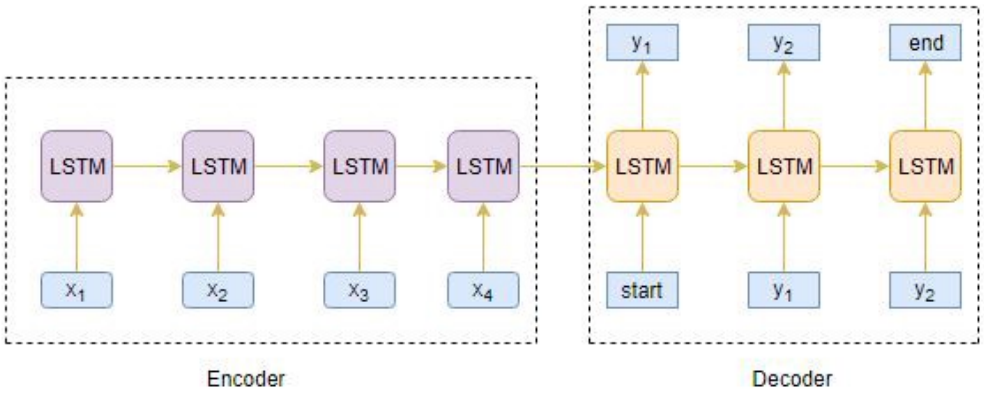


Figure 3.11: An example of Seq2Seq model architecture.

3.4.2.2 Understanding the Encoder-Decoder Architecture

The Encoder-Decoder architecture is mainly used to solve the sequence-to-sequence (Seq2Seq) problems where the input and output sequences are of different lengths. Let's understand this from the perspective of text summarization. The input is a long sequence of words and the output will be a short version of the input sequence. Generally,

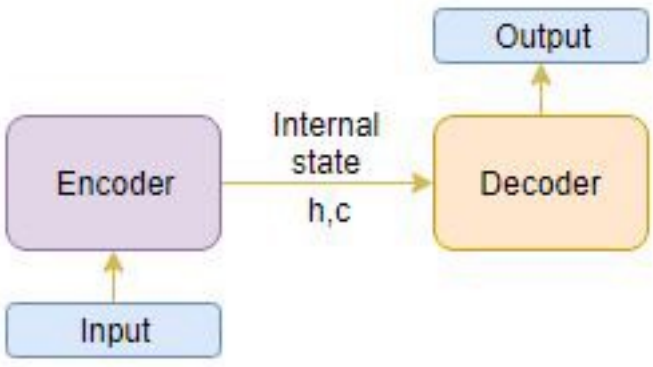


Figure 3.12: Encoder-Decoder high-level architecture.

variants of Recurrent Neural Networks (RNNs), i.e. Gated Recurrent Neural Network

(GRU) or Long Short Term Memory (LSTM), are preferred as the encoder and decoder components. This is because they are capable of capturing long term dependencies by overcoming the problem of vanishing gradient.

3.4.2.3 Training phase

In the training phase, we will first set up the encoder and decoder. We will then train the model to predict the target sequence offset by one timestep. Let us see in detail on how to set up the encoder and decoder.

3.4.2.3.1 Encoder An Encoder Long Short Term Memory model (LSTM) reads the entire input sequence wherein, at each timestep, one word is fed into the encoder. It then processes the information at every timestep and captures the contextual information present in the input sequence. In Figure 3.13 you can see the diagram which illustrates this process. The hidden state h_i and cell state c_i of the last time step is used to initialize the decoder.

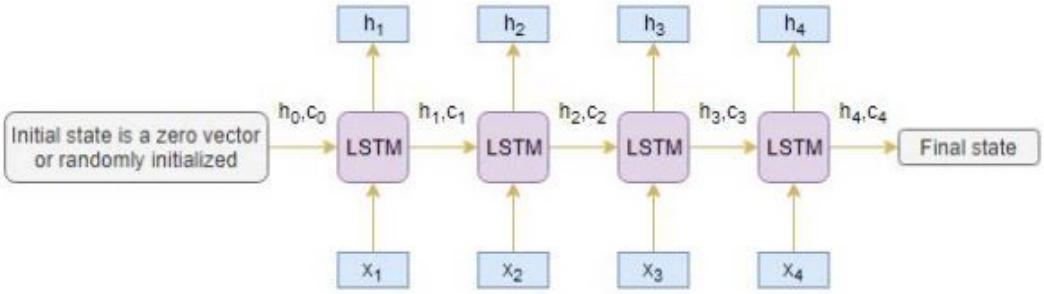


Figure 3.13: Data flow through the Encoder in a typical Seq2Seq model.

Remember, this is because the encoder and decoder are two different sets of LSTM architecture.

3.4.2.3.2 Decoder The decoder is also an LSTM network which reads the entire target sequence word-by-word and predicts the same sequence offset by one timestep. **The decoder is trained to predict the next word in the sequence given the previous word. <start>** and **<end>** are the special tokens which are added to the target sequence before feeding it into the decoder. The target sequence is unknown while decoding the test sequence. So, we start predicting the target sequence by passing the first word into the decoder which would be always the **<start>** token. And the **<end>** token signals the end of the sentence.

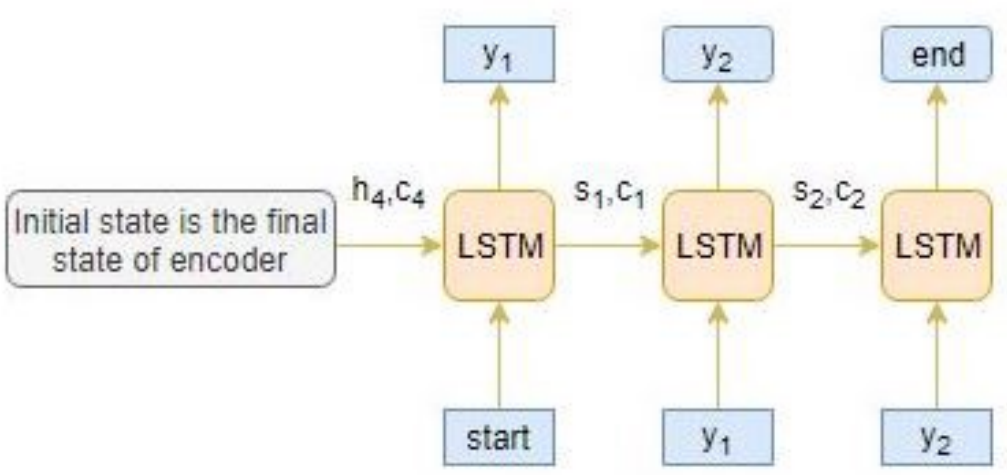


Figure 3.14: Data flow through the Decoder in a typical Seq2Seq model.

3.4.2.4 Inference Phase

After training, the model is tested on new source sequences for which the target sequence is unknown. So, we need to set up the inference architecture to decode a test sequence as in Figure 3.15.

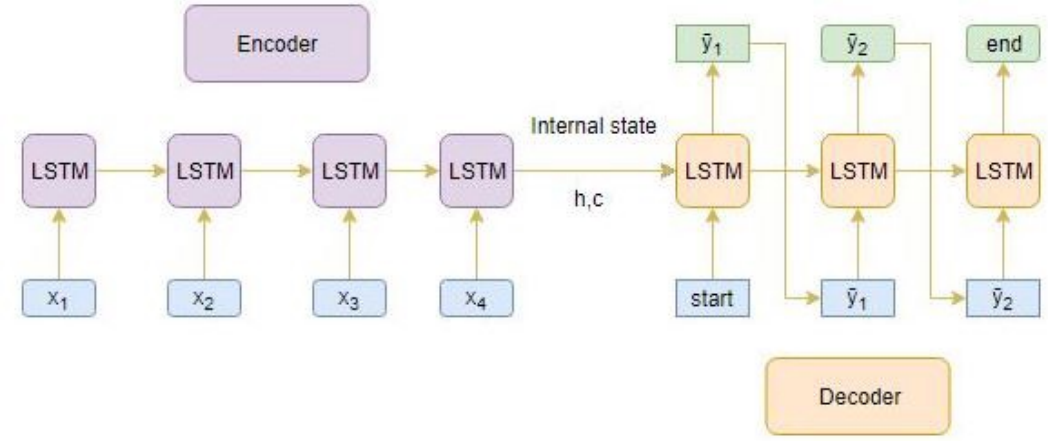


Figure 3.15: An example of Encoder-Decoder inference architecture.

3.4.2.4.1 How does the inference process work? Here are the steps to decode the test sequence:

1. Encode the entire input sequence and initialize the decoder with internal states of the encoder;
2. Pass **<start>** token as an input to the decoder;
3. Run the decoder for one time step with the internal states;

4. The output will be the probability for the next word. The word with the maximum probability will be selected;

5. Pass the sampled word as an input to the decoder in the next time step and update the internal states with the current time step;

6. Repeat steps 3–5 until we generate **<end>** token or hit the maximum length of the target sequence.

3.4.2.5 Limitations of the Encoder – Decoder Architecture

As useful as this encoder-decoder architecture is, there are certain limitations that come with it:

- the encoder converts the entire input sequence into a fixed length vector and then the decoder predicts the output sequence. **This works only for short sequences** since the decoder is looking at the entire input sequence for the prediction;

- here comes the problem with long sequences. **It is difficult for the encoder to memorize long sequences into a fixed length vector**.

So how do we overcome this problem of long sequences? This is where the concept of attention mechanism comes into the picture. It aims to predict a word by looking at a few specific parts of the sequence only, rather than the entire sequence.

3.4.2.6 The Intuition behind the Attention Mechanism

Let's consider a simple example to understand how Attention Mechanism works:

- Source sequence: "Which sport do you like the most?

- Target sequence: "I love cricket"

The first word 'I' in the target sequence is connected to the fourth word 'you' in the source sequence. Similarly, the second-word 'love' in the target sequence is associated with the fifth word 'like' in the source sequence.
So, instead of looking at all the words in the source sequence, we can increase the importance of specific parts of the source sequence that result in the target sequence. This is the basic idea behind the attention mechanism.
There are 2 different classes of attention mechanism depending on the way the attended context vector is derived:

- Global Attention

- Local Attention

3.4.2.6.1 Global Attention Here, the attention is placed on all the source positions. In other words, all the hidden states of the encoder are considered for deriving the attended context vector. This process is shown in Figure 3.16. An example of the entire encoding-

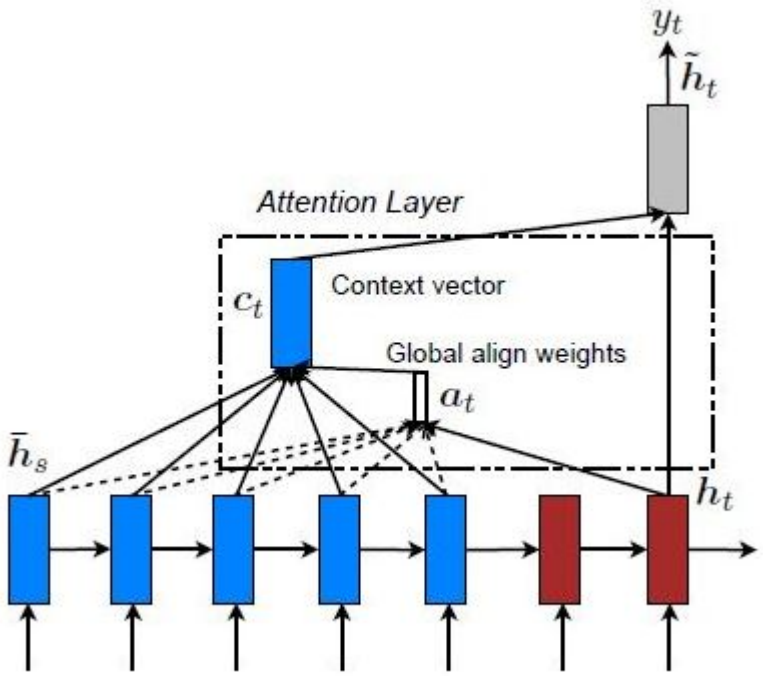


Figure 3.16: Illustration of the global attention mechanism.

decoding process with global attention is shown in Figure 3.17 [81].

3.4.2.6.2 Local Attention Here, the attention is placed on only a few source positions. Only a few hidden states of the encoder are considered for deriving the attended context vector. This process is shown in Figure 3.18.

3.4.2.7 Transformer architecture

Transformers are a type of neural network architecture that have been gaining popularity. Transformers were recently used by OpenAI in their language models, and also used recently by DeepMind for AlphaStar — their program to defeat a top professional Starcraft player. [82]

Transformers were developed to solve the problem of sequence transduction, or neural machine translation (Figure 3.19). That means any task that transforms an input sequence to an output sequence. This includes speech recognition, text-to-speech transformation, etc.. Transformer [33] is a model that uses attention to boost the speed. More specifically, it uses self-attention.

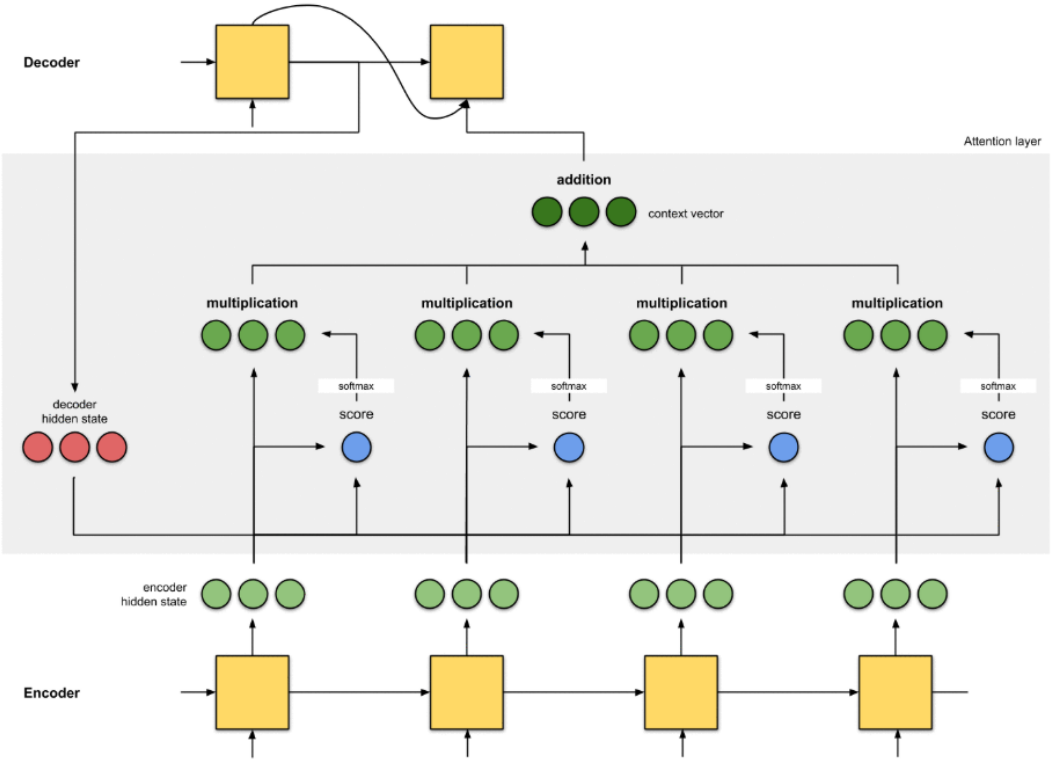


Figure 3.17: An example architecture of an Encoder-Decoder with Attention.

Internally, the Transformer has a similar kind of architecture as the previous models above. But the Transformer consists of six encoders and six decoders as one can see in Figure 3.20. Each encoder is very similar to each other. All encoders have the same architecture. Decoders share the same property, i.e. they are also very similar to each other. Each encoder consists of two layers: **Self-attention** and a **feed Forward Neural Network** (Figure 3.21).

The encoder's inputs first flow through a self-attention layer. It helps the encoder look at other words in the input sentence as it encodes a specific word. The decoder (Figure 3.22) has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence.

3.4.2.7.1 Self-Attention Let's start to look at the various vectors/tensors and how they flow between these components to turn the input of a trained model into an output. As is the case in NLP applications in general, we begin by turning each input word into a vector using an embedding algorithm.

Each word is embedded into a vector of size 512. Those vectors are represented with these simple boxes.

The embedding only happens in the bottom-most encoder. The abstraction that is common to all the encoders is that they receive a list of vectors each of the size 512.

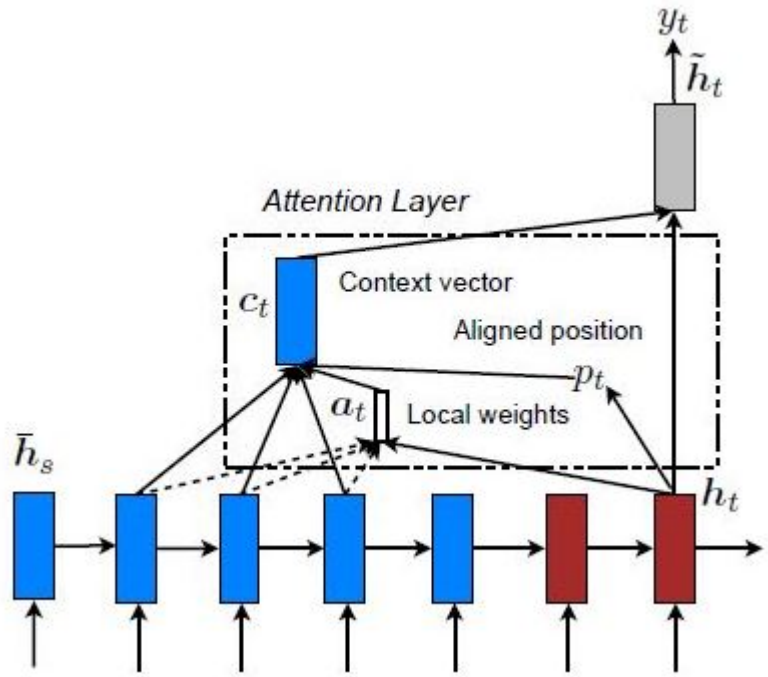


Figure 3.18: Illustration of the local attention mechanism.

Figure 3.19: Transformer high-level illustration.

In the bottom encoder that would be the word embeddings, but in other encoders, it would be the output of the encoder that's directly below. After embedding the words in our input sequence, each of them flows through each of the two layers of the encoder.

Here we begin to see one key property of the Transformer, which is that the word in each position flows through its own path in the encoder. There are dependencies between these paths in the self-attention layer. The feed-forward layer does not have those dependencies, however, and thus the various paths can be executed in parallel while flowing through the feed-forward layer.

Let us first look at how to calculate self-attention (Figure 3.25) using vectors, then proceed to look at how it is actually implemented — using matrices.

The first step in calculating self-attention is to create three vectors from each of the encoder's input vectors (in this case, the embedding of each word). So for each word, we

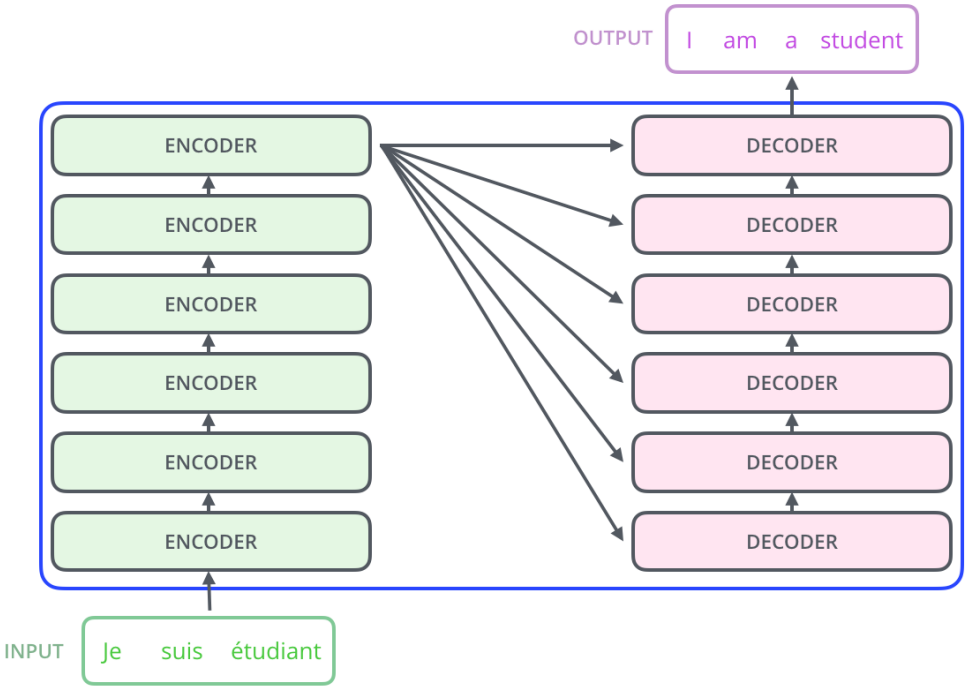


Figure 3.20: Transformer high-level encoders and decoders.

create a Query vector, a Key vector, and a Value vector. These vectors are created by multiplying the embedding by three matrices that we trained during the training process. Notice that these new vectors are smaller in dimension than the embedding vector. They do not have to be smaller than 64: this is an architecture choice to make the computation of multiheaded attention (mostly) constant.

Multiplying x_1 by the W^Q weight matrix produces q_1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

But **what are the "query", "key", and "value" vectors**?

They are abstractions that are useful for calculating and thinking about attention. Once you proceed with reading how attention is calculated below, you will know pretty much all you need to know about the role each of these vectors plays.

The second step in calculating self-attention is to calculate a score. Say we are calculating the self-attention for the first word in this example, "Thinking". We need to score each word of the input sentence against this word. The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring. So if we are processing the self-attention for the word in position #1, the first score would be the dot product of q_1 and k_1. The second score would be the dot product of q_1 and k_2 (and so on). This calculation is shown in Figure 3.27.

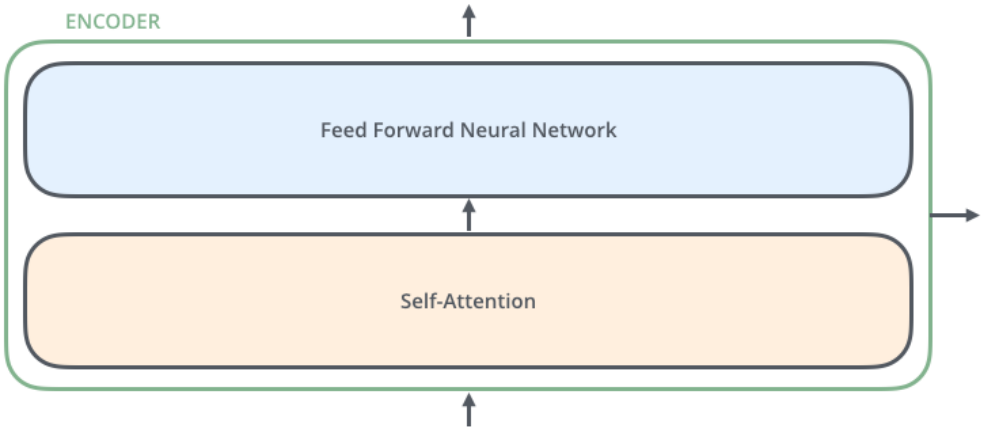


Figure 3.21: Transformer high-level encoder architecture.

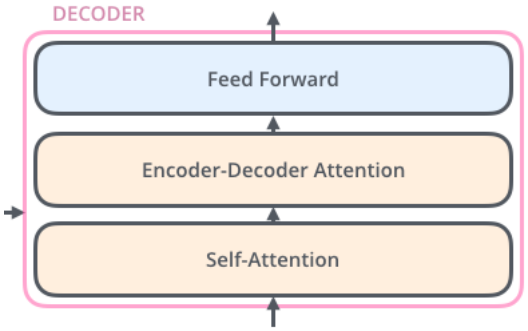


Figure 3.22: Transformer high-level decoder architecture.

The third and forth steps are to divide the scores by 8 (the square root of the dimension of the key vectors used in the paper — 64. This leads to having more stable gradients. There could be other possible values here, but this is the default), then pass the result through a softmax operation. Softmax normalizes the scores so they're all positive and add up to 1. The calculation described so far is shown in Figure 3.28. This softmax score determines how much each word will be expressed at this position. Clearly the word at this position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word.

The fifth step is to multiply each value vector by the softmax score (in preparation to sum them up). The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).

The sixth step is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word). The entire self-attention calculation is

Figure 3.23: Word embeddings associated to their corresponding words.

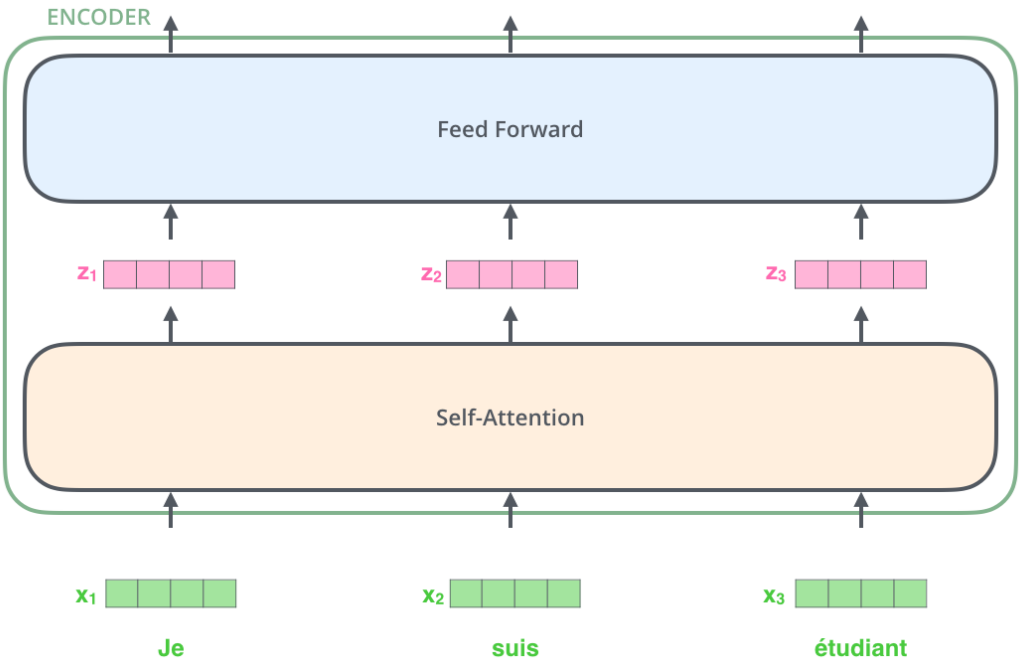


Figure 3.24: Transformer Encoder data flow.

shown in Figure 3.29. That concludes the self-attention calculation. The resulting vector is one we can send along to the feed-forward neural network. In the actual implementation, however, this calculation is done in matrix form for faster processing.

3.4.2.7.2 Multihead attention Transformers basically work like that. There are a few other details that make them work better. For example, instead of only paying attention to each other in one dimension, Transformers use the concept of Multihead attention.
The idea behind it is that whenever you are translating a word, you may pay different attention to each word based on the type of question that you are asking. The images below show what that means. For example, whenever you are translating “kicked” in the sentence “I kicked the ball”, you may ask “Who kicked”. Depending on the answer, the translation of the word to another language can change. Or ask other questions, like “Did what?”, etc.

3.4.2.7.3 Positional Encoding Another important step on the Transformer is to add positional encoding when encoding each word. Encoding the position of each word is relevant, since the position of each word is relevant to the translation.

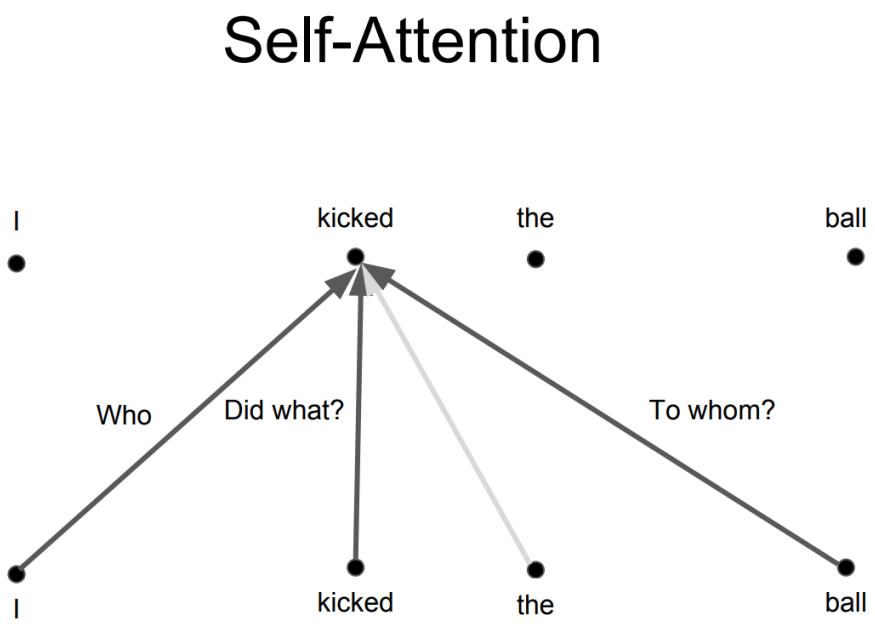


Figure 3.25: Self Attention mechanism representation.

3.4.2.8 GPT-2 Transformer

GPT-2 [1] is a large transformer-based language model with 1.5 billion parameters, trained on a dataset of 8 million web pages. [83]
GPT-2 is trained with a simple objective: predict the next word, given all of the previous words within some text. The diversity of the dataset causes this simple goal to contain naturally occurring demonstrations of many tasks across diverse domains. GPT-2 is a direct scale-up of GPT, with more than 10X the parameters and trained on more than 10X the amount of data.
GPT-2 displays a broad set of capabilities, including the ability to generate conditional synthetic text samples of unprecedented quality, where we prime the model with an input and have it generate a lengthy continuation. In addition, GPT-2 outperforms other language models trained on specific domains (like Wikipedia, news, or books) without needing to use these domain-specific training datasets. On language tasks like question answering, reading comprehension, summarization, and translation, GPT-2 begins to learn these tasks from the raw text, using no task-specific training data. While scores on these downstream tasks are far from state-of-the-art, they suggest that the tasks can benefit from unsupervised techniques, given sufficient (unlabeled) data and compute.
GPT-2 generates synthetic text samples in response to the model being primed with an arbitrary input. The model is chameleon-like—it adapts to the style and content of the

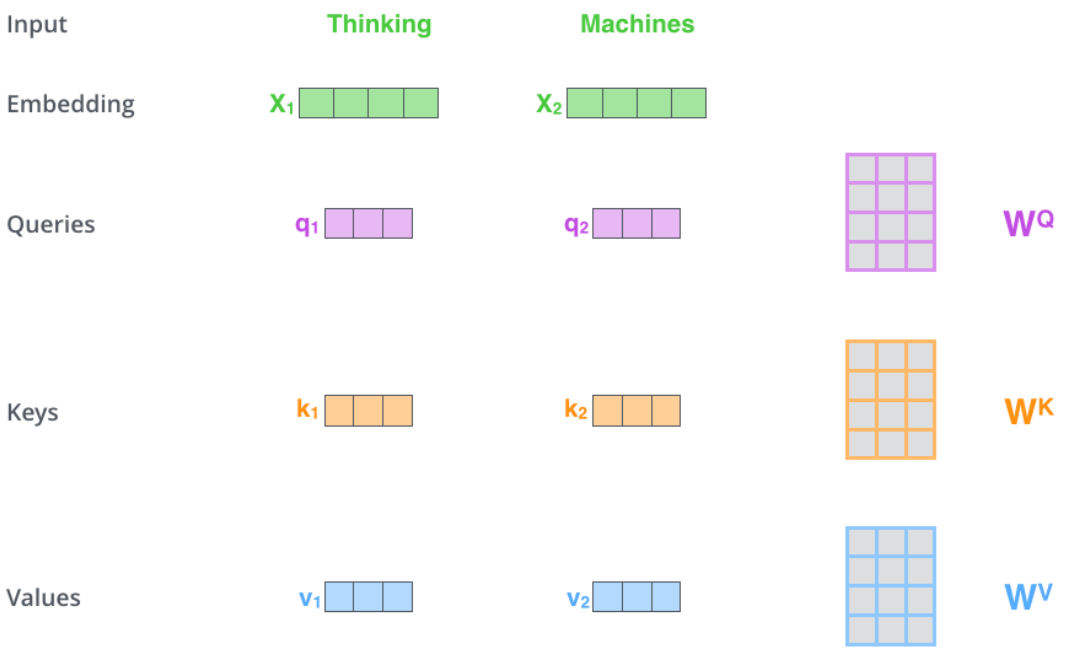


Figure 3.26: Self-Attention - Query, Key, Value vectors creation.

conditioning text. This allows the user to generate realistic and coherent continuations about a topic of their choosing, as seen by the following select samples.

3.4.2.9 Reformer: The Efficient Transformer

Recently, Google introduced the Reformer [84] [85] architecture, a Transformer model designed to efficiently handle processing very long sequences of data (e.g. up to 1 million words in a language processing). Execution of Reformer requires much lower memory consumption and achieves impressive performance even when running on only a single GPU. The paper Reformer: The efficient Transformer will be presented in ICLR 2020 (and has received a near-perfect score in the reviews). Reformer model is expected to have a significant impact on the filed by going beyond language applications (e.g. music, speech, image and video generation). [86]

3.4.2.9.1 What is missing from the Transformer? Before diving deep in the Reformer, let us review what is challenging about the Transformer model.
Although Transformer models yield great results being used on increasingly long sequences — e.g. 11K long text examples in (Liu et al., 2018) — many of such large models can only be trained in large industrial compute platforms and even cannot be fine-tuned on a single GPU even for a single training step due to their memory requirements. For example, the full GPT-2 model consists of roughly 1.5B parameters. The number of parameters in the largest configuration reported in (Shazeer et al., 2018) exceeds 0.5B per layer, while the number of

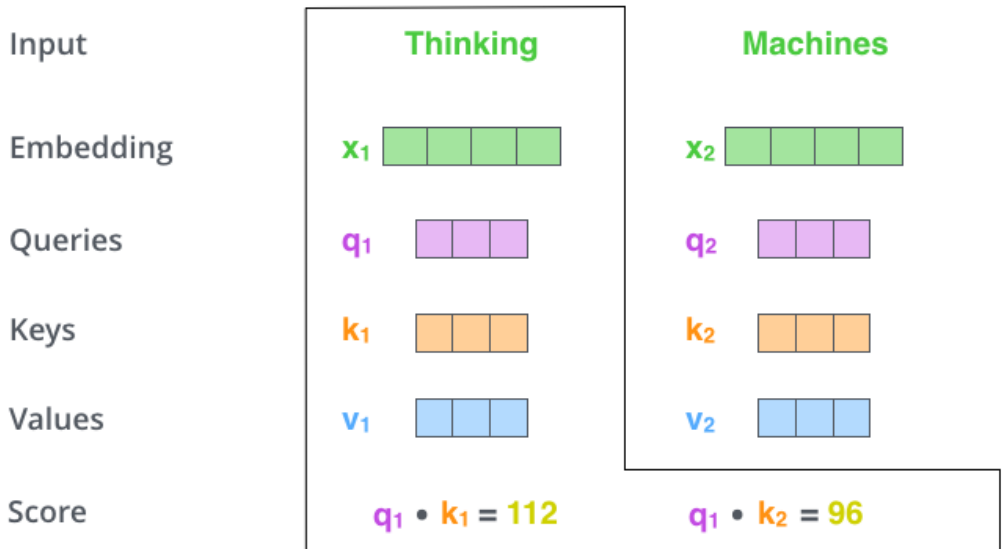


Figure 3.27: Self-Attention - Score calculation.

layers goes up to 64 in (Al-Rfou et al., 2018).
Memory and computation issues include:

- **Attention computation** $\longrightarrow$ Computing attention on sequences of length L is $O(L^2)$ (both time and memory). Imagine what happens if we have a sequence of length 64K.
- **Large number of layers** $\longrightarrow$ A model with N layers consumes N-times larger memory than a single-layer model, as activations in each layer need to be stored for back-propagation.
- **Depth of feed-forward layers** $\longrightarrow$ the depth of intermediate feed-forward layers is often much larger than the depth of attention activations.

The Reformer model addresses the above three main sources of memory consumption in the Transformer and improves upon them in such a way that the Reformer model can handle context windows of up to 1 million words, all on a single accelerator and using only 16GB of memory.

3.4.2.9.2 Optimization techniques In a nutshell, the Reformer model combines two techniques to solve the problems of attention and memory allocation:
locality-sensitive-hashing (LSH) [87] to reduce the complexity of attending over long sequences (from $O(L^2)$ to $O(L \times logL)$), and **reversible residual layers** [88] to more efficiently use the memory available (using reversible residual layers instead of standard residuals enables storing activations only once during the training process instead of N times).

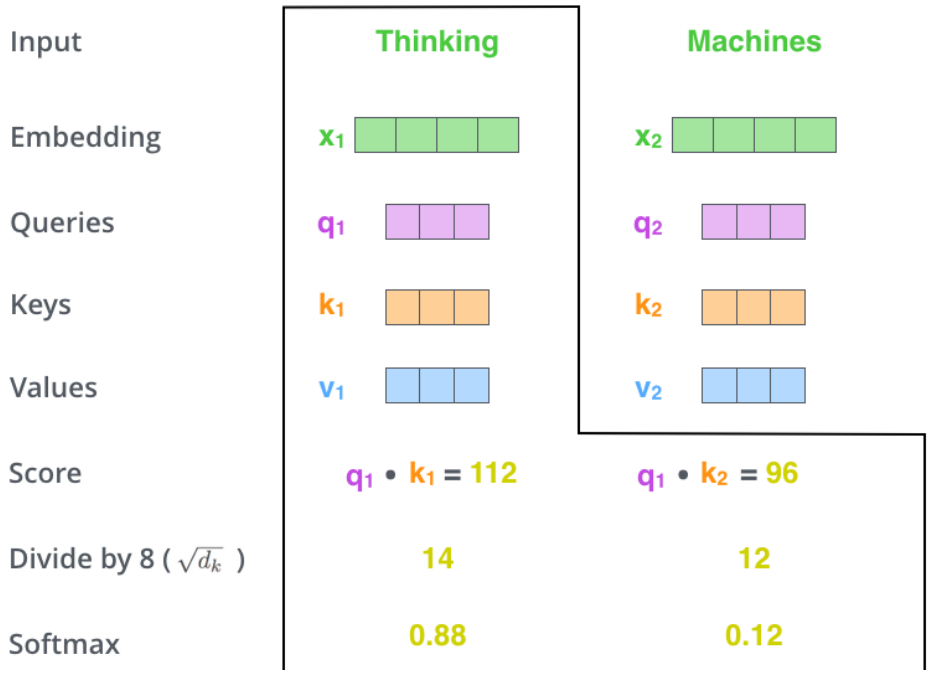


Figure 3.28: Self-Attention - Score calculation with division and softmax.

In addition, a portion of efficiency improvements in the Reformer deal with the 3rd problem, i.e. high dimensional intermediate vectors of the feed-forward layers — that can go up to 4K and higher in dimensions.
This computation improvement is **chunking**: due to the fact that computations in feed-forward layers are independent across positions in a sequence, the computations for the forward and backward passes as well as the reverse computation can be all split into chunks.

3.4.2.9.3 Experimental results The authors conducted experiments on two tasks: the image generation task imagenet64 (with sequences of length 12K) and the text task enwik8 (with sequences of length 64K), and evaluated the effect of reversible Transformer and LSH hashing on the memory, accuracy, and speed.
Reversible Transformer matches baseline: their experiment results showed that the reversible Transformer saves memory without sacrificing accuracy.
LSH attention matches baseline (note that as LSH attention is an approximation of full attention, its accuracy improves as the hash value increases). When the hash value is 8, LSH attention is almost equivalent to full attention.
They also demonstrated that the conventional attention slows down as the sequence length increases, while LSH attention speed remains steady, and it runs on sequences of length 100K at usual speed on 8GB GPUs.
Thus, to sum up: the final Reformer model performed similarly compared to the

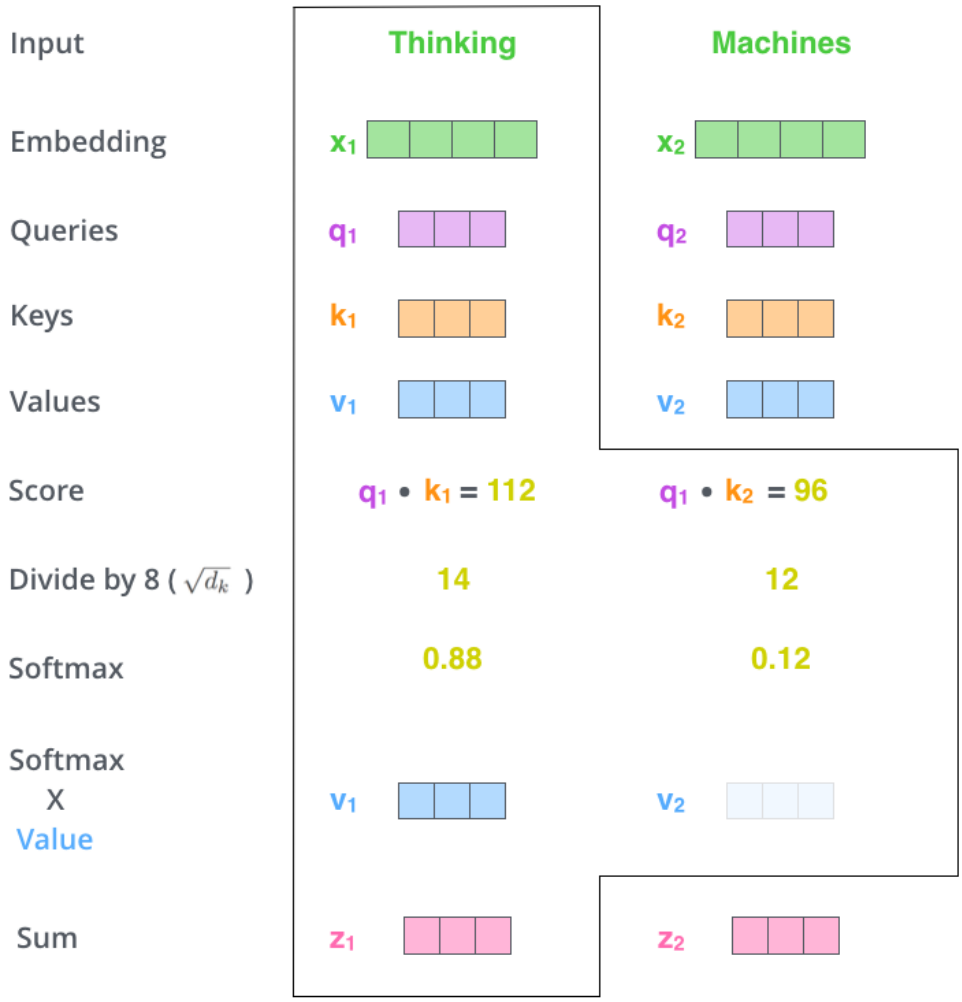


Figure 3.29: Self-Attention - Full calculation process.

Transformer model, but showed higher storage efficiency and faster speed on long sequences.

3.5 Evaluation techniques

The most common way to evaluate the informativeness of automatic summaries is to compare them with human-made model summaries.
Evaluation techniques fall into intrinsic and extrinsic, inter-textual and intra-textual.

3.5.1 Intrinsic and extrinsic evaluation

An intrinsic evaluation tests the summarization system in and of itself while an extrinsic evaluation tests the summarization based on how it affects the completion of some other task. Intrinsic evaluations have assessed mainly the coherence and informativeness of summaries. Extrinsic evaluations, on the other hand, have tested the impact of

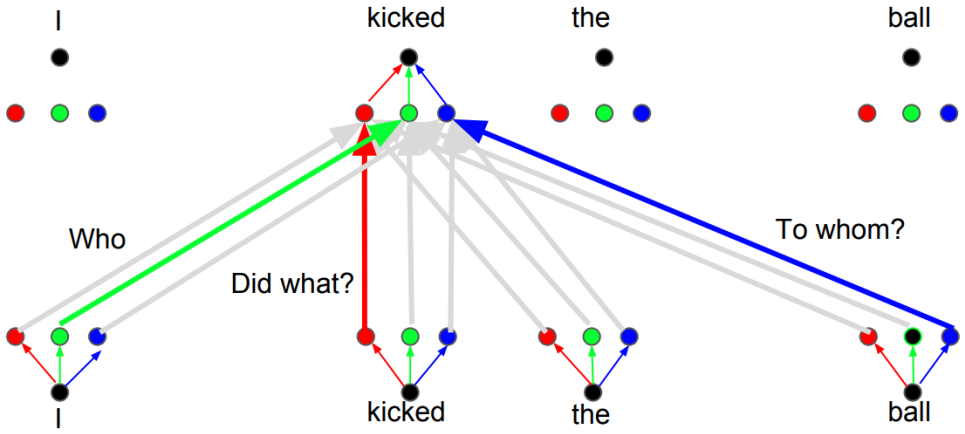


Figure 3.30: Multihead attention mechanism for 3 different questions.

summarization on tasks like relevance assessment, reading comprehension, etc.

3.5.2 Inter-textual and intra-textual

Intra-textual methods assess the output of a specific summarization system, and the inter-textual ones focus on contrastive analysis of outputs of several summarization systems.

Human judgement often has wide variance on what is considered a "good" summary, which means that making the evaluation process automatic is particularly difficult. Manual evaluation can be used, but this is both time and labor-intensive as it requires humans to read not only the summaries but also the source documents. Other issues are those concerning coherence and coverage.

One of the metrics used in NIST's annual Document Understanding Conferences, in which research groups submit their systems for both summarization and translation tasks, is the ROUGE metric (Recall-Oriented Understudy for Gisting Evaluation). It essentially calculates n-gram overlaps between automatically generated summaries and previously-written human summaries. A high level of overlap should indicate a high level of shared concepts between the two summaries. Note that overlap metrics like this are unable to provide any feedback on a summary's coherence. Anaphor resolution remains another problem yet to be fully solved. Similarly, for image summarization, Tschiatschek et al., developed a Visual-ROUGE score which judges the performance of algorithms for image summarization.

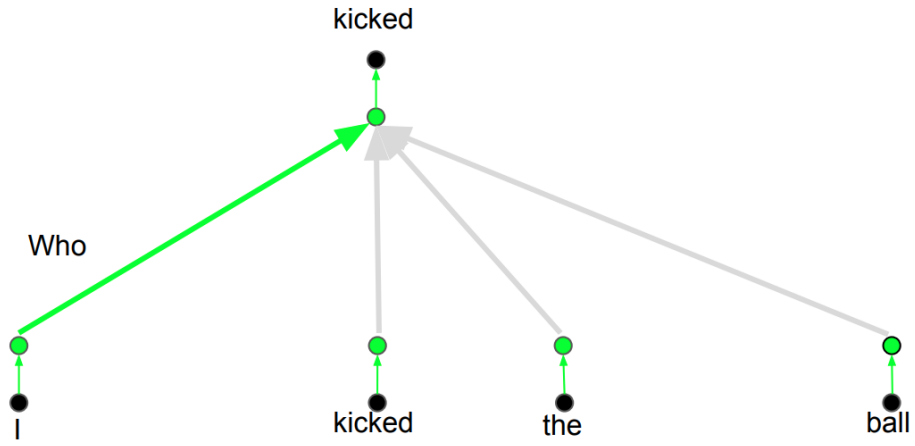


Figure 3.31: Multihead attention mechanism for the question "Who kicked?".

3.5.2.1 ROUGE metric

ROUGE[13] score measures the overlapping between the produced text and the reference one (i.e. an example of good summary usually produced by humans). In our case the reference summary will be the set of keyphrase for each legal case report. For instance, Rouge 1 measures single word overlap between a reference summary and summarized text whereas Rouge 2 measures bi gram overlap between reference summary and the generated one, and so on.
The following five evaluation metrics are available:

- ROUGE-N: Overlap of N-grams between the system and reference summaries.
- ROUGE-L: Longest Common Subsequence (LCS) based statistics that take into account sentence level structure similarity naturally and identifies longest co-occurring in sequence (not necessarily consecutive) n-grams automatically.
- ROUGE-W: Weighted LCS-based statistics that favors consecutive LCSes.
- ROUGE-S: Skip-bigram[89] based co-occurrence statistics. Skip-bigram is any pair of words in their sentence order.
- ROUGE-SU: Skip-bigram plus unigram-based co-occurrence statistics.

For each ROUGE metric, Precision, Recall and F1 scores can be calculated. Below is an example of Precision, Recall and F1 for ROUGE-1:

- ROUGE-1 recall=40% means that 40% of the unigrams (single words) in the reference summary are also present in the generated summary.

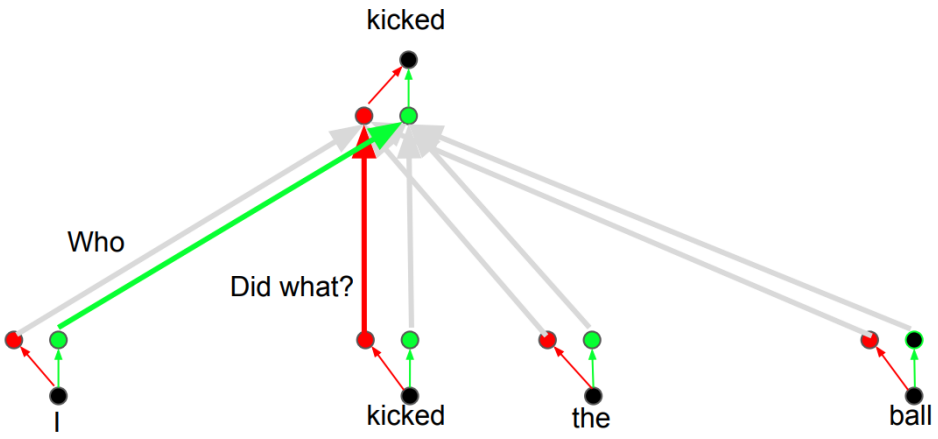


Figure 3.32: Multihead attention mechanism for the questions "Who kicked?" and "Did what?".

- ROUGE-1 precision=40% means that 40% of the unigrams in the generated summary are also present in the reference summary.
- ROUGE-1 F1 is the F1-score calculated using ROUGE-1 recall and ROUGE-1 precision.

Note: since ROUGE score only looks at word overlapping and not readability and consistency of the text, it is not a perfect metric as text with high rouge score can be a badly written or inconsistent summary.

3.5.3 Domain specific versus domain independent summarization techniques

Domain independent summarization techniques generally apply sets of general features which can be used to identify information-rich text segments. Recent research focus has drifted to domain-specific summarization techniques that utilize the available knowledge specific to the domain of text. For example, automatic summarization research on medical text generally attempts to utilize the various sources of codified medical knowledge and ontologies.

3.5.4 Evaluating summaries qualitatively

The main drawback of the evaluation systems existing so far is that we need at least one reference summary, and for some methods more than one, to be able to compare automatic summaries with models. This is a hard and expensive task. Much effort has to be done in

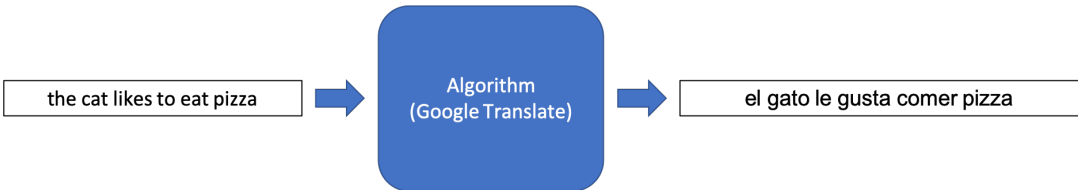
order to have corpus of texts and their corresponding summaries. Furthermore, for some methods, not only do we need to have human-made summaries available for comparison, but also manual annotation has to be performed in some of them (e.g. SCU in the Pyramid Method). In any case, what the evaluation methods need as an input, is a set of summaries to serve as gold standards and a set of automatic summaries. Moreover, they all perform a quantitative evaluation with regard to different similarity metrics.

Chapter 4

Neural Machine Translation

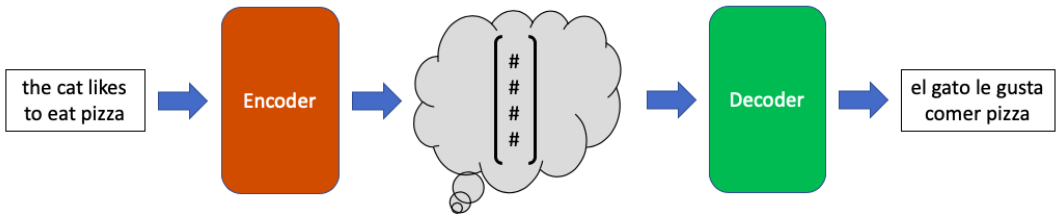
4.0.1 Introduction

- Machine Translation (MT) is a subfield of computational linguistics that is focused on translating text from one language to another.
- With the power of deep learning, Neural Machine Translation (NMT) has arisen as the most powerful algorithm to perform this task.
- While Google Translate is the leading industry example of NMT, tech companies all over the globe are going all in on NMT.
- This state-of-the-art algorithm is an application of deep learning in which massive datasets of translated sentences are used to train a model capable of translating between any two languages.



4.0.2 Neural Machine Translation Architecture

With the vast amount of research in recent years, there are several variations of NMT currently being investigated and deployed in the industry.
One of the older and more established versions of NMT is the Encoder-Decoder structure. This architecture is composed of two recurrent neural networks (RNNs) used together in tandem to create a translation model.
And when coupled with the power of attention mechanisms, this architecture can achieve impressive results.



4.0.3 Encoder-Decoder limitation and improvements

Although effective, the Encoder-Decoder architecture has problems with long sequences of text to be translated.
The problem stems from the fixed-length internal representation that must be used to decode each word in the output sequence.
The solution is the use of an attention mechanism that allows the model to learn where to place attention on the input sequence as each word of the output sequence is decoded.
Using a fixed-sized representation to capture all the semantic details of a very long sentence is very difficult.
A more efficient approach, however, is to read the whole sentence or paragraph, then to produce the translated words one at a time each time focusing on a different part of he input sentence to gather the semantic details required to produce the next output word.

4.0.4 State-of-art in NMT: the Transformer

The encoder-decoder neural network architecture with attention is currently the state-of-the-art on some benchmark problems for machine translation.
Currently the Transformer [33] which represents the state-of-art technique in Automatic Translation, uses feed-forward neural networks instead recurrent ones.
Recurrent models typically factor computation along the symbol positions of the input and output sequences.
Aligning the positions to steps in computation time, they generate a sequence of hidden states h_t, as a function of the previous hidden state h_{t-1} and the input for position t. This inherently sequential nature precludes parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching across examples.
Attention mechanisms have become an integral part of compelling sequence modeling and transduction models in various tasks, allowing modeling of dependencies without regard to their distance in the input or output sequences. In all but a few cases, however, such attention mechanisms are used in conjunction with a recurrent network.
The Transformer model architecture eschews recurrence and instead relying entirely on an

attention mechanism to draw global dependencies between input and output.
The Transformer allows for significantly more parallelization and can reach a new state of the art in translation quality.
The Encoder-Decoder architecture is used in the heart of the Google Neural Machine Translation system, or GNMT, used in their Google Translate service.
Although effective, the neural machine translation systems still suffer some issues such as scaling to larger vocabularies of words and the slow speed of training the models.

Chapter 5

Selected Technologies

5.1 Python

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects. [90]

5.1.1 Main features

Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Python is often described as a "batteries included" language due to its comprehensive standard library.Python interpreters are available for many operating systems.

5.1.2 Brief history

Python was conceived in the late 1980s as a successor to the ABC language. Python 2.0, released in 2000, introduced features like list comprehensions and a garbage collection system capable of collecting reference cycles. Python 3.0, released in 2008, was a major revision of the language that is not completely backward-compatible, and much Python 2 code does not run unmodified on Python 3. Due to concern about the amount of code written for Python 2, support for Python 2.7 (the last release in the 2.x series) was extended to 2020. Language developer Guido van Rossum shouldered sole responsibility for the project until July 2018 but now shares his leadership as a member of a five-person steering council. A global community of programmers develops and maintains CPython, an open source reference implementation. A non-profit organization, the Python Software Foundation, manages and directs resources for Python and CPython development.

5.2 Pandas

In computer programming, Pandas [91] [92] is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series. It is free software released under the three-clause BSD license. The name is derived from the term "panel data", an econometrics term for data sets that include observations over multiple time periods for the same individuals.

5.2.1 Library features

Main library feature include:

- DataFrame object for data manipulation with integrated indexing.
- Tools for reading and writing data between in-memory data structures and different file formats.
- Data alignment and integrated handling of missing data.
- Reshaping and pivoting of data sets.
- Label-based slicing, fancy indexing, and subsetting of large data sets.
- Data structure column insertion and deletion.
- Group by engine allowing split-apply-combine operations on data sets.
- Data set merging and joining.
- Hierarchical axis indexing to work with high-dimensional data in a lower-dimensional data structure.
- Time series-functionality: Date range generation and frequency conversion, moving window statistics, moving window linear regressions, date shifting and lagging.
- Provides data filtration.
- The library is highly optimized for performance, with critical code paths written in Cython or C.

5.2.2 Dataframes

Pandas is mainly used for machine learning in form of dataframes. Pandas allow importing data of various file formats such as csv, excel etc. Pandas allows various data manipulation operations such as groupby, join, merge, melt, concatenation as well as data cleaning features such as filling, replacing or imputing null values.

5.3 Flair NLP Library

Flair [45][93] is a very simple framework for state-of-the-art NLP which has been developed by Zalando Research.

5.3.1 Library features

Main Flair library features include:

- **powerful NLP library**. Flair allows you to apply our state-of-the-art natural language processing (NLP) models to your text, such as named entity recognition (NER), part-of-speech tagging (PoS), sense disambiguation and classification;
- **multilingual**. Thanks to the Flair community, we support a rapidly growing number of languages. We also now include 'one model, many languages' taggers, i.e. single models that predict PoS or NER tags for input text in various languages;
- **text embedding library**. Flair has simple interfaces that allow you to use and combine different word and document embeddings, including our proposed Flair embeddings, BERT embeddings and ELMo embeddings;
- **PyTorch NLP framework**. Our framework builds directly on PyTorch, making it easy to train your own models and experiment with new approaches using Flair embeddings and classes.

5.4 Keras

Keras [94] is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.
Use Keras if you need a deep learning library that:

- Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility).
- Supports both convolutional networks and recurrent networks, as well as combinations of the two.
- Runs seamlessly on CPU and GPU.

5.4.1 Library features

Keras [95] contains numerous implementations of commonly used neural-network building blocks such as layers, objectives, activation functions, optimizers, and a host of tools to make working with image and text data easier. The code is hosted on GitHub, and community support forums include the GitHub issues page, and a Slack channel.
In addition to standard neural networks, Keras has support for convolutional and recurrent neural networks. It supports other common utility layers like dropout, batch normalization, and pooling.
Keras allows users to productize deep models on smartphones (iOS and Android), on the web, or on the Java Virtual Machine. It also allows use of distributed training of deep-learning models on clusters of Graphics Processing Units (GPU) and Tensor processing units (TPU).

5.5 PyTorch

PyTorch [96][97] is an open source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing. It is primarily developed by Facebook's artificial intelligence research group. It is free and open-source software released under the Modified BSD license. Although the Python interface is more polished and the primary focus of development, PyTorch also has a C++ frontend. Furthermore, Uber's Pyro probabilistic programming language software uses PyTorch as a backend.

5.5.1 Library features

PyTorch provides two high-level features:

- Tensor computing (like NumPy) with strong acceleration via graphics processing units (GPU);
- Deep neural networks built on a tape-based autodiff system.

Tensors, while from mathematics, are different in programming, where they can be treated as multidimensional array data structures (arrays). Tensors in PyTorch are similar to NumPy arrays, but can also be operated on a CUDA-capable Nvidia GPU. PyTorch supports various types of tensors.

5.6 Transformers

Transformers [98] (formerly known as pytorch-transformers and pytorch-pretrained-bert) provides general-purpose architectures (BERT, GPT-2, RoBERTa, XLM, DistilBert,

XLNet...) for Natural Language Understanding (NLU) and Natural Language Generation (NLG) with over 32+ pretrained models in 100+ languages and deep interoperability between TensorFlow 2.0 and PyTorch.

5.6.1 Library features

Main library features include:

- As easy to use as pytorch-transformers
- As powerful and concise as Keras
- High performance on NLU and NLG tasks
- Low barrier to entry for educators and practitioners
- State-of-the-art NLP for everyone:
  - Deep learning researchers
  - Hands-on practitioners
  - AI/ML/NLP teachers and educators
- Lower compute costs, smaller carbon footprint:
  - Researchers can share trained models instead of always retraining
  - Practitioners can reduce compute time and production costs
  - 8 architectures with over 30 pretrained models, some in more than 100 languages
- Choose the right framework for every part of a model's lifetime:
  - Train state-of-the-art models in 3 lines of code
  - Deep interoperability between TensorFlow 2.0 and PyTorch models
  - Move a single model between TF2.0/PyTorch frameworks at will
  - Seamlessly pick the right framework for training, evaluation, production

Chapter 6

Project CRISP-DM Phases

The description of the project development has been divided into the phases described by the CRISP-DM [99] methodology.
CRISP-DM stands for cross-industry process for data mining. The CRISP-DM methodology provides a structured approach to planning a data mining project. It is a robust and well-proven methodology.
This model is an idealised sequence of events. In practice many of the tasks can be performed in a different order and it will often be necessary to backtrack to previous tasks and repeat certain actions. The model does not try to capture all possible routes through the data mining process. [100]

6.1 Business Understanding

The first stage of the CRISP-DM process is to understand what you want to accomplish from a business perspective. The goal of this stage of the process is to uncover important factors that could influence the outcome of the project.

6.1.1 Business objectives

Nowadays lawyers and judges have to analyse many legal sentences during the performance of their duties. This task is really time-consuming since it involves reading and summarisation processes.
Legal maxims are useful to understand the essential elements a legal sentence, but unluckily these latter ones are not available in most cases.
Thus, in order to save time, it would helpful to have a system which is able to automatically generate a summary of a given legal text.
This leads to the main objective of this project: the automatic generation of coherent summaries related to legal sentences written in Italian.

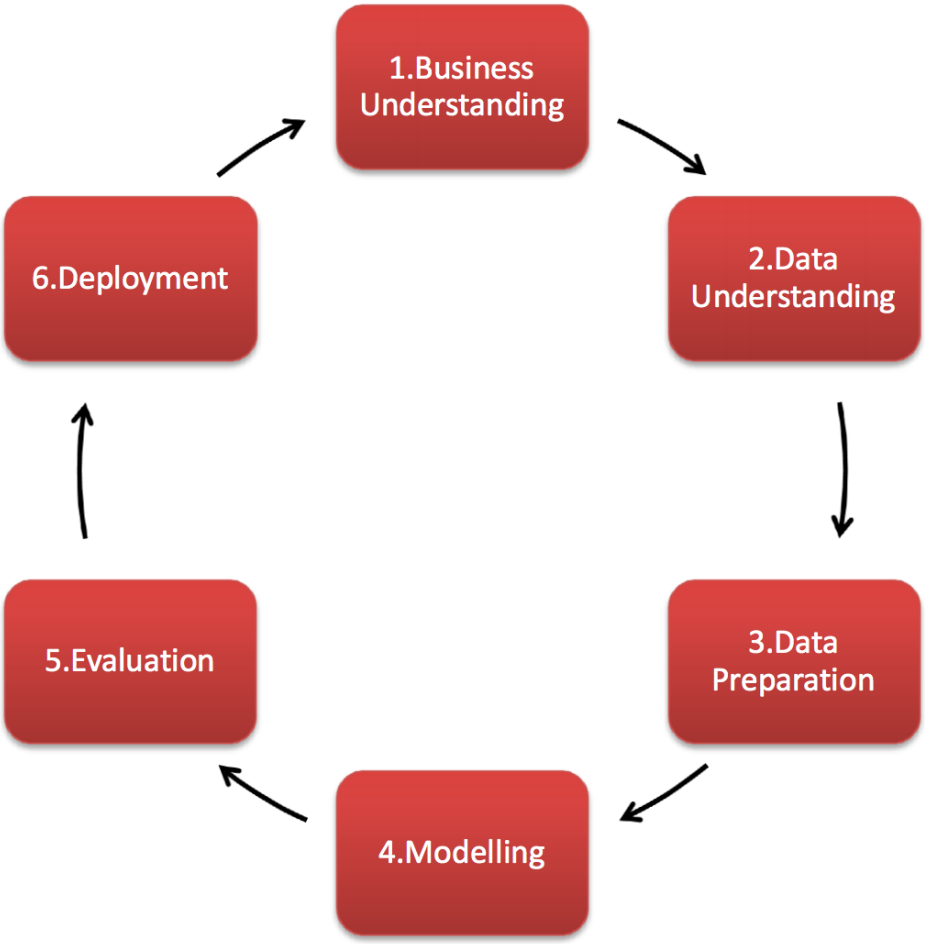


Figure 6.1: Phases of the CRISP-DM process.

6.1.2 Inventory of resources

6.1.2.1 Computational resources

Computational resources which have been used for this project include:

- Asus Vivobook Pro N580VD Laptop (i7 7700-HQ, Nvidia GeForce GTX 1050)
- Google Colab Notebook with GPU Support
- 2 Remote Desktop Computers accessed via SSH which provides:
  1. a quad-core CPU and Nvidia Titax Xp GPU
  2. a quad-core CPU and 2 Nvidia Titax Xp GPUs

6.1.2.2 Data resources

- Legal Case Reports Data Set [101] containing case reports written in English
- Italian Legal Case Reports

6.1.3 Text mining goals: Extractive and Abstractive Summarisation

Data Mining goal of the extractive summarisation is twofold:

- produce summaries which maximize the main ROUGE metrics such as ROUGE-1, ROUGE-l, etc;
- maximize the F1 score of the binary classifier which selects the keyphrases.

The Data Mining goal of the abstractive summarisation will be to maximize the main ROUGE metrics such as ROUGE-1, ROUGE-l, etc.

6.1.4 Business success criteria

- Both legal texts and generated summaries have to be also in languages other than English including Italian.
- Legal experts will assert the generated summaries are reasonably coherent and correct after reading them.

6.1.5 Project Plan

Analysis work starts from the Australian Case Report Dataset, then the summaries generation will switch to the italian case reports. In order to produce italian summaries they have been used pretrained models and transfer learning techniques since the number of labelled case reports written in Italian is not enough to train a model from scratch.

6.1.5.1 Extractive Summarisation - Single Language

2 in-domain experiment types have been conducted:

- 2 extractive summarisations of the "Australian Legal Case Report Dataset" each one on a data sampling of different size
  - Training our custom neural network with 70 and 700 Australian reports (written in English)
  - Testing on respectively 30 and 300 Australian reports
- The two same experiments mentioned above but using the "Australian Legal Case Report Dataset" translated into several languages via Google Translate APIs
  - Training our custom neural network with 70 and 700 reports written in a language other than English
  - Testing on respectively 30 and 300 reports in the same language (other than English) of texts used in the training process

6.1.5.2 Extractive Summarisation - Cross-Language

- Fine-tuning technique has been used as Transfer Learning approach.
- 4 main experiments have been conducted:
  - Testing of the extractive model trained on 70 English, on 70 Italian reports
  - Fine-tuning of the extractive model trained on 70 English reports, with 70 Italian reports and testing of the resulting model on 30 Italian reports
  - Testing of the extractive model trained on 700 English, on 60 Italian reports
  - Fine-tuning of the extractive model trained on 700 English reports, with 140 Italian reports and testing the resulting model on 60 Italian reports
- Table 6.5 and 6.6 show the results achieved by performing the extractive summarisation task in the cross-language scenario.
- Our models show a good generalization capability across different language domains.
- In both fine-tuning scenarios, performances have been boosted with respect to the other two experiments.

6.1.5.3 Abstractive Summarisation - Cross Domain

2 main experiment types have been conducted:

- Two abstractive summarisations of the ”Australian Legal Case Report Dataset” each one on a data sampling of different size
  - Fine-tuning our chosen generative model (GPT-2 Transformer [1] pre-trained to predict the next word in 40GB of Internet text) with 70 and 700 Australian reports
  - Generate 30 and 300 report summaries (written in English), respectively
  - Testing the generated report summaries on 30 and 300 Australian reports, respectively
- The two same experiments mentioned above but using the ”Australian Legal Case Report Dataset” translated into several languages via Google Translate APIs
  - Fine-tuning our chosen generative model (GPT-2 Transformer [1] pre-trained to predict the next word in 40GB of Internet text) with 70 and 700 reports written in a language other than English
  - Generate 30 and 300 report summaries (written in the language of the texts used during the fine-tuning process), respectively

- Testing the generated report summaries on 30 and 300 reports (translated into the language of the texts used during the fine-tuning process), respectively

6.1.5.4 FactCC model evaluation on Legal Case Reports

FactCC [3] is a classification model from a recent work of SalesForce Research department. Currently used metrics for assessing summarization algorithms do not account for whether summaries are factually consistent with source documents. They proposed a weakly-supervised, model-based approach for verifying factual consistency and identifying conflicts between source documents and a generated summary.
Training data is generated by applying a series of rule-based transformations to the sentences of source documents. The factual consistency model is then trained jointly for three tasks:

1. identify whether sentences remain factually consistent after transformation
2. extract a span in the source documents to support the consistency prediction
3. extract a span in the summary sentence that is inconsistent if one exists.

FactCC high-level functioning can be seen in Figure 6.2.
Our work aims to train their model on the "Australian Legal Case Report Dataset" (which will be applied transformations to) and then evaluate the model capability to distinguish coherent summaries from inconsistent summaries.
They will be conducted two main experiments: BERT (base uncased) fine-tuning using 100 and 1000 original legal case reports, respectively.

6.2 Data Understanding

The second stage of the CRISP-DM process requires you to acquire the data listed in the project resources. This initial collection includes data loading, if this is necessary for data understanding.

6.2.1 Data description

Australian Legal Case Reports Data Set.
A textual corpus of 4000 legal cases for automatic summarization and citation analysis.
For each document the authors collected catchphrases (catchphrases are meant to present the important legal points of a document with respect of identifying precedents), citations sentences, citation catchphrases and citation classes.
This dataset contains Australian legal cases from the Federal Court of Australia (FCA). The cases were downloaded from AustLII. They included all cases from the year

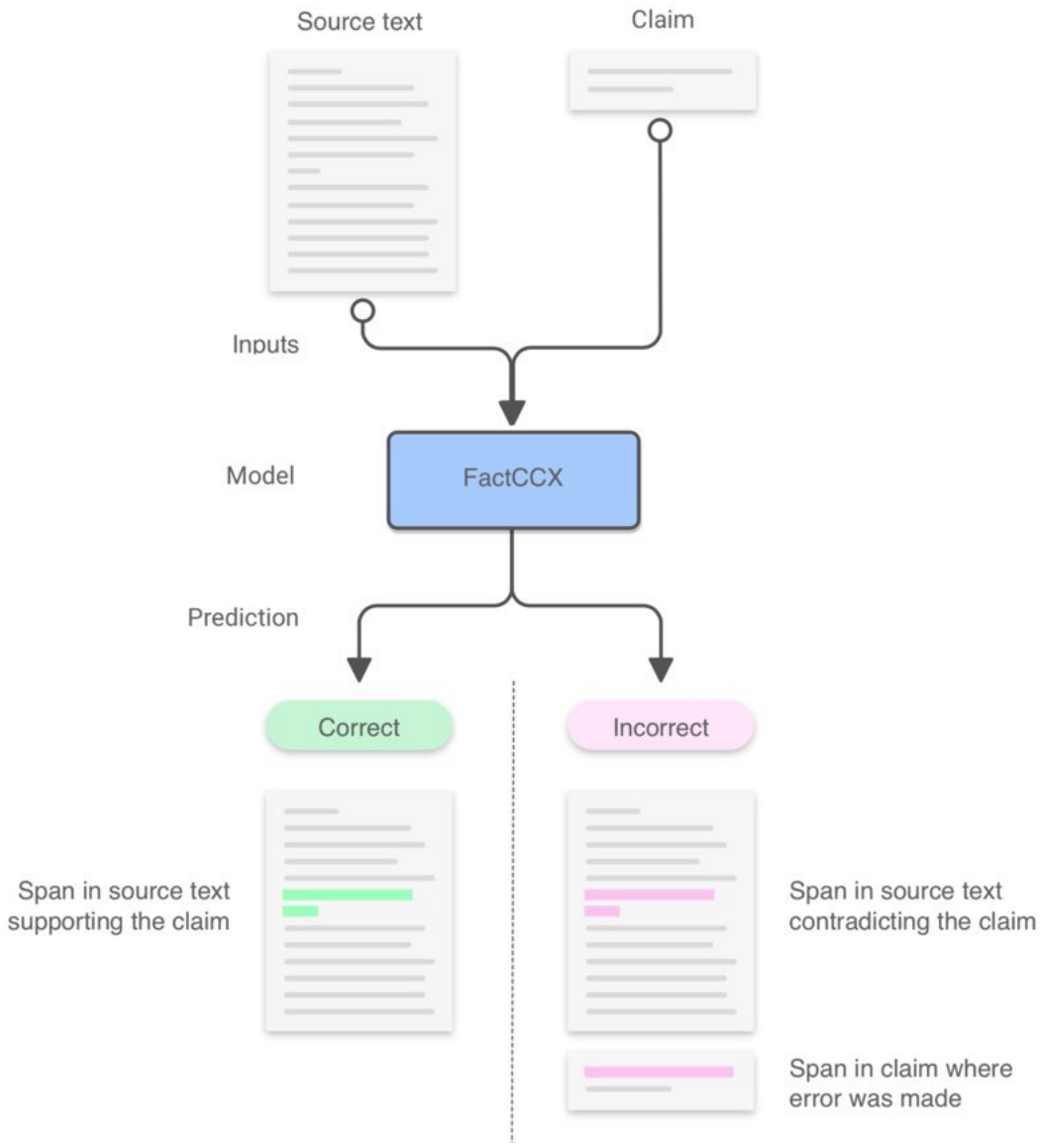


Figure 6.2: High-level functioning of FactCC model

2006,2007,2008 and 2009. The authors built it to experiment with automatic summarization and citation analysis [102]. For each document they collected catchphrases, citations sentences, citation catchphrases, and citation classes.

Catchphrases are found in the document: they used the catchphrases as gold standard for their summarization experiments.

Citation sentences are found in later cases that cite the present case: they use citation sentences for summarization.

Citation catchphrases are the catchphrases (where available) of both later cases that cite the present case, and older cases cited by the present case.

Citation classes are indicated in the document, and indicate the type of treatment given to the cases cited by the present case.

6.2.2 Data exploration

Australian Legal Case Reports Data Set.
This dataset is structured in 3 directories:

- **fulltext**: contains the full text and the catchphrases of all the cases from the FCA.
  Every document (<case>) contains:
  - <name> : name of the case
  - <AustLII> : link to the austlii page from where the document was taken
  - <catchphrases> : contains a list of <catchphrase> elements
    * <catchphrase> : a catchphrase for the case, with an id attribute
  - <sentences> : contains a list of <sentence> elements
    * <sentence> : a sentence with id attribute
- **citations_summ**: contains citations element for each case.
  Fields:
  - <name> : name of the case
  - <AustLII> : link to the austlii page from where the document was taken
  - <citphrases> : contains a list of <citphrase> elements
    * <citphrase> : a citphrase for the case, this is a catchphrase from a case which is cited or cite the current one.
      Attributes:
      · id
      · type (cited or citing)
      · from (the case from where the catchphrase is taken)
  - <citances> : contains a list of <citance> elements
    * <citance> : a citance for the case, this is a sentence from a later case that mention the current case.
      Attributes:
      · id
      · from (the case from where the citance is taken)
  - <legistitles> : contains a list of <title> elements
    * <title> : title of a piece of legislation cited by the current case (can be an act or a specific section).

- **citations_class**: contains for each case a list of labelled citations.
  Fields:
  - <name> : name of the case
  - <AustLII> : link to the austlii page from where the document was taken
  - <citations> : contains a list of <citation> elements
    * <citation> : a citation to an older case, it has an id attribute and contains the following elements:
      · <class> : the class of the citation as indicated on the document (considered, followed, cited, applied, notfollowed, referred to, etc.)
      · <tocase> : the name of the case which is cited
      · <AustLII> : the link to the document of the case which is cited
      · <text> : paragraphs in the cited case where the current case is mentioned

6.2.3 Data quality report

No missing values have been found in fulltext and citation_class directory files, while some values are missing in citation_summ documents.
XML files contain many HTML entity characters. These latter ones made XML parsing invalid since "&" characters would indicate entities of XML types (and not HTML ones. These characters had to be removed.

6.3 Data Preparation

6.3.1 Data selection

Australian Legal Case Reports Data Set.
Data used to perform the analysis have been selected from the fulltext directory.
For each legal case (i.e. a XML file) <name>, <catchphrase> and <sentence> have been used.

6.3.2 Data cleaning

Australian Legal Case Reports Data Set.
HTML Special Entities had to be removed from the text retrieved from XML files in order to parse it correctly. This task has been accomplished defining the function shown in Figure 6.3. This function substitutes HTML entity characters with the corresponding textual representation.

```
def preprocess_xml_file(xml_file_path):
    with open(xml_file_path, "r", encoding="utf-8") as f:
        file_content = f.read()
        return re.sub(r'&amp;', 'and', file_content) \
            .replace('&#8226;', '-') \
            .replace('&#8356;', '£') \
            .replace('&eacute;', 'é') \
            .replace('&Eacute;', 'É') \
            .replace('&agrave;', 'à') \
            .replace('&reg;', '') \
            .replace('&gt;', '>') \
            .replace(' ', ' ') \
            .replace('"id=', 'id="') \
            .replace('&ouml;', 'ö') \
            .replace('&auml;', 'ä') \
            .replace('&aacute;', 'á') \
            .replace('&aelig;', 'æ') \
            .replace('&acirc;', 'â') \
            .replace('&iuml;', 'ï') \
            .replace('&egrave;', 'È') \
            .replace('&ccedil;', 'ç') \
            .replace('&ecirc;', 'ê') \
            .replace('<?xml version="1.0"?>', '<?xml version="1.0" encoding="utf-8"?>') \
            .replace('type=cited', 'type="cited"') \
            .replace('type=citing', 'type="citing"')
```

Figure 6.3: Defined function used to remove HTML Entities characters from the dataset.

6.3.3 Required data construction

Target variable creation.

In order to create the target variable (i.e. the feature representing the class) of our extractive summarisation experiments, a label for each legal phrase is needed. This latter one indicates whether a phrase should be included in a case report summary or not. Thus, the class variable will be binary. Since this information is not directly specified in the metadata, it has been generated using the following approach: for each phrase of a legal case it is checked whether at least one of the catchphrases (catchphrases are meant to present the important legal points of a document with respect of identifying precedents) for that legal case is included in the current legal phrase examined. If this condition is true then the label of that phrase will be `True`, otherwise it will be `False`.

FactCC model evaluation on Australian Legal Case Report Dataset.

The data generation scripts provided by the authors will be used. Such scripts generate positive and negative examples from a jsonl file. Negative examples are created by applying some syntactic transformations to the original texts.

Sentence embeddings creation.

In the extractive summarisation scenario the word/sentence embeddings have been

generated by using the Flair NLP library. This latter let us choose among different embedding methods, and generating easily tensors representing words/sentences from the relative string.
Instead, in the abstractive summarisation scenario the embedding process is integrated in the Transformer architecture. Sequences of words are transformed into numeric vectors by the Tokenizer object of the relative Transformer version.

6.3.4 Data integration

The legal phrases of each case reports have been added to a common dataframe. So each instance of this latter structure will represent a phrase.

6.3.5 Data balancing and shuffling

The instances contained in the dataframe have been balanced by class (represented by `is_catchphrase` attribute).
Afterwards, data instances have been shuffled by groups of phrases from the same legal case report. This has been performed in order to avoid the situation where the model will classify phrases from legal case reports already seen during training time.

6.4 Modeling

6.4.1 Modeling technique selection

In the following subsections the modeling techniques used will be described. These latter techniques will be divided in extractive and abstractive ones.
Extraction-based summarisation.
The main mining techniques used in this case is represented by neural networks. In order to generate contextualized word embeddings, BERT-Base-Multilingual pre-trained model has been applied. This choice has two main reasons: BERT has passed the state of the art in various NLP tasks; the multilingual model allowed us to overcome the problem of the absence of an Italian legal case reports dataset. Sentence embeddings have been produced by carrying out the mean of the word embeddings related to words within the same sentence. For the binary classification, it has been built a custom model which is basically composed by:

1. a 1D CNN [59] layer
2. a bidirectional GRU [57] layer
3. 4 fully-connected layers

Abstraction-based summarisation.
GPT-2 Transformer model has been used in order to generate new sentences of legal case reports.

6.4.2 Test design generation

Extraction-based summarisation.
The model will be tested by evaluating the F1 and ROUGE score. F1 metric will be calculated since both training set and test set are not perfectly class-balanced since after the class-balancing operation, data is grouped by case report groups and shuffled (this will be also done in order to simulate a production environment where an entire new legal case report is passed as input to our classifier). Generally speaking, when you perform a summarisation task this makes more sense than considering only accuracy as you have to take into account recall (the ratio of relevant phrases retrieved by the model out of all relevant phrases in the dataset) and precision (the ratio of relevant phrases retrieved out of all the phrases in the produced summary) metrics.
Abstraction-based summarisation. In the abstraction-based summarisation scenario the evaluation metrics will be the ROUGE score and a recent approach named FactCC [3] to evaluate the factual consistency of the summaries with respect to the report original texts which are related to.
FactCC model evaluation on Australian Legal Case Report Dataset.
In the FactCC model evaluation on Australian Legal Case Report Dataset, F1 Score and balanced accuracy metrics will be calculated.

6.4.3 Model building

Extraction-based neural network architecture.
The Neural Network used in the extractive summarisation process is basically a mixed architecture composed by a CNN [59] followed by GRU [57] layer. This latter is connected by a group of 4 fully-connected layers.
This model has been trained using 70% of the dataset sampled in each experiment.
Below is the model architecture description provided by Keras function `model.summary()`:

```
    _________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_1 (Conv1D)            (None, 1, 1024)           3146752
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 1, 1024)           0
_________________________________________________________________
dropout_1 (Dropout)          (None, 1, 1024)           0
_________________________________________________________________
```

```
bidirectional_1 (Bidirection (None, 1, 2048)           12589056
_________________________________________________________________
global_max_pooling1d_1 (Glob (None, 2048)              0
_________________________________________________________________
dropout_2 (Dropout)          (None, 2048)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               1049088
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               65664
_________________________________________________________________
dropout_4 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 32)                4128
_________________________________________________________________
dropout_5 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_4 (Dense)              (None, 2)                 66
=================================================================
Total params: 16,854,754
Trainable params: 16,854,754
Non-trainable params: 0
_________________________________________________________________
```

Algorithm 6.1: Architecture of the neural network used in the extractive summarisation process.

Model parameters.

- `nl1` (Number of Dense layers in Fully-Connected group #1) = 1
- `nl2` (Number of Dense layers in Fully-Connected group #2) = 1
- `nl3` (Number of Dense layers in Fully-Connected group #3) = 1
- `nn1` (Number of neurons in GRU layer) = 1024
- `nn2` (Number of neurons in Fully-Connected group layers #1) = 512
- `nn3` (Number of neurons in Fully-Connected group layers #2) = 128
- `nn4` (Number of neurons in Fully-Connected group layers #2) = 32
- `Learning Rate` = 0.0001
- `Decay` = 1e-6
- `L1` = 0
- `L2` = 0.001
- `Activation function` = Relu

- `Dropout` = 0.1
- `CNN filters` = 1024
- `Use Piscaglia-Moro layer` = False
- `Adam beta_1` = 0.9
- `Adam beta_2` = 0.999
- `kernel_size` = 1
- `pool_size` = 1

Abstraction-based neural network.
The neural network used in abstraction-based summarisation is basically a GPT2-Transformer architecture which has been fine-tuned providing all the legal sentences contained in the training data (70%).
Training data (Figure 6.4): each instance of our GPT-2 fine-tuning data is sequentially composed by the **input text**, a `<summarize>` **tag** and the **input text summary**. In our case, the input text will be the original text of each legal case report while the summary will be the set of the sentences labeled as relevant for each report.

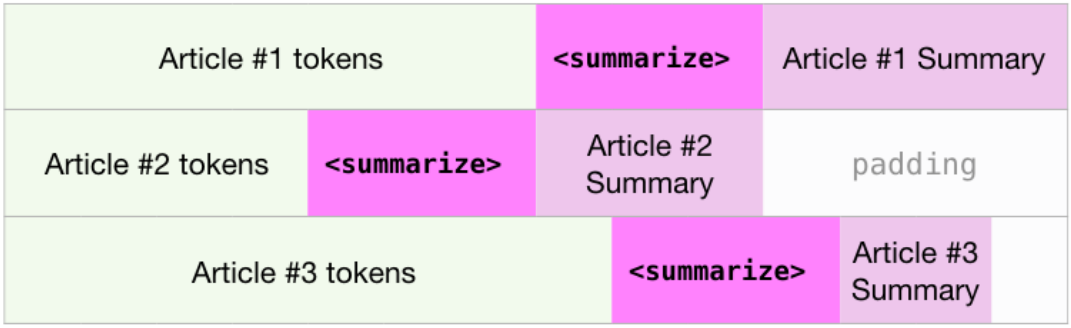


Figure 6.4: Abstractive summarisation with GPT-2 - Training data

Inference phase (Figure 6.5): each test data instance is composed by the **original text** of a new legal report followed by the `<summarize>` **tag**.
Below are the main parameters used in fine-tuning process:

- `model_type` = "gpt2"
- `model_name_or_path` = "gpt2"
- `mlm` = False
- `mlm_probability` = 0.15

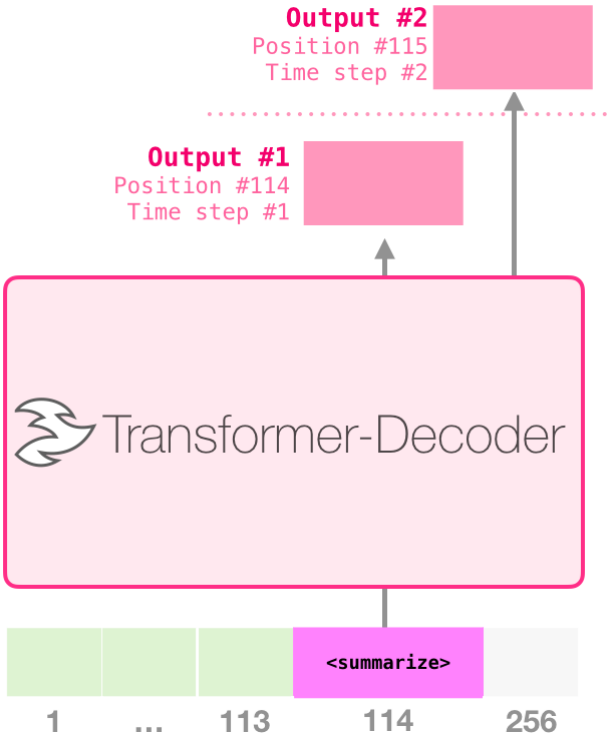


Figure 6.5: Abstractive summarisation with GPT-2 - Inference phase

- input `block_size` = 512
- `do_train` = True
- `do_eval` = False
- `evaluate_during_training` = False
- `do_lower_case` = False
- `per_gpu_train_batch_size` = 2
- `per_gpu_eval_batch_size` = 2
- `gradient_accumulation_steps` = 1
- `learning_rate` = 5×10^{-5}
- `weight_decay` = 0.0
- `adam_epsilon` = 1×10^{-8}
- `num_train_epochs` = 4.0
- `warmup_steps` = 0

- `no_cuda` = False
- `seed` = 42

6.4.4 Model assessment

In the following subsections they will be reported the main results of the summarisations processes performed. All the analysis have been performed both on a dataset of size about 100 and 1000 (some legal case reports have been truncated since not encoded as UTF-8 strings). It has been generated one table for:

- each language which the dataset has been translated to;
- each summarisation technique adopted.

6.4.4.1 Extractive Summarisation

Embedding methods Assessment.
In order to evaluate the numerous embedding methods among those proposed in the literature, several extractive summarisation experiments were carried out on Australian sentences. The results are shown in Table 6.1.

Extractive Summarisation - Embedding Methods Assessment (F1 Score)		
Examples #	*100*	*1000*
Word2Vec	67.10 %	72.69 %
GloVe	64.93 %	73.31 %
ELMo	74.83 %	75.89 %
BERT-Base-Multilingual	70.41 %	**74.22** %
BERT-Large	70.92 %	**78.33** %

Table 6.1: Extractive Summarisation - In-Domain - Embedding Methods Assessment

BERT-large embedding model achieved the best F1 Score even though BERT-Base-Multilingual one presents the most similar results and it has been chosen in order to tackle the absence of labelled datasets in languages other than English.
In-Domain Assessment.
The results of the extractive summarisation processes performed on the original Australian Case Report Dataset are shown in Table 6.2. As one could expect, the ROUGE scores obtained in the experiments with 1000 legal case reports are the highest.
In-Domain Translation Assessment.
Table 6.3 shows the results achieved by the extractive summarisation performed on 100 translated legal case reports. The experiment with the Spanish dataset achieves the best results in almost all metrics except for the F1 Score percentage where the experiment with the Danish dataset got the first place.

	Legal Case Report #					
	≈ 100			≈ 1000		
F1 Score (%)	68.33			**70.51**		
rouge-1	P: 42.01	R: 47.82	F1: 43.18	P: 47.99	R: 54.35	F1: **49.51**
rouge-2:	P: 26.42	R: 30.04	F1: 26.94	P: 32.17	R: 35.57	F1: **32.91**
rouge-3	P: 23.24	R: 26.57	F1: 23.64	P: 28.79	R: 31.60	F1: **29.39**
rouge-4	P: 22.17	R: 25.44	F1: 22.53	P: 27.73	R: 30.44	F1: **28.28**
rouge-l	P: 44.62	R: 50.09	F1: 45.89	P: 49.79	R: 55.33	F1: **51.26**
rouge-w1.2	P: 31.02	R: 17.85	F1: 21.26	P: 34.91	R: 19.90	F1: **24.20**

Table 6.2: Australian Legal Case Reports - Extractive Summarisation assessment

	Legal Case Report # 100						
	F1 Scores						
	%	R-1	R-2	R-3	R-4	R-l	R-w1.2
Australian	68.33	43.18	26.94	23.64	22.53	45.89	21.26
Italian	65.20	61.29	50.43	48.78	47.87	63.91	32.36
German	66.64	47.10	33.79	31.43	30.40	50.00	23.33
Spanish	69.68	**77.73**	**69.60**	**67.48**	**66.28**	**79.86**	**40.78**
Danish	**69.78**	60.69	50.15	48.00	46.95	63.44	32.16
French	67.97	59.85	46.82	44.11	42.82	61.36	29.34

Table 6.3: Extractive Summarisation assessment - 100 translated reports

Instead, Table 6.4 shows the results achieved by the extractive summarisation performed on 1000 translated legal case reports. Best performances were found in the German translation scenario except for the F1 Score percentage where the experiment with the Italian dataset achieved the best score.

	Legal Case Report # 1000						
	F1 Scores						
	%	R-1	R-2	R-3	R-4	R-l	R-w1.2
Australian	70.51	49.51	32.91	29.39	28.28	51.26	24.20
Italian	**70.90**	64.74	55.68	53.83	52.79	67.69	34.68
German	69.62	**67.70**	**59.85**	**58.07**	**56.96**	**70.68**	**36.28**
Spanish	69.63	66.88	53.71	50.79	49.59	69.06	33.64
Danish	70.01	62.55	52.08	49.77	48.38	65.61	32.01
French	69.80	55.73	40.64	37.45	36.14	57.73	26.77

Table 6.4: Extractive Summarisation assessment - 1000 translated reports

Cross-Domain Translation Assessment.

Table 6.5 and 6.6 show the results achieved by performing the extractive summarisation task in the cross-domain scenario. Fine-tuning technique has been used as Transfer Learning approach. 4 main experiments have been conducted:

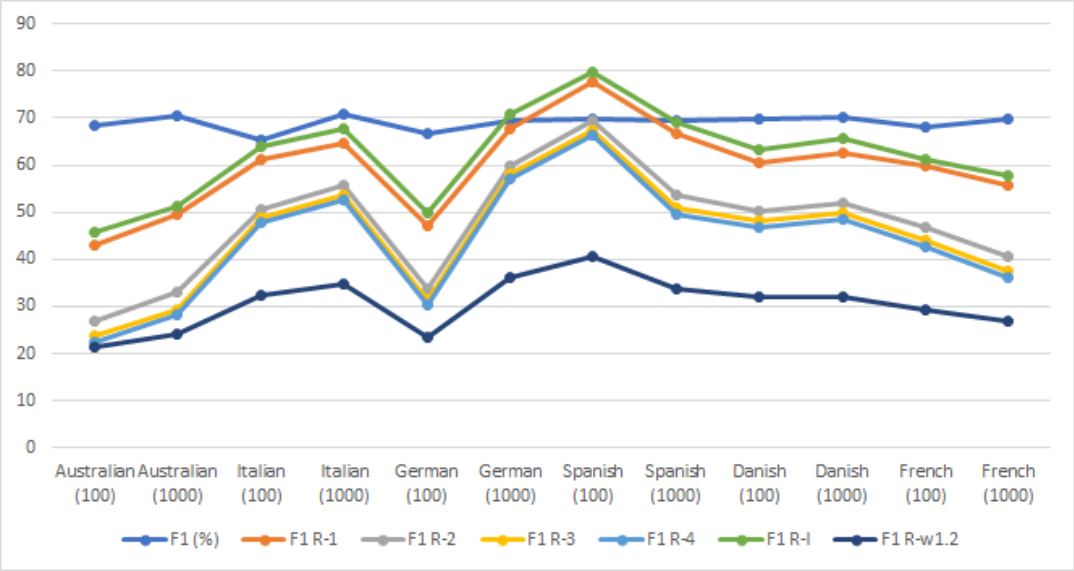


Figure 6.6: Extractive Summarisation - In-Domain Translation Assessment - Line Chart

Cross-Domain Extractive Summarisation (model trained on 70 English reports)		
	No Fine-Tuning	With Fine-Tuning
EN $\longrightarrow$ IT	70 EN $\longrightarrow$ 30 IT	70 EN, 70 IT $\longrightarrow$ 30 IT
F1 Score (%)	64.43	**71.61**
ROUGE-1 (F1)	57.49	**64.06**
ROUGE-2 (F1)	44.37	**55.66**
ROUGE-3 (F1)	42.11	**53.93**
ROUGE-4 (F1)	40.85	**52.87**
ROUGE-l (F1)	60.93	**66.96**
ROUGE-w1.2 (F1)	29.67	**35.46**

Table 6.5: Results of the cross domain experiments where an extractive model trained on 70 English reports has been tested on 30 reports translated into Italian.

- Testing of the extractive model trained on 70 English, on 70 Italian reports
- Fine-tuning of the extractive model trained on 70 English reports, with 70 Italian reports and testing of the resulting model on 30 Italian reports
- Testing of the extractive model trained on 700 English, on 60 Italian reports
- Fine-tuning of the extractive model trained on 700 English reports, with 140 Italian reports and testing the resulting model on 60 Italian reports

Our models show a good generalization capability across different language domains. In both fine-tuning scenarios, performances have been boosted with respect to the other two experiments.

Cross-Domain Extractive Summarisation (model trained on 700 English reports)		
	No Fine-Tuning	With Fine-Tuning
EN $\longrightarrow$ IT	700 EN $\longrightarrow$ 60 IT	700 EN, 140 IT $\longrightarrow$ 60 IT
F1 Score (%)	66.21	**72.44**
ROUGE-1 (F1)	**66.87**	66.75
ROUGE-2 (F1)	57.12	**57.94**
ROUGE-3 (F1)	55.10	**56.15**
ROUGE-4 (F1)	53.86	**55.10**
ROUGE-l (F1)	**69.77**	69.59
ROUGE-w1.2 (F1)	**36.15**	35.94

Table 6.6: Results of the cross domain experiments where an extractive model trained on 700 English reports has been tested on 60 reports translated into Italian.

6.4.4.2 Abstractive Summarisation

The results of the abstractive summarisation processes performed on the original Australian Case Report Dataset are shown in Table 6.7. As expected, the results achieved in 1000 legal case reports scenario are much higher for each metric than the ones obtained in the experiment with 100 reports.

	Legal Case Report #					
	$\approx$ 100			$\approx$ 1000		
ROUGE-1	P: 26.55	R: 21.51	F1: 22.84	P: 28.93	R: 26.18	F1: **27.18**
ROUGE-2	P: 3.30	R: 2.76	F1: 2.88	P: 4.86	R: 4.43	F1: **4.59**
ROUGE-3	P: 0.17	R: 0.14	F1: 0.15	P: 0.99	R: 0.92	F1: **0.94**
ROUGE-4	P: 0.00	R: 0.00	F1: 0.00	P: 0.42	R: 0.41	F1: **0.42**
ROUGE-l	P: 28.10	R: 23.29	F1: 24.70	P: 30.37	R: 27.93	F1: **28.87**
ROUGE-w	P: 14.74	R: 5.71	F1: 7.93	P: 15.68	R: 7.06	F1: **9.64**

Table 6.7: Results of the abstractive Summarisation on the Australian Legal Case Report Dataset

Cross-Domain Translation Assessment.

Table 6.8 shows the results of the abstractive summarisation performed on 100 translated reports of the Australian Legal Case Report Dataset. Best ROUGE-3 score is achieved in the French dataset scenario while from the experiment with the Spanish dataset we got the best results.

Instead, Table 6.9 shows the results of the abstractive summarisation performed on 1000 translated reports of the Australian Legal Case Report Dataset. The best scores are achieved in the Spanish translation scenario for all metrics except for the ROUGE-3 and ROUGE-4 cases where the experiment with the French dataset got the first place.

FactCC.

	Legal Case Report # 100					
	F1 Scores					
	R-1	R-2	R-3	R-4	R-l	R-w1.2
Australian	22.84	2.88	0.15	0.00	24.70	7.93
Italian	22.00	1.19	0.10	0.00	24.25	8.06
German	21.83	2.51	0.41	0.03	24.12	8.02
Spanish	**33.47**	**4.93**	0.61	**0.19**	**36.31**	**11.97**
Danish	22.89	1.67	0.31	0.03	25.85	8.45
French	30.18	4.15	**0.78**	0.18	31.42	10.02

Table 6.8: Results of the abstractive Summarisation performed on 100 translated reports of the Australian Legal Case Report Dataset

	Legal Case Report # 1000					
	F1 Scores					
	R-1	R-2	R-3	R-4	R-l	R-w1.2
Australian	27.18	4.59	0.94	0.42	28.87	9.64
Italian	25.11	2.75	0.56	0.15	27.59	9.25
German	25.31	4.02	1.01	0.26	28.06	9.47
Spanish	**35.70**	**6.57**	1.56	0.53	**37.52**	**12.38**
Danish	25.39	2.83	0.73	0.23	27.90	9.15
French	32.44	6.14	**1.83**	**0.76**	33.46	10.97

Table 6.9: Results of the abstractive summarisation performed on 1000 translated reports of the Australian Legal Case Report Dataset

Table 6.10 shows the results of the FactCC model evaluation on 30 and 900 Australian legal case reports scenarios, respectively. The legal case report number is doubled since their approach provides transformations which create another dataset with negative examples. As expected the model trained with more examples (2100) achieves the best results (among the two experiments), with much higher balanced accuracy and F1 score. This makes us think the model needs a number of training data to replicate or overcome the results obtained in the FactCC paper.

As the FactCC model has performed well on Australian Legal Case Reports, it will be used as metric for the evaluation of our generated summaries.

FactCC model assessment			
Evaluated Legal Case Report #	60	1800	4660
Balanced Accuracy (%	50	76.91	**77.61**
F1 Score (%	66.15	77.50	**78.53**

Table 6.10: FactCC model evaluation on Australian Legal Case Report Dataset

Consistency assessment of abstractive summaries using FactCC.

The goal here is to evaluate our abstractive summaries by running the previously trained

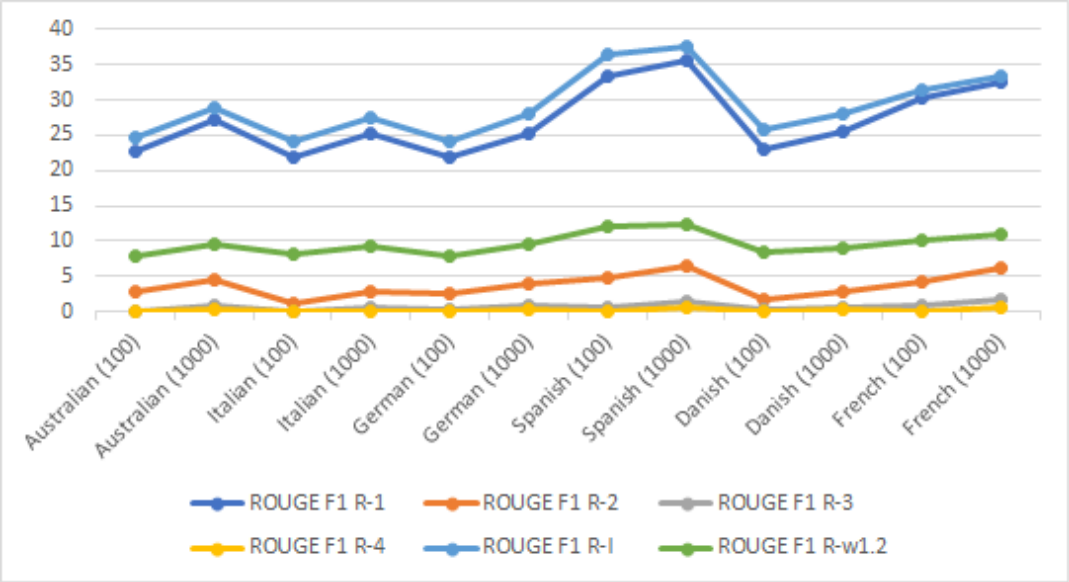


Figure 6.7: Abstractive Summarisation - Cross-Domain Translation Assessment - Line Chart

FactCC model (with 2100 training legal reports). This latter has been used to classify 100 and 1000 abstractive machine-generated summaries, respectively as `CORRECT` or `INCORRENT`. If a summary is evaluated as `CORRECT` we have reasonable assurance that is fluently and coherently written (syntactically speaking). The chosen training technique is BERT-base-uncased fine-tuning (for 8 epochs) using the default parameters of the FactCC model. Table 6.11 shows the ratio of report summaries classified as `CORRECT` out of the all evaluated summaries. In the evaluation of 300 abstractive summaries, it turned out about 46% of them are classified as `CORRECT`.

Abstractive summaries assessment via FactCC			
Machine-generated Abstractive Summaries #	30	300	650
CORRECT ratio (%)	41.67	46.70	**49.92**

Table 6.11: Australian abstractive summaries assessment via FactCC fine-tuned model

6.5 Evaluation

6.5.1 Extractive Summarisation

In order to compare our results with the state-of-art, we searched all the works which used the Australian Legal Case Report Dataset. The latest one we found is "Automatic Catchphrase Extraction from Legal Case Documents via Scoring using Deep Neural Networks" [18] by Vu Tran, Minh Le Nguyen, Ken Satoh. This latter has been used as baseline for the extractive summarisation tasks since it has analogies with our work:

- it used the same data;
- it did not involve citation data in the training process;
- it only used sentences and words from catchphrases as target data.

Vu Tran et al. achieved the results in Figure 6.8.

	ROUGE-1			ROUGE-SU6			ROUGE-W-1.2		
	Pre	Rec	Fm	Pre	Rec	Fm	Pre	Rec	Fm
Results of our system									
2006	0.2148	0.3244	0.2242	0.0611	0.1186	0.0524	0.1373	0.1508	0.1189
2007	0.2258	0.3204	0.2321	0.0644	0.1129	0.0539	0.1404	0.1393	0.1182
2008	0.2255	0.2906	0.2202	0.0656	0.0996	0.0501	0.1430	0.1285	0.1141
2009	0.2584	0.2982	0.2414	0.0830	0.0999	0.0583	0.1591	0.1267	0.1189
Average	0.2311	0.3084	0.2295	0.0685	0.1078	0.0537	0.1450	0.1363	0.1175

Figure 6.8: Results from the paper "Automatic Catchphrase Extraction from Legal Case Documents via Scoring using Deep Neural Networks"

Basing our comparison on their average results, it can be stated our model achieved really good performance in syntactical terms. ROUGE-1 and ROUGE-W.1.2 scores obtained in our experiments are much higher. The main motivation could be the use of BERT as word/sentence embedding builder system. As it has been explained in the first chapter, the choice of a good contextualized embedding model is a key choice to achieve better performances. Another reason could be the use of more expressive classification model: it has been used LSTM networks combined with CNN ones while in their work only CNNs have been applied. Extractive summarisations of translated reports achieve better results for almost all metrics than the English scenario. In particular the best scores have been obtained by the experiment with the Spanish dataset for the ROUGE metrics and the Danish dataset for the F1 Score %. This give us the intuition that the model benefits from the use of BERT multilingual model as embedding builder. This latter let us generate expressive contextualized word embeddings and allow us to work with different languages.

6.5.2 Abstractive Summarisation

Since no similiar work on the abstractive summarisation of the same dataset has been found, here extractive summarisation results have been used as baseline. As one can expect, ROUGE scores of the abstractive summarisation tasks are much lower than the extractive summarisation ones. Indeed, ROUGE score is a mere lexical measure and in the extractive scenario if the model succeeds in classify one sentence as relevant or not, then all the words of that sentence represents an overlapping between the generated and reference summary. This does not apply to the abstractive summarisation task by definition because here the

goal is to produce a new summary using also words that do not exist in the input text to summarize. Anyway, the ROUGE scores of the abstractive summarisation experiments are similiar to Vu Tran et al. work [18] so we have the intuition that our abstractive model has been able to generate speechs which are inherent to the input text even if consistency and fluidity of speech is to verify yet. In order to do that, FactCC model has been applied and it turned out about 46% of 300 machine-generated reports have been classified as `CORRECT`. Even though our fine-tuned FactCC model has a 77% F1 score (i.e. it is affected by errors), this fact give us the intuition our abstractive summaries have a certain degree of fluency and coherency with respect to their related legal report original texts, which are not reflected in the ROUGE rating. Abstractive summarisations of translated reports achieve similiar results to the English scenario and even better for Spanish and French languages. This give us the intuition that the model keep working well also in languages other than English and could be applied to a bunch of use cases.

6.6 Deployment

6.6.1 Extractive Summarisation

A Keras model has been saved at the end of each training for the extractive summarisation experiments. It is possible to load that model afterwards and use it to classify each input sentence as relevant or not. In the end sentences classified as relevant will be concatenated together to build the generate summary. In order to load the Keras model one can use the popular Keras API:

```
keras.models.load\_model("model_path")
```

Algorithm 6.2: The Keras API command to load the saved extractive model

While in order to generate a summary one can use the function `generate_summary` present in the project code.

```
def generate_summary(text_to_summarise, model):
    sentences = split_text_into_sentences(text_to_summarise)
    selected_sentences = []

    for sentence in sentences:
        if args.document_embedding_method == 'mean':
            sentence_embedding = embed_sentence_by_avg(sentence.
                to_original_text()).cpu().numpy()
        else:
            embedding_builder.embed(sentence)
            sentence_embedding = sentence.embedding.view(-1,
                embedding_features_num).detach().cpu().numpy()
        sentence_embedding = np.reshape(sentence_embedding, (1, 1,
            x_train.shape[2]))
        prediction = model.predict_classes(sentence_embedding)
        # check if the prediction tell us it is a catchphrase or not
```

```
            if bool(prediction[0]):
                selected_sentences.append(sentence.to_original_text())

        summary = '\n'.join(str(sentence) for sentence in
            selected_sentences)

        return summary
```

Algorithm 6.3: The developed Python function to generate the summary string given the input text and the loaded model.

6.6.2 Abstractive Summarisation

Each Transformer training produces a model which is saved in the `transformer_trained_model` directory (easily configurable by changing the script parameters). One can use that model by launching the `run_generation_with_transfomers.py` script which prompts for an input text and outputs the generated text. If the proper tags (`<abstract>`, `<eos>`, etc.) are used (to divide the text from the summary and mark the end of the sentence) this can be used to test some examples. At production time the model can be used the `run_text_generation` present in the project code.

```
    def run_text_generation(args):
    args.device = torch.device("cuda" if torch.cuda.is_available() and not
         args.no_cuda else "cpu")
    args.n_gpu = torch.cuda.device_count()

    set_seed(args)

    args.model_type = args.model_type.lower()
    model_class, tokenizer_class = MODEL_CLASSES[args.model_type]
    tokenizer = tokenizer_class.from_pretrained(args.model_name_or_path)
    model = model_class.from_pretrained(args.model_name_or_path)
    model.to(args.device)
    model.eval()

    if args.length < 0 and model.config.max_position_embeddings > 0:
        args.length = model.config.max_position_embeddings
    elif 0 < model.config.max_position_embeddings < args.length:
        args.length = model.config.max_position_embeddings  # No
            generation bigger than model size
    elif args.length < 0:
        args.length = MAX_LENGTH  # avoid infinite loop

    logger.info(args)
    if args.model_type in ["ctrl"]:
        if args.temperature > 0.7:
            logger.info('CTRL typically works better with lower
                temperatures (and lower top_k).')

    while True:
        xlm_lang = None
```

```
        # XLM Language usage detailed in the issues #1414
        if args.model_type in ["xlm"] and hasattr(tokenizer, 'lang2id')
            and hasattr(model.config, 'use_lang_emb') \
                and model.config.use_lang_emb:
            if args.xlm_lang:
                language = args.xlm_lang
            else:
                language = None
                while language not in tokenizer.lang2id.keys():
                    language = input(
                        "Using XLM. Select language in " + str(list(
                            tokenizer.lang2id.keys())) + " >>> ")
            xlm_lang = tokenizer.lang2id[language]

        # XLM masked-language modeling (MLM) models need masked token (see
             details in sample_sequence)
        is_xlm_mlm = args.model_type in ["xlm"] and 'mlm' in args.
            model_name_or_path
        if is_xlm_mlm:
            xlm_mask_token = tokenizer.mask_token_id
        else:
            xlm_mask_token = None

        summaries = []

        raw_text = args.input_instances if args.input_instances else input
            ("Model prompt >>> ")
        if args.model_type in ["transfo-xl", "xlnet"]:
            # Models with memory likes to have a long prompt for short
                inputs.
            raw_text = (args.padding_text if args.padding_text else
                PADDING_TEXT) + raw_text
        context_tokens = tokenizer.encode(raw_text)
        if args.model_type == "ctrl":
            if not any(context_tokens[0] == x for x in tokenizer.
                control_codes.values()):
                logger.info(
                    "WARNING! You are not starting your generation from a
                        control code so you won't get good results")
        out = sample_sequence(
            model=model,
            context=context_tokens,
            length=args.length,
            temperature=args.temperature,
            top_k=args.top_k,
            top_p=args.top_p,
            repetition_penalty=args.repetition_penalty,
            is_xlnet=bool(args.model_type == "xlnet"),
            is_xlm_mlm=is_xlm_mlm,
            xlm_mask_token=xlm_mask_token,
            xlm_lang=xlm_lang,
            device=args.device,
        )
        out = out[0, len(context_tokens):].tolist()

        text = tokenizer.decode(out, clean_up_tokenization_spaces=True,
```

```
            skip_special_tokens=True)
        text = text[: text.find(args.stop_token) if args.stop_token else
            None]

        print()
        logger.info("Generated␣text:␣" + text)

        if args.input_instances:
            break

    return text
```

Algorithm 6.4: The Python function adapted from a Transformers library script to generate a text given the input text and other model parameters.

Text generation parameters include:

```
# ============ PARAMETERS ==============
# "Model type selected in the list: " + ", ".join(MODEL_CLASSES.keys()
    - REQUIRED
model_type = "gpt2"
# "Path to pre-trained model or shortcut name selected in the list: "
    + ", ".join(ALL_MODELS) - REQUIRED
model_name_or_path = "transformer_trained_model"
# Text to pass
padding_text = ""
# Optional language when used with the XLM model
xlm_lang = ""
# temperature of 0 implies greedy sampling
temperature = 1.0
# primarily useful for CTRL model; in that case, use 1.2
repetition_penalty = 1.0
top_k = 0
top_p = 0.9
# Avoid to use when CUDA is available
no_cuda = False
# random seed for initialization
seed = 42
# Token at which text generation is stopped
stop_token = '<eos>'
# ======================================
```

Algorithm 6.5: Parameters of the text generation script.

Chapter 7

Experiments Manual

7.1 Extractive-based summarisation

7.1.1 Single-Language

```
python legal_case_reports_summarisation.py
    --dataset="[DATASET_LANGUAGE]"
    --legal_report_number=[NUMBER_OF_DOCUMENTS]
```

Algorithm 7.1: Bash/DOS command for launching an extractive summarisation with a minimal set of arguments.

Example.

```
python legal_case_reports_summarisation.py
    --dataset="australian"
    --legal_report_number=100
```

Algorithm 7.2: An example of bash/DOS command for launching an extractive summarisation with a minimal set of arguments.

7.1.2 Cross-Language without and with fine-tuning

Cross-Language without Fine-Tuning.

There are two main steps:

1. Training the model on a source language (if you have not already trained it)
2. Test the model on a target language

The Keras model (i.e. "`my_model.h5`") has to be placed in the same directory of the `extractive_summarisation_test.py` script.

```
  # Step 1 (Optional if you already have the source domain model)
python legal_case_reports_summarisation.py
    --dataset="[SOURCE_DATASET_LANGUAGE]"
```

```
    --legal_report_number=[NUMBER_OF_DOCUMENTS] &&
  # Step 2
python extractive_summarisation_test.py
    --model_path="[KERAS_MODEL_PATH]"
    --test_data_path="[TXT_DATASET_PATH]"
```

Algorithm 7.3: Bash/DOS command for launching a cross-language extractive summarisation without fine-tuning.

Example.

```
    # Step 1 (Optional if you already have the source domain model)
python legal_case_reports_summarisation.py
    --corpus_saving_mode="perform"
    --dataset="australian"
    --legal_report_number=1000 &&
    # Step 2
python extractive_summarisation_test.py
    --model_path="my_model.h5"
    --test_data_path="corpus_ita/fulltext/corpus.txt"
```

Algorithm 7.4: An example of Bash/DOS command for launching a cross-language extractive summarisation without fine-tuning.

Cross-Language with Fine-Tuning.

There are three main steps:

1. Training the model on a source language (if you have not already trained it)
2. Execute the fine-tuning process on target documents
3. Test the resulting model on other documents written in the target language

The Keras model (i.e. "`my_model.h5`") has to be placed in the PATH specified by the parameter `model_to_load_path` passed to the `legal_case_reports_summarisation.py` script.

```
  # Step 1 (Optional if you already have the source domain model)
python legal_case_reports_summarisation.py
    --dataset="[SOURCE_DATASET_LANGUAGE]"
    --legal_report_number=[NUMBER_OF_DOCUMENTS] &&
  # Step 2 & 3
python legal_case_reports_summarisation.py
    --legal_report_number=[FINE_TUNING_DOCUMENT_NUMBER]
    --dataset="[TARGET_DATASET_LANGUAGE]"
    --model_to_load_path="[KERAS_MODEL_PATH]"
```

Algorithm 7.5: Bash/DOS command for launching a cross-language extractive summarisation with fine-tuning.

Example.

```
    # Step 1 (Optional if you already have the source domain model)
 python legal_case_reports_summarisation.py
      --dataset="australian"
      --legal_report_number=1000 &&
  # Step 2 & 3
python legal_case_reports_summarisation.py
      --legal_report_number=100
      --dataset="italian"
      --model_to_load_path="my_model.h5"
```

Algorithm 7.6: An example of Bash/DOS command for launching a cross-language extractive summarisation with fine-tuning.

7.2 Abstractive-based summarisation

7.2.1 Preprocess

```
python legal_case_reports_summarisation.py
    --dataset="[DATASET_LANGUAGE]"
    --corpus_saving_mode="[PREPROCESS_MODE]"
    --legal_report_number=[NUMBER_OF_DOCUMENTS]
```

Algorithm 7.7: Bash/DOS command with a minimal set of arguments for launching data preprocess in order to preprocess data for an abstractive summarisation.

Example.

```
python australian_sentences_summarisation.py
    --dataset="australian"
    --corpus_saving_mode="perform"
    --legal_report_number=100
```

Algorithm 7.8: An example of bash/DOS command with a minimal set of arguments for launching data preprocess in order to perform an abstractive summarisation.

7.2.2 Model Fine Tuning

```
python transformer_fine_tuning.py
```

Algorithm 7.9: Bash/DOS command with a minimal set of arguments for launching a fine tuning of the model.

7.2.3 Text Generation

```
python transformer_text_generation.py --dataset="[DATASET_LANGUAGE]"
```

Algorithm 7.10: Bash/DOS command with a minimal set of arguments for launching a text generation.

7.2.4 Full Abstractive Summarisation process

```
python australian_sentences_summarisation.py
    --dataset="[DATASET_LANGUAGE]"
    --corpus_saving_mode=[PREPROCESS_MODE]
    --legal_report_number=[NUMBER_OF_DOCUMENTS] &&
python transformer_transformer_fine_tuning.py &&
python transformer_text_generation.py --dataset="[DATASET_LANGUAGE]"
```

Algorithm 7.11: An example of bash/DOS command with a minimal set of arguments for launching the entire abstractive summarisation process.

Example.

```
python australian_sentences_summarisation.py
    --dataset="australian"
    --corpus_saving_mode="perform"
    --legal_report_number=100 &&
python transformer_fine_tuning.py &&
python transformer_text_generation.py --dataset="australian"
```

Algorithm 7.12: An example of bash/DOS command with a minimal set of arguments for launching the entire abstractive summarisation process.

7.3 Australian Legal Case Reports Translation

```
python translate_and_convert_dataset_in_txt.py
    --dest_lang=[DESTINATION_LANGUAGE_CODE]
```

Algorithm 7.13: Bash/DOS command for launching the translation of the Australian Legal Case Report dataset from English.

Example.

```
python translate_and_convert_dataset_in_txt.py --dest_lang="it"
```

Algorithm 7.14: Bash/DOS command for launching the translation of the Australian Legal Case Report dataset from English to Italian.

7.4 FactCC evaluation on Australian Legal Case Reports

7.4.1 Data generation

7.4.1.1 Saving JSONL files containing legal case reports

```
python australian_sentences_summarisation.py
    --dataset="[DATASET_LANGUAGE]"
    --corpus_saving_mode="perform"
    --legal_report_number=[NUMBER_OF_DOCUMENTS]
```

Algorithm 7.15: Step 1 - Bash/DOS command for saving JSONL files containing legal case reports.

Example.

```
python australian_sentences_summarisation.py
  --dataset="australian"
  --corpus_saving_mode="perform"
  --legal_report_number=100
```

Algorithm 7.16: Step 1 - Example of /DOS command for saving JSONL files containing 100 Australian legal case reports.

7.4.1.2 FactCC data generation

Training data generation.

```
python factCC/data_generation/create_data.py data-train.jsonl
  --augmentations [AUGMENTATIONS_LIST]
```

Algorithm 7.17: Step 2 - Bash/DOS command for generating training data of FactCC model.

Example of training data generation.

```
python factCC/data_generation/create_data.py data-train.jsonl
    --augmentations pronoun_swap date_swap number_swap entity_swap
        negation noise
```

Algorithm 7.18: Step 2 - Example of Bash/DOS command for generating training data of FactCC model.

Validation data generation.

```
python factCC/data_generation/create_data.py data-dev.jsonl
  --augmentations [AUGMENTATIONS_LIST]
```

Algorithm 7.19: Step 3 - Bash/DOS command for generating validation data of FactCC model.

Example of training data generation.

```
python factCC/data_generation/create_data.py data-dev.jsonl
    --augmentations pronoun_swap date_swap number_swap entity_swap
        negation noise
```

Algorithm 7.20: Step 3 - Example of Bash/DOS command for generating validation data of FactCC model.

FactCC example concatenation.

Commands for Unix systems:

```
cat data-train-negative.jsonl data-train-positive.jsonl > data-train.jsonl
```

Algorithm 7.21: Step 4 - Bash command for concatenating training positive and negative data of FactCC model.

```
cat data-dev-negative.jsonl data-dev-positive.jsonl > data-dev.jsonl
```

Algorithm 7.22: Step 5 - Bash command for concatenating validation positive and negative data of FactCC model.

Commands for Windows systems:

```
type data-train-negative.jsonl data-train-positive.jsonl > data-train.
    jsonl
```

Algorithm 7.23: Step 4 - DOS command for concatenating training positive and negative data of FactCC model.

```
type data-dev-negative.jsonl data-dev-positive.jsonl > data-dev.jsonl
```

Algorithm 7.24: Step 5 - DOS command for concatenating validation positive and negative data of FactCC model.

7.4.2 Model training and validation

7.4.2.1 Training from scratch without evaluation

```
python factCC/modeling/run.py
        --task_name factcc_generated
        --do_train
        --do_lower_case
        --train_from_scratch
        --data_dir ./
        --model_type bert
        --model_name_or_path bert-base-uncased
        --max_seq_length 512
        --per_gpu_train_batch_size <BATCH_SIZE>
        --learning_rate <LEARNING_RATE>
        --num_train_epochs <TRAINING_EPOCHS_NUMBER>
        --overwrite_cache
        --output_dir <OUTPUT_DIR_PATH>
```

Algorithm 7.25: Step 6 - Bash/DOS command for training from scratch the FactCC model without evaluation during the process.

Example.

```
python factCC/modeling/run.py
    --task_name factcc_generated
    --do_train
    --do_lower_case
    --train_from_scratch
    --data_dir ./
    --model_type bert
    --model_name_or_path bert-base-uncased
    --max_seq_length 512
    --per_gpu_train_batch_size 12
    --learning_rate 2e-5
    --num_train_epochs 20.0
    --overwrite_cache
    --output_dir $HOME/LAILA-Project-Software/factCC/bert-base-uncased-
        factcc_generated-train-$RANDOM
```

Algorithm 7.26: Step 6 - Example of Bash command for training from scratch the FactCC model without evaluation during the process.

7.4.2.2 Training from scratch with evaluation

```
python factCC/modeling/run.py
        --task_name factcc_generated
        --do_train
        --do_lower_case
        --train_from_scratch
        --data_dir ./
        --model_type bert
        --model_name_or_path bert-base-uncased
        --max_seq_length 512
        --per_gpu_train_batch_size <BATCH_SIZE>
        --learning_rate <LEARNING_RATE>
        --num_train_epochs <TRAINING_EPOCHS_NUMBER>
        --overwrite_cache
        --output_dir <OUTPUT_DIR_PATH>
        --evaluate_during_training
        --eval_all_checkpoints
        --do_eval
```

Algorithm 7.27: Step 6 - Bash/DOS command for training from scratch the FactCC model with evaluation during the process.

Example.

```
python factCC/modeling/run.py
    --task_name factcc_generated
    --do_train
    --do_lower_case
    --train_from_scratch
    --data_dir ./
    --model_type bert
    --model_name_or_path bert-base-uncased
    --max_seq_length 512
```

```
    --per_gpu_train_batch_size 12
    --learning_rate 2e-5
    --num_train_epochs 20.0
    --overwrite_cache
    --output_dir $HOME/LAILA-Project-Software/factCC/bert-base-uncased-
        factcc_generated-train-$RANDOM
    --evaluate_during_training
    --eval_all_checkpoints
    --do_eval
```

Algorithm 7.28: Step 6 - Example of Bash command for training from scratch the FactCC model without validation during the process.

7.4.2.3 Fine tuning BERT with validation during training

```
python factCC/modeling/run.py
        --task_name factcc_generated
        --do_train
        --do_lower_case
        --data_dir ./
        --model_type bert
        --model_name_or_path bert-base-uncased
        --max_seq_length 512
        --per_gpu_train_batch_size <BATCH_SIZE>
        --learning_rate <LEARNING_RATE>
        --num_train_epochs <FINE_TUNING_EPOCHS_NUMBER>
        --overwrite_cache
        --output_dir <OUTPUT_DIR_PATH>
        --evaluate_during_training
        --eval_all_checkpoints
        --do_eval
```

Algorithm 7.29: Step 6 - Bash/DOS command for fine tuning BERT and evaluate the model during the process.

Example.

```
python factCC/modeling/run.py
    --task_name factcc_generated
    --do_train
    --do_lower_case
    --data_dir ./
    --model_type bert
    --model_name_or_path bert-base-uncased
    --max_seq_length 512
    --per_gpu_train_batch_size 12
    --learning_rate 2e-5
    --num_train_epochs 10.0
    --overwrite_cache
    --output_dir $HOME/LAILA-Project-Software/factCC/bert-base-uncased-
        factcc_generated-train-$RANDOM
    --evaluate_during_training
    --eval_all_checkpoints
    --do_eval
```

Algorithm 7.30: Step 6 - Example of Bash command for fine tuning BERT and evaluate the model during the process.

7.4.3 Model evaluation

You can evaluate the FactCC model with your data after the training process. Evaluation data have to be structured in a jsonl file named `data-dev.jsonl` containing one JSON object for each instance. Each JSON object should have the following fields: `<id>`, `<claim>`, `<text>`, `<label>`

```
python factCC/modeling/run.py
        --task_name factcc_annotated
        --do_lower_case
        --data_dir ./
        --model_type bert
        --model_name_or_path bert-base-uncased
        --max_seq_length 512
        --per_gpu_eval_batch_size <BATCH_SIZE>
        --overwrite_cache
        --output_dir <OUTPUT_DIR_PATH>
        --eval_all_checkpoints
        --do_eval
```

Algorithm 7.31: Step 6 - Bash/DOS command for evaluating the model.

Example.

```
python factCC/modeling/run.py
    --task_name factcc_annotated
    --do_lower_case
    --data_dir ./
    --model_type bert
    --model_name_or_path bert-base-uncased
    --max_seq_length 512
    --per_gpu_eval_batch_size 12
    --overwrite_cache
    --output_dir $HOME/LAILA-Project-Software/factCC/bert-base-uncased-
        factcc_generated-train-$RANDOM
    --eval_all_checkpoints
    --do_eval
```

Algorithm 7.32: Step 7 - Example of Bash command for evalute your previously trained FactCC model.

7.4.4 Evaluating the Factual Consistency of Abstractive Summaries

The model evaluation command mentioned above can also be used to evaluate summaries generated by the abstractive summarisation model (i.e. GPT-2 in our case). First, the text generation script, namely `transformer_text_generation.py`, has to be launched

passing the `factCC_evaluation=True` parameter. This way, the script will save the generated abstractive summaries in a JSONL file named `data-dev.jsonl` (as requested for the FactCC model evaluation). Afterwards the model evaluation script can be called as seen in the previous subsection. The metrics values shown by the model evaluation script represents the ratio of summaries classified as `CORRECT`: that is the percentage of consistent summaries with respect to their related report full text.

Chapter 8

Conclusions

In this work automatic summarisation of legal case reports has been tackled both by presenting extractive and abstractive summarisation approaches. In particular, the Australian Legal Case Report Dataset has been used in order to generate and evaluate automatic summaries. The solution proposed in the project rely on NLP and Deep Learning state-of-the-art techniques.
The extractive summarisation results obtained overtake the ones produced by the latest work on the same dataset in the literature, while the abstractive summarisation experiments achieve good results in syntactical terms even if speech consistency and fluidity should be tested.
In order to automatically evaluate that, FactCC model has been applied to this problem. It turned out about 50% of 650 machine-generated reports have been classified as pertinent to their related legal report text and syntactically correct. Even though our fine-tuned FactCC model can make inference errors on test data (F1: 77%) and the model itself still has limitations, this fact give us the intuition our abstractive summaries have a certain degree of fluency and coherency with respect to their related legal report original texts, which are not reflected in the ROUGE rating.
A translation task has been achieved in order to train our models and evaluate their ability to understand and summarize texts written in a language other than English. It turned out that the summarisation of translated reports achieves better results than the English report scenario for some other languages. Especially Spanish and French seems to perform generally better than other languages in the abstractive summarisation case while the summarisation of German reports achieves the best results in the extractive summarisation scenario with 1000 reports.
Hence, our models seems to handle in an effective way the summarisation of other languages than English and could be applied to other types of legal case report. This is supported by the use of Google Translate API and the BERT multilingual embedding model (this one only in the extractive summarisation scenario).

Transfer Learning tasks have been performed both for extractive and abstractive summarisation approaches. It turned out our models are able to generalize also in a cross-language scenario. Fine tuning technique has been applied and boosted almost all performance metrics of our extractive summarisation model.

8.1 Future work

The main challenges will be the improvement of the quality of the abstractive machine-generated summaries and their evaluation, especially in the abstractive summarisation scenario. This could be done by legal human experts but it is not a scalable approach, thus future work will be related to the automatic evaluation of generated summaries. Some automatic evaluation approaches like "Evaluating the Factual Consistency of Abstractive Text Summarization" [3] by Kryscinski et al. have been proposed in the literature and represent an improvement to the previous solutions to this problem even though they still have limitations.

Bibliography

[1] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 2019.

[2] Ben Hamner. Nips 2015 papers. `https://www.kaggle.com/benhamner/nips-2015-papers`, May 2017.

[3] Wojciech Kryściński, Bryan McCann, Caiming Xiong, and Richard Socher. Evaluating the factual consistency of abstractive text summarization. *arXiv preprint arXiv:1910.12840*, 2019.

[4] Abigail See, Peter J. Liu, and Christopher D. Manning. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1073–1083, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[5] Yang Liu and Mirella Lapata. Text summarization with pretrained encoders. *arXiv preprint arXiv:1908.08345*, 2019.

[6] Sanghwan Bae, Taeuk Kim, Jihoon Kim, and Sang-goo Lee. Summary level training of sentence rewriting for abstractive summarization. *arXiv preprint arXiv:1909.08752*, 2019.

[7] Ming Zhong, Pengfei Liu, Danqing Wang, Xipeng Qiu, and Xuanjing Huang. Searching for effective neural extractive summarization: What works and what's next. *arXiv preprint arXiv:1907.03491*, 2019.

[8] Xingxing Zhang, Furu Wei, and Ming Zhou. Hibert: Document level pre-training of hierarchical bidirectional transformers for document summarization. *arXiv preprint arXiv:1905.06566*, 2019.

[9] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter J Liu. Pegasus: Pre-training with extracted gap-sentences for abstractive summarization. *arXiv preprint arXiv:1912.08777*, 2019.

[10] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

[11] Filippo Galgani and Achim Hoffmann. Lexa: Towards automatic legal citation classification. In *Australasian Joint Conference on Artificial Intelligence*, pages 445–454. Springer, 2010.

[12] Filippo Galgani, Paul Compton, and Achim Hoffmann. Towards automatic generation of catchphrases for legal case reports. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 414–425. Springer, 2012.

[13] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[14] Filippo Galgani, Paul Compton, and Achim Hoffmann. Knowledge acquisition for categorization of legal case reports. In *Pacific Rim Knowledge Acquisition Workshop*, pages 118–132. Springer, 2012.

[15] Filippo Galgani, Paul Compton, and Achim Hoffmann. Combining different summarization techniques for legal text. In *Proceedings of the workshop on innovative hybrid approaches to the processing of textual data*, pages 115–123. Association for Computational Linguistics, 2012.

[16] Filippo Galgani, Paul Compton, and Achim Hoffmann. Citation based summarisation of legal texts. In *Pacific Rim International Conference on Artificial Intelligence*, pages 40–52. Springer, 2012.

[17] Arpan Mandal, Kripabandhu Ghosh, Arindam Pal, and Saptarshi Ghosh. Automatic catchphrase identification from legal court case documents. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 2187–2190, 2017.

[18] Vu Tran, Minh Le Nguyen, and Ken Satoh. Automatic catchphrase extraction from legal case documents via scoring using deep neural networks. *arXiv preprint arXiv:1809.05219*, 2018.

[19] Tshepho Koboyatshwene, Moemedi Lefoane, and Lakshmi Narasimhan. Machine learning approaches for catchphrase extraction in legal documents. In *FIRE (Working Notes)*, pages 95–98, 2017.

[20] Rupal Bhargava, Sukrut Nigwekar, and Yashvardhan Sharma. Catchphrase extraction from legal documents using lstm networks. In *FIRE (Working Notes)*, pages 72–73, 2017.

[21] S Kayalvizhi and D Thenmozhi. Deep learning approach for extracting catch phrases from legal documents. In *Neural Networks for Natural Language Processing*, pages 143–158. IGI Global, 2020.

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[23] Wikipedia. Language model — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Language%20model&oldid=922455992`, 2019. [Online; accessed 28-October-2019].

[24] William A Gale and Geoffrey Sampson. Good-turing frequency estimation without tears. *Journal of quantitative linguistics*, 2(3):217–237, 1995.

[25] Wikipedia. Good–Turing frequency estimation — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Good%E2%80%93Turing%20frequency%20estimation&oldid=910414241`, 2019. [Online; accessed 02-November-2019].

[26] Slava Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE transactions on acoustics, speech, and signal processing*, 35(3):400–401, 1987.

[27] Wikipedia. Katz's back-off model — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Katz's%20back-off%20model&oldid=842234206`, 2019. [Online; accessed 02-November-2019].

[28] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[29] Wikipedia. Word2vec — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Word2vec&oldid=922225367`, 2019. [Online; accessed 28-October-2019].

[30] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407, 1990.

[31] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[32] Wikipedia. GloVe (machine learning) — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=GloVe%20(machine%20learning)&oldid=922639597`, 2019. [Online; accessed 28-October-2019].

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[34] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[35] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[36] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[37] Iz Beltagy, Arman Cohan, and Kyle Lo. Scibert: Pretrained contextualized embeddings for scientific text. *arXiv preprint arXiv:1903.10676*, 2019.

[38] Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, and Jaewoo Kang. Biobert: pre-trained biomedical language representation model for biomedical text mining. *arXiv preprint arXiv:1901.08746*, 2019.

[39] Alan Akbik, Duncan Blythe, and Roland Vollgraf. Contextual string embeddings for sequence labeling. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 1638–1649, 2018.

[40] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[41] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[42] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.

[43] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[44] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

[45] Alan Akbik, Tanja Bergmann, Duncan Blythe, Kashif Rasul, Stefan Schweter, and Roland Vollgraf. Flair: An easy-to-use framework for state-of-the-art nlp. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 54–59, 2019.

[46] Wikipedia. Automatic summarization — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Automatic%20summarization&oldid=920519653`, 2019. [Online; accessed 02-November-2019].

[47] Aravind Pai and PSG College of Technology. Comprehensive guide to text summarization using deep learning in python. `https://bit.ly/2NCiS7f`, Jul 2019.

[48] Priya Dwivedi. Text summarization using deep learning. `https://towardsdatascience.com/text-summarization-using-deep-learning-6e379ed2e89c`, Mar 2019.

[49] Peter D Turney. Learning algorithms for keyphrase extraction. *Information retrieval*, 2(4):303–336, 2000.

[50] Wikipedia. Feedforward neural network — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Feedforward%20neural%20network&oldid=923043924`, 2019. [Online; accessed 05-November-2019].

[51] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[52] Frank Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, Cornell Aeronautical Lab Inc Buffalo NY, 1961.

[53] Marvin Minsky and Seymour Papert. Perceptron: an introduction to computational geometry. *The MIT Press, Cambridge, expanded edition*, 19(88):2, 1969.

[54] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[55] Wikipedia. Long short-term memory — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Long%20short-term%20memory&oldid=924024916`, 2019. [Online; accessed 05-November-2019].

[56] Wikipedia. Gated recurrent unit — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Gated%20recurrent%20unit&oldid=905922637`, 2019. [Online; accessed 05-November-2019].

[57] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[58] Wikipedia. Convolutional neural network — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Convolutional%20neural%20network&oldid=923101619`, 2019. [Online; accessed 05-November-2019].

[59] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object recognition with gradient-based learning. In *Shape, contour and grouping in computer vision*, pages 319–345. Springer, 1999.

[60] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[61] Rajat Newatia. How to implement cnn for nlp tasks like sentence classification. `https://bit.ly/2Qemjnc`, May 2019.

[62] Denny Britz. Understanding convolutional neural networks for nlp, Jan 2016.

[63] Prakhar Ganesh. Sentiment analysis : Simplified, Jun 2019.

[64] Gianluca Moro, Andrea Pagliarani, Roberto Pasolini, and Claudio Sartori. Cross-domain & in-domain sentiment analysis with memory-based deep neural networks. In Ana L. N. Fred and Joaquim Filipe, editors, *Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, IC3K 2018, Volume 1: KDIR, Seville, Spain, September 18-20, 2018*, pages 125–136. SciTePress, 2018.

[65] Andrea Pagliarani, Gianluca Moro, Roberto Pasolini, and Giacomo Domeniconi. Transfer learning in sentiment classification with deep neural networks. In Ana L. N. Fred, David Aveiro, Jan L. G. Dietz, Kecheng Liu, Jorge Bernardino, Ana Salgado, and Joaquim Filipe, editors, *Knowledge Discovery, Knowledge Engineering and Knowledge Management - 9th International Joint Conference, IC3K 2017, Funchal, Madeira, Portugal, November 1-3, 2017, Revised Selected Papers*, volume 976 of *Communications in Computer and Information Science*, pages 3–25. Springer, 2017.

[66] Giacomo Domeniconi, Gianluca Moro, Andrea Pagliarani, and Roberto Pasolini. On deep learning in cross-domain sentiment classification. In Ana L. N. Fred and Joaquim Filipe, editors, *Proceedings of the 9th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management - (Volume 1), Funchal, Madeira, Portugal, November 1-3, 2017*, pages 50–60. SciTePress, 2017.

[67] Giacomo Domeniconi, Konstantinos Semertzidis, Gianluca Moro, Vanessa López, Spyros Kotoulas, and Elizabeth M. Daly. Identifying conversational message threads by integrating classification and data clustering. In Chiara Francalanci and Markus Helfert, editors, *Data Management Technologies and Applications - 5th International Conference, DATA 2016, Lisbon, Portugal, July 24-26, 2016, Revised Selected Papers*, volume 737 of *Communications in Computer and Information Science*, pages 25–46. Springer, 2016.

[68] Giacomo Domeniconi, Konstantinos Semertzidis, Vanessa López, Elizabeth M. Daly, Spyros Kotoulas, and Gianluca Moro. A novel method for unsupervised and supervised conversational message thread detection. In Chiara Francalanci and Markus Helfert, editors, *DATA 2016 - Proceedings of 5th International Conference on Data Management Technologies and Applications, Lisbon, Portugal, 24-26 July, 2016*, pages 43–54. SciTePress, 2016.

[69] Giacomo Domeniconi, Gianluca Moro, Roberto Pasolini, and Claudio Sartori. A comparison of term weighting schemes for text classification and sentiment analysis with a supervised variant of tf.idf. In Markus Helfert, Andreas Holzinger,

Orlando Belo, and Chiara Francalanci, editors, *Data Management Technologies and Applications - 4th International Conference, DATA 2015, Colmar, France, July 20-22, 2015, Revised Selected Papers*, volume 584 of *Communications in Computer and Information Science*, pages 39–58. Springer, 2015.

[70] Giacomo Domeniconi, Gianluca Moro, Andrea Pagliarani, and Roberto Pasolini. Cross-domain sentiment classification via polarity-driven state transitions in a markov model. In Ana L. N. Fred, Jan L. G. Dietz, David Aveiro, Kecheng Liu, and Joaquim Filipe, editors, *Knowledge Discovery, Knowledge Engineering and Knowledge Management - 7th International Joint Conference, IC3K 2015, Lisbon, Portugal, November 12-14, 2015, Revised Selected Papers*, volume 631 of *Communications in Computer and Information Science*, pages 118–138. Springer, 2015.

[71] Giacomo Domeniconi, Gianluca Moro, Roberto Pasolini, and Claudio Sartori. A study on term weighting for text categorization: A novel supervised variant of tf.idf. In Markus Helfert, Andreas Holzinger, Orlando Belo, and Chiara Francalanci, editors, *DATA 2015 - Proceedings of 4th International Conference on Data Management Technologies and Applications, Colmar, Alsace, France, 20-22 July, 2015*, pages 26–37. SciTePress, 2015.

[72] Giacomo Domeniconi, Gianluca Moro, Andrea Pagliarani, and Roberto Pasolini. Markov chain based method for in-domain and cross-domain sentiment classification. In Ana L. N. Fred, Jan L. G. Dietz, David Aveiro, Kecheng Liu, and Joaquim Filipe, editors, *KDIR 2015 - Proceedings of the International Conference on Knowledge Discovery and Information Retrieval, part of the 7th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2015), Volume 1, Lisbon, Portugal, November 12-14, 2015*, pages 127–137. SciTePress, 2015.

[73] Giacomo Domeniconi, Gianluca Moro, Roberto Pasolini, and Claudio Sartori. Cross-domain text classification through iterative refining of target categories representations. In Ana L. N. Fred and Joaquim Filipe, editors, *KDIR 2014 - Proceedings of the International Conference on Knowledge Discovery and Information Retrieval, Rome, Italy, 21 - 24 October, 2014*, pages 31–42. SciTePress, 2014.

[74] Giacomo Domeniconi, Gianluca Moro, Roberto Pasolini, and Claudio Sartori. Iterative refining of category profiles for nearest centroid cross-domain text classification. In Ana L. N. Fred, Jan L. G. Dietz, David Aveiro, Kecheng Liu,

and Joaquim Filipe, editors, *Knowledge Discovery, Knowledge Engineering and Knowledge Management - 6th International Joint Conference, IC3K 2014, Rome, Italy, October 21-24, 2014, Revised Selected Papers*, volume 553 of *Communications in Computer and Information Science*, pages 50–67. Springer, 2014.

[75] Gianluca Moro, Roberto Pasolini, and Claudio Sartori. Personalized web search via query expansion based on user's local hierarchically-organized files. In Ana L. N. Fred and Joaquim Filipe, editors, *Proceedings of the 9th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management - (Volume 1), Funchal, Madeira, Portugal, November 1-3, 2017*, pages 157–164. SciTePress, 2017.

[76] Giacomo Domeniconi, Gianluca Moro, Andrea Pagliarani, and Roberto Pasolini. Learning to predict the stock market dow jones index detecting and mining relevant tweets. In Ana L. N. Fred and Joaquim Filipe, editors, *Proceedings of the 9th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management - (Volume 1), Funchal, Madeira, Portugal, November 1-3, 2017*, pages 165–172. SciTePress, 2017.

[77] Gianluca Moro, Roberto Pasolini, Giacomo Domeniconi, Andrea Pagliarani, and Andrea Roli. Prediction and trading of dow jones from twitter: A boosting text mining method with relevant tweets identification. In Ana L. N. Fred, David Aveiro, Jan L. G. Dietz, Kecheng Liu, Jorge Bernardino, Ana Salgado, and Joaquim Filipe, editors, *Knowledge Discovery, Knowledge Engineering and Knowledge Management - 9th International Joint Conference, IC3K 2017, Funchal, Madeira, Portugal, November 1-3, 2017, Revised Selected Papers*, volume 976 of *Communications in Computer and Information Science*, pages 26–42. Springer, 2017.

[78] Rada Mihalcea and Paul Tarau. Textrank: Bringing order into text. In *Proceedings of the 2004 conference on empirical methods in natural language processing*, pages 404–411, 2004.

[79] Taher H Haveliwala. Topic-sensitive pagerank. In *Proceedings of the 11th international conference on World Wide Web*, pages 517–526. ACM, 2002.

[80] Günes Erkan and Dragomir R Radev. Lexrank: Graph-based lexical centrality as salience in text summarization. *Journal of artificial intelligence research*, 22:457–479, 2004.

[81] Raimi Karim. Attn: Illustrated attention. `https://towardsdatascience.com/attn-illustrated-attention-5ec4ad276ee3`, Sep 2019.

[82] Giuliano Giacaglia. How transformers work. https://towardsdatascience.com/transformers-141e32e69591, Apr 2019.

[83] Alec Radford. Better language models and their implications. https://openai.com/blog/better-language-models/, Oct 2019.

[84] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

[85] Reformer: The efficient transformer, Jan 2020.

[86] Alireza Dirafzoon. Illustrating the reformer, Feb 2020.

[87] Wikipedia. Locality-sensitive hashing — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Locality-sensitive%20hashing&oldid=943049600, 2020. [Online; accessed 09-March-2020].

[88] Aidan N. Gomez, Mengye Ren, Raquel Urtasun, and Roger B. Grosse. The reversible residual network: Backpropagation without storing activations. *CoRR*, abs/1707.04585, 2017.

[89] Chin-Yew Lin and Franz Josef Och. Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, page 605. Association for Computational Linguistics, 2004.

[90] Wikipedia. Python (programming language) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Python%20(programming%20language)&oldid=925025558, 2019. [Online; accessed 07-November-2019].

[91] Python data analysis library. https://pandas.pydata.org/.

[92] Wikipedia. Pandas (software) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Pandas%20(software)&oldid=923916407, 2019. [Online; accessed 04-November-2019].

[93] Zalandoresearch. zalandoresearch/flair. https://github.com/zalandoresearch/flair, Oct 2019.

[94] Keras: The python deep learning library. https://keras.io/.

[95] Wikipedia. Keras — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Keras&oldid=915428659, 2019. [Online; accessed 04-November-2019].

[96] Pytorch. https://pytorch.org/.

[97] Wikipedia. PyTorch — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=PyTorch&oldid=920603645, 2019. [Online; accessed 04-November-2019].

[98] Transformers. https://huggingface.co/transformers/.

[99] Wikipedia. Cross-industry standard process for data mining — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Cross-industry%20standard%20process%20for%20data%20mining&oldid=910404916, 2019. [Online; accessed 02-November-2019].

[100] What is the crisp-dm methodology? https://www.sv-europe.com/crisp-dm-methodology/.

[101] https://archive.ics.uci.edu/ml/datasets/LegalCaseReports.

[102] Filippo Galgani, Paul Compton, and Achim Hoffmann. Citation based summarisation of legal texts. In *PRICAI 2012*, volume LNCS 7458, pages 40–52. Springer, Heidelberg, 2012.

[103] Wikipedia. Bidirectional recurrent neural networks — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Bidirectional%20recurrent%20neural%20networks&oldid=928719052, 2019. [Online; accessed 22-December-2019].

[104] Miguel Romero Calvo. Dissecting bert part 1: The encoder. https://medium.com/dissecting-bert/dissecting-bert-part-1-d3c3d495cdb3, May 2019.

[105] Miguel Romero Calvo. Dissecting bert appendix: The decoder. https://medium.com/dissecting-bert/dissecting-bert-appendix-the-decoder-3b86f66b0e5f, Nov 2018.

[106] Jay Alammar. The illustrated gpt-2 (visualizing transformer language models). http://jalammar.github.io/illustrated-gpt2/.

[107] Zihang Dai, Zhilin Yang, Yiming Yang, William W Cohen, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

Thanks

First of all I want to thank Professor Moro for having involved me in this project and supported throughout the whole process with great confidence in me.

I would also like to thank Professor Giovanni Sartor for allowing me to participate in the LAILA project and the Summer School on AI & Law of the European University Institute.

The biggest thanks goes to my family, for always spending for me in every situation and in particular for my education. I hope that the achievement of the master's degree will pay off, at least in part, the patience, commitment and trust that have always been lavished for me. I thank them in particular for allowing me to live the exciting and formative experience of my trip to Norway for the Erasmus+ program which allowed me to grow so much from a professional point of view but above all from a human point of view.

A special thought goes to Elisa to instil so much joy and love into my life. Thank you for making me feel alive and remembering that happiness exists every day.

The final thanks goes to the friends of a lifetime and to my fellow students Mattia, Martina, Federica, Chiara, Alessandro, Enrico, Raphael, Bjorn and Sondre for having accompanied me with enthusiasm and collaboration along this path.

Bidirectional Neural Networks

A Bidirectional Recurrent Neural Networks

BRNNs connect two hidden layers of opposite directions to the same output. [103]
With this form of generative deep learning, the output layer can get information from past (backwards) and future (forward) states simultaneously.
BRNNs were introduced in 1997 by Schuster and Paliwal to increase the amount of input information available to the network. BRNN are especially useful when the context of the input is needed. For example, in handwriting recognition, the performance can be enhanced by knowledge of the letters located before and after the current letter.

B BRNN Architecture

The principle of BRNN is to split the neurons of a regular RNN into two directions, one for positive time direction (forward states), and another for negative time direction (backward states).
Those two states' output are not connected to inputs of the opposite direction states.
By using two time directions, input information from the past and future of the current time frame can be used
unlike standard RNN which requires the delays for including future information.

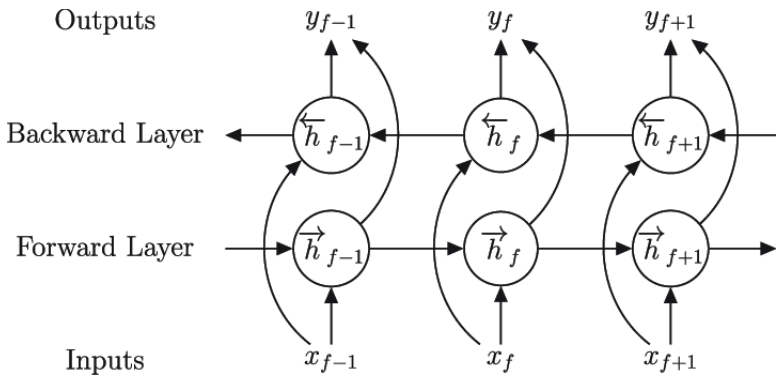


Figure 1: Bidirectional Recurrent Neural Network - an example of architecture

C BRNN Training

BRNNs are trained with similar algorithms as RNNs, since the two directional neurons do not interact with one another.
If back-propagation is necessary, some additional process is needed, since input and output layers cannot both be updated at once.

1. In general training, forward and backward states are processed first in the “forward” pass, before output neurons are passed.

2. For the backward pass, the opposite takes place; output neurons are processed first, then forward and backward states are passed next.

3. Weights are updated only after the forward and backward passes are complete.

Dissecting the Transformer

D Transformer (A. Vaswani et al., 2017)

Transformers are a popular type of neural network architecture recently used by OpenAI in their language models, and also used recently by DeepMind for AlphaStar. This latter is their program to defeat a top professional Starcraft player.
They were originally developed to solve the problem of sequence transduction, or neural machine translation. That means any task that transforms an input sequence to an output sequence (speech recognition, text-to-speech transformation, etc..).

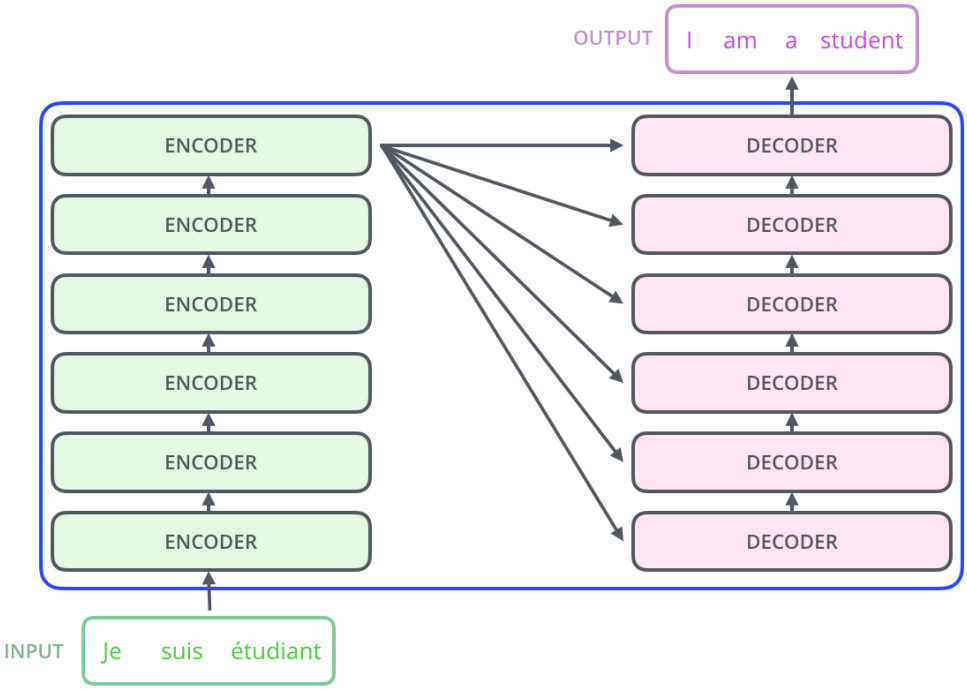


Figure 2: Transformer high-level architecture

E Model Architecture

Most competitive neural sequence transduction models have an encoder-decoder structure. Here, the encoder maps an input sequence of symbol representations $(x_1, \ldots, x_n)$ to a

sequence of continuous representations $z = (z_1, \ldots, z_n)$.
Given z, the decoder then generates an output sequence $(y_1, \ldots, y_m)$ of symbols one element at a time.
At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next.
The Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder [104] (left part) and decoder [105] (right part) shown in Figure 3.

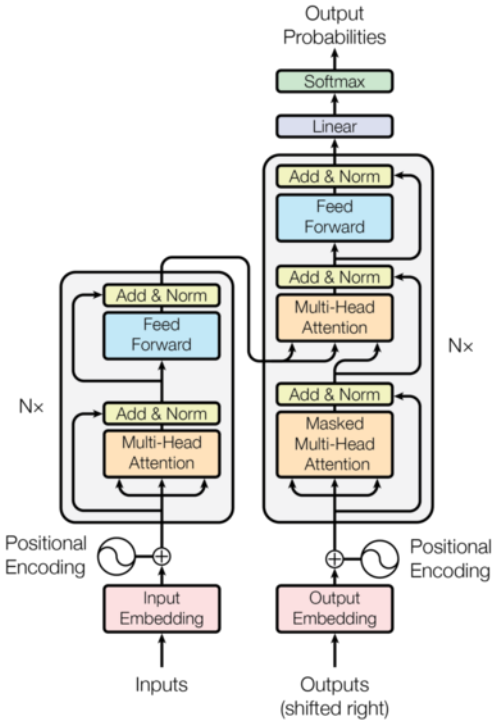


Figure 3: Transformer architecture

F Notation

- `emb_dim`: dimension of the token embeddings.
- `input_length`: length of the input sequence (the same in all sequences in a specific batch due to padding).
- `hidden_dim`: size of the Feed-Forward network's hidden layer.
- `vocab_size`: amount of words in the vocabulary (derived from the corpus)
- `target input`: we will use this term interchangeably to describe the input string (set of sentences) or sequence in the decoder.

G Information Flow

The data flow through the architecture is as follows:

1. The model represents each token as a vector of `emb_dim` size. With one embedding vector for each of the input tokens, we have a matrix of dimensions (`input_length`) x (`emb_dim`) for a specific input sequence.

2. It then adds positional information (positional encoding). This step returns a matrix of dimensions (`input_length`) x (`emb_dim`), just like in the previous step.

3. The data goes through N encoder blocks. After this, we obtain a matrix of dimensions (`input_length`) x (`emb_dim`).

4. The target sequence is masked and sent through the decoder's equivalent of 1) and 2). The output has dimensions (`target_length`) x (`emb_dim`).

5. The result of 4) goes through N decoder blocks. In each of the iterations, the decoder is using the encoder's output 3). This is represented in figure by the arrows from the encoder to the decoder. The dimensions of the output are (`target_length`) x (`emb_dim`).

6. Finally, it applies a fully connected layer and a row-wise softmax. The dimensions of the output are (`target_length`) x (`vocab_size`).

The dimensions of the input and output of the encoder block are the same. Hence, it makes sense to use the output of one encoder block as the input of the next encoder block.
The blocks do not share weights with each other.
In the original paper ("Attention Is All You Need"), $N = 6$

H Input Embedding

H.1 From Words to Vectors: tokenization, numericalization and word embeddings

Given a sentence in a corpus: " Hello, how are you?"

1. The first step is to tokenize it:
" Hello, how are you?" $\longrightarrow$ ["Hello", "," , "how", "are", "you", "?"]

2. This is followed by numericalization, mapping each token to a unique integer in the corpus' vocabulary:
["Hello", ", ", "how", "are", "you", "?"] $\longrightarrow$ [34, 90, 15, 684, 55, 193]

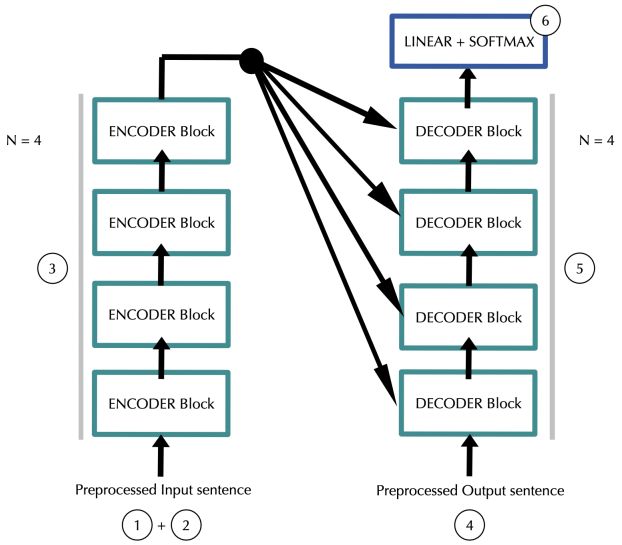


Figure 4: Transformer Information Flow

3. Next, we get the embedding for each word in the sequence. Each word of the sequence is mapped to a `emb_dim` dimensional vector that the model will learn during training.

   You can think about it as a vector look-up for each token. The elements of those vectors are treated as model parameters and are optimized with back-propagation just like any other weights.

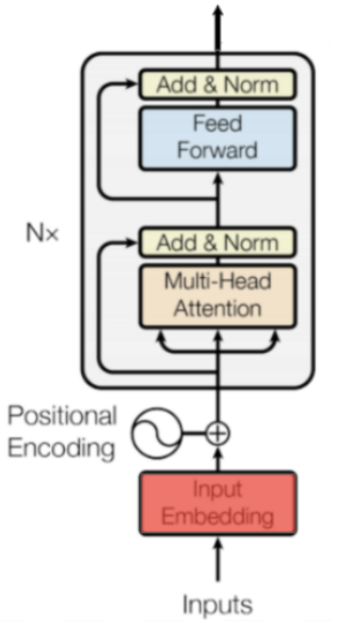


Figure 5: Transformer Input Embedding block

Stacking each of the vectors together we obtain a matrix Z of dimensions (`input_length`) x (`emb_dim`): It is important to remark that padding was used to make

	$<$	$-$	d_{emb_dim}	$-$	$>$
$Hello$	123.4	0.32	$\cdots$	94	32
,	83	34	$\cdots$	77	19
how	0.2	50	$\cdots$	33	30
are	289	432.98	$\cdots$	150	92
you	80	46	$\cdots$	23	32
?	41	21	$\cdots$	74	33

Figure 6: Word Embedding Matrix

the input sequences in a batch have the same length. That is, we increase the length of some of the sequences by adding `<pad>` tokens.

I Positional Encoding

At this point, we have a matrix (`input_length` $\times$ `embed_dim`) representation of our sequence. However, these representations are not encoding the fact that words appear in different positions.

Intuitively, we aim to be able to modify the represented meaning of a specific word depending on its position. We do not want to change the full representation of the word but we want to modify it a little to encode its position.

The approach chosen in the paper is to add numbers between [-1,1] using predetermined (non-learned) sinusoidal functions to the token embeddings. Now, for the rest of the Encoder, the word will be represented slightly differently depending on the position the word is in (even if it is the same word).

Moreover, we would like the Encoder to be able to use the fact that some words are in a given position while, in the same sequence, other words are in other specific positions. That is, we want the network to able to understand relative positions and not only absolute ones. The sinuosidal functions chosen by the authors allow positions to be represented as linear combinations of each other and thus allow the network to learn relative relationships between the token positions.

The approach chosen in the paper to add this information is adding to Z a matrix P with positional encodings.

$$Z + P \tag{1}$$

The authors chose to use a combination of sinusoidal functions. Mathematically, using i for the position of the token in the sequence and j for the position of the embedding feature: More specifically, for a given sentence P, the positional embedding matrix would be as follows:

$$p_{i,j} = \begin{cases} \sin\left(\frac{i}{10000^{\frac{j}{d_{emb\_dim}}}}\right) & \text{if } j \text{ is even} \\ \cos\left(\frac{i}{10000^{\frac{j-1}{d_{emb\_dim}}}}\right) & \text{if } j \text{ is odd} \end{cases}$$

Figure 7: Positional Encoding formula

$$\begin{array}{c} \\ Hello \\ , \\ how \\ are \\ you \\ ? \end{array} \begin{array}{c} \overset{<}{} \\ \begin{pmatrix} \sin\left(\frac{0}{10000^{\frac{0}{emb_{dim}}}}\right) \\ \sin\left(\frac{1}{10000^{\frac{0}{emb_{dim}}}}\right) \\ \sin\left(\frac{2}{10000^{\frac{0}{emb_{dim}}}}\right) \\ \sin\left(\frac{3}{10000^{\frac{0}{emb_{dim}}}}\right) \\ \sin\left(\frac{4}{10000^{\frac{0}{emb_{dim}}}}\right) \\ \sin\left(\frac{5}{10000^{\frac{0}{d_{emb_{dim}}}}}\right) \end{pmatrix} \end{array} \begin{array}{c} \overset{-}{} \\ \begin{pmatrix} \cos\left(\frac{0}{10000^{\frac{0}{emb_{dim}}}}\right) \\ \cos\left(\frac{1}{10000^{\frac{0}{emb_{dim}}}}\right) \\ \cos\left(\frac{2}{10000^{\frac{0}{emb_{dim}}}}\right) \\ \cos\left(\frac{3}{10000^{\frac{0}{emb_{dim}}}}\right) \\ \cos\left(\frac{4}{10000^{\frac{0}{emb_{dim}}}}\right) \\ \cos\left(\frac{5}{10000^{\frac{0}{emb_{dim}}}}\right) \end{pmatrix} \end{array} \begin{array}{c} \overset{d_{emb\_dim}}{} \\ \begin{pmatrix} \sin\left(\frac{0}{10000^{\frac{2}{emb_{dim}}}}\right) \\ \sin\left(\frac{1}{10000^{\frac{2}{emb_{dim}}}}\right) \\ \sin\left(\frac{2}{10000^{\frac{2}{emb_{dim}}}}\right) \\ \sin\left(\frac{3}{10000^{\frac{2}{emb_{dim}}}}\right) \\ \sin\left(\frac{4}{10000^{\frac{2}{emb_{dim}}}}\right) \\ \sin\left(\frac{5}{10000^{\frac{2}{emb_{dim}}}}\right) \end{pmatrix} \end{array} \begin{array}{c} \overset{-}{} \\ \begin{pmatrix} \cos\left(\frac{0}{10000^{\frac{2}{emb_{dim}}}}\right) \\ \cos\left(\frac{1}{10000^{\frac{2}{emb_{dim}}}}\right) \\ \cos\left(\frac{2}{10000^{\frac{2}{emb_{dim}}}}\right) \\ \cos\left(\frac{3}{10000^{\frac{2}{emb_{dim}}}}\right) \\ \cos\left(\frac{4}{10000^{\frac{2}{emb_{dim}}}}\right) \\ \cos\left(\frac{5}{10000^{\frac{2}{emb_{dim}}}}\right) \end{pmatrix} \end{array} \begin{array}{c} \overset{>}{} \\ \begin{pmatrix} \cdots \\ \cdots \\ \cdots \\ \cdots \\ \cdots \\ \cdots \end{pmatrix} \end{array}$$

Figure 8: Positional Encoding matrix

The resulting matrix:

$$X = Z + P \tag{2}$$

is the input of the first encoder block and has dimensions (`input_length`) x (`emb_dim`).

J Encoder block

J.1 Encoder block

A total of N encoder blocks are chained together to generate the Encoder's output. A specific block is in charge of finding relationships between the input representations and encode them in its output. Intuitively, this iterative process through the blocks will help the neural network capture more complex relationships between words in the input sequence. You can think about it as iteratively building the meaning of the input sequence as a whole.

J.2 Multi-Head Attention

The **Transformer** uses Multi-Head Attention, which means it computes attention h different times with different weight matrices and then concatenates the results together.

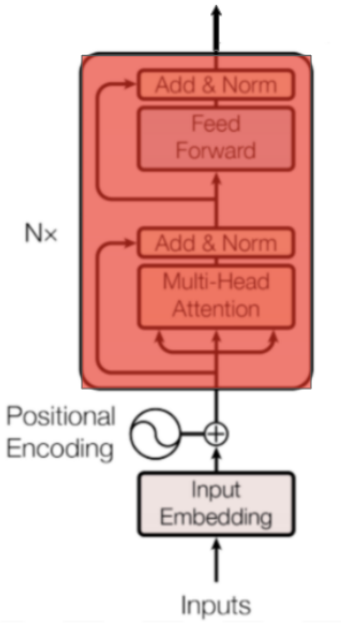


Figure 9: Transformer Encoder block

The result of each of those parallel computations of attention is called a head. We are going to denote a specific head and the associated weight matrices with the subscript i.
As shown in Figure 11, once all the heads have been computed they will be concatenated. This will result in a matrix of dimensions (`input_length`) x ($h \times d_v$).
Afterwards, a linear layer with weight matrix W^0 of dimensions ($h \times d_v$) x (`emb_dim`) will be applied leading to a final result of dimensions (`input_length`) x (`emb_dim`).
Mathematically:

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O \tag{3}$$

where

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \tag{4}$$

Where Q,K and V are placeholders for different input matrices. In particular, for this case Q,K and V will be replaced by the output matrix of the previous step X.

J.3 Scaled Dot-Product Attention

Each head is going to be characterized by three different projections (matrix multiplications) given by matrices:

- W_i^K with dimensions $d_{embed\_dim} \times d_k$
- W_i^Q with dimensions $d_{embed\_dim} \times d_k$
- W_i^V with dimensions $d_{embed\_dim} \times d_v$

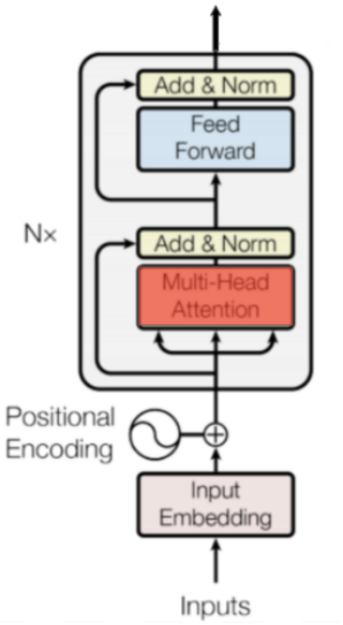


Figure 10: Transformer Encoder - Multi-Head Attention

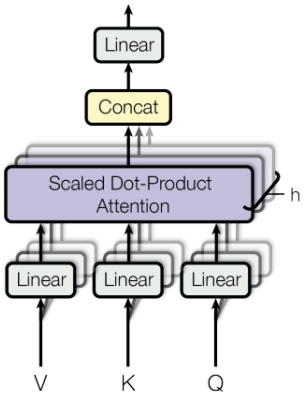


Figure 11: Scaled Dot-Product Attention representation

To compute a head we will take the input matrix X and separately project it with the above weight matrices:

- $XW_i^K = K_i$ with dimensions $input\_length \times d_k$
- $XW_i^Q = Q_i$ with dimensions $input\_length \times d_k$
- $XW_i^V = V_i$ with dimensions $input\_length \times d_v$

In the paper d_k and d_v are set such that $d_k = d_v = emb\_dim/h$
Once we have K_i, Q_i and V_i we use them to compute the Scaled Dot-Product Attention:

$$Attention(Q, K, V) = Softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{5}$$

In the encoder block the computation of attention does not use a mask.

J.4 Scaled Dot Product Attention: explanation

Let us start by looking at the matrix product between Q_i and K_i^T, i.e. $Q_i K_i^T$

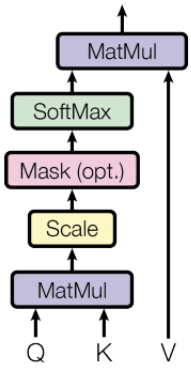


Figure 12: Dot Product Attention representation

Remember that Q_i and K_i were different projections of the tokens into a d_k dimensional space.

Therefore, we can think about the dot product of those projections as a measure of similarity between tokens projections.

For every vector projected through Q_i the dot product with the projections through K_i measures the similarity between those vectors.

If we call v_i and u_j the projections of the i-th token and the j-th token through Q_i and K_i respectively, their dot product can be seen as:

$$v_i v_j = cos(v_i, v_j)||v_i||_2||v_j||_2 \tag{6}$$

Thus, this is a measure of how similar are the directions of u_i and v_j and how large are their lengths (the closest the direction and the larger the length, the greater the dot product).

Another way of thinking about this matrix product is as the encoding of a specific relationship between each of the tokens in the input sequence (the relationship is defined by the matrices K_i, Q_i).

After this multiplication, the matrix is divided element-wise by the square root of d_k for scaling purposes.

The next step is a Softmax applied row-wise (one softmax computation for each row):

$$Softmax(\frac{Q_i K_i^T}{\sqrt{d_k}}) \tag{7}$$

In our example, this could be visualized in Figure 13.

The result would is rows with numbers between zero and one that sum to one. Finally, the result is multiplied by V_i to get the result of the head.

$$Softmax(\frac{Q_i K_i^T}{\sqrt{d_k}})V_i \tag{8}$$

$$
\begin{array}{c}
 \\
Hello \\
, \\
how \\
are \\
you \\
?
\end{array}
\begin{array}{c}
\begin{array}{cccccc}
Hello & , & how & are & you & ?
\end{array} \\
\begin{pmatrix}
78.49 & 43.29 & 1.2 & 41.74 & 91.43 & 74.47 \\
95.84 & 28.78 & 57.13 & 68.20 & -60.94 & 26.85 \\
-95.69 & -52.16 & 17.00 & 45.71 & 48.49 & 64.35 \\
-69.92 & 85.16 & 94.94 & 91.04 & -92.83 & 77.49 \\
65.85 & 55.85 & 62.54 & -97.46 & 76.38 & 13.20 \\
-30.05 & -4.52 & 76.02 & 42.35 & 15.29 & 63.61
\end{pmatrix}
\end{array}
\Longrightarrow
$$

Figure 13: An example of Attention Matrix

$$
\begin{array}{c}
 \\
Hello \\
, \\
how \\
are \\
you \\
?
\end{array}
\begin{array}{c}
\begin{array}{cccccc}
Hello & , & how & are & you & ?
\end{array} \\
\begin{pmatrix}
72.40 * 10^{-06} & 1.23 * 10^{-21} & 6.51 * 10^{-40} & 2.62 * 10^{-22} & 9.99 * 10^{-01} & 4.30 * 10^{-08} \\
1.00 * 10^{+00} & 7.51 * 10^{-30} & 1.54 * 10^{-17} & 9.91 * 10^{-13} & 8.15 * 10^{-69} & 1.09 * 10^{-30} \\
3.12 * 10^{-70} & 2.51 * 10^{-51} & 2.72 * 10^{-21} & 8.03 * 10^{-09} & 1.29 * 10^{-07} & 9.99 * 10^{-01} \\
2.47 * 10^{-72} & 5.54 * 10^{-05} & 9.80 * 10^{-01} & 1.98 * 10^{-02} & 2.77 * 10^{-82} & 2.58 * 10^{-08} \\
2.67 * 10^{-05} & 1.21 * 10^{-09} & 9.75 * 10^{-07} & 3.17 * 10^{-76} & 9.99 * 10^{-01} & 3.64 * 10^{-28} \\
8.59 * 10^{-47} & 1.05 * 10^{-35} & 9.99 * 10^{-01} & 2.38 * 10^{-15} & 4.21 * 10^{-27} & 4.07 * 10^{-06}
\end{pmatrix}
\begin{array}{c}
= 1 \\ = 1 \\ = 1 \\ = 1 \\ = 1 \\ = 1
\end{array}
\end{array}
$$

Figure 14: Attention Matrix after Softmax application

J.5 Multi-Head Attention: motivation

The idea behind it is that whenever you are translating a word, you may pay different attention to each word based on the type of question that you are asking.

For example, whenever you are translating “kicked” in the sentence “I kicked the ball”, you may ask “Who kicked”.

Depending on the answer, the translation of the word to another language can change.

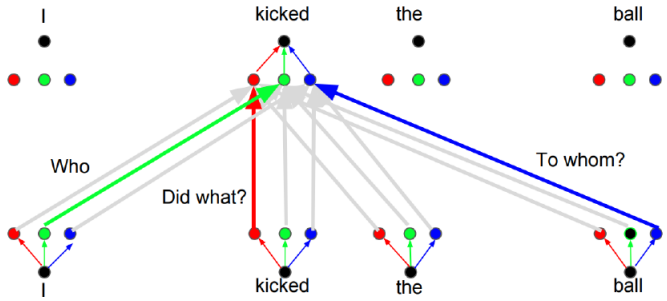


Figure 15: Multi-Head Attention representation

J.6 Position-wise Feed-Forward Network

This step is composed of the following layers:

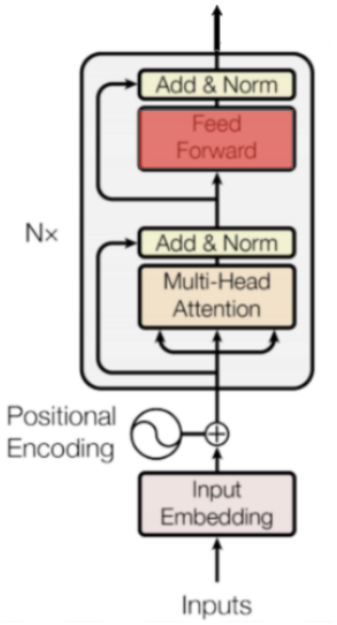


Figure 16: Transformer Encoder - Feed Forward

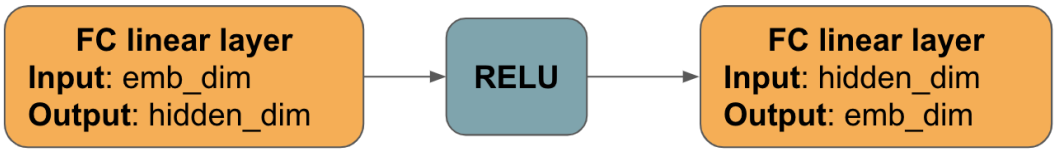


Figure 17: Position-wise Feed-Forward network representation

Mathematically, for each row in the output of the previous layer:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2 \tag{9}$$

where W_1 and W_2 are $(emb\_dim) \times (d_F)$ and $(d_F) \times (emb\_dim)$ matrices respectively. Observe that during this step, vector representations of tokens don't "interact" with each other. It is equivalent to run the calculations row-wise and stack the resulting rows in a matrix.

The output of this step has dimension $(input\_length) \times (emb\_dim)$.

J.7 Dropout, Add & Norm

Before this layer, there is always a layer for which inputs and outputs have the same dimensions (Multi-Head Attention or Feed-Forward). We will call that layer $Sublayer$ and its input x.

After each Sublayer, dropout is applied with 10% probability. Call this result $Dropout(Sublayer(x))$. This result is added to the Sublayer's input x, and we get $x + Dropout(Sublayer(x))$.

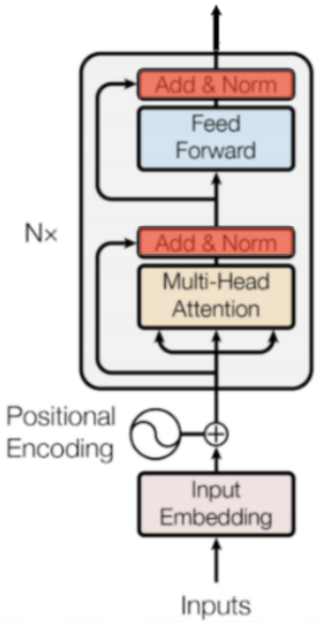


Figure 18: Transformer Encoder - Add & Norm

Observe that in the context of a Multi-Head Attention layer, this means adding the original representation of a token x to the representation based on the relationship with other tokens. It is like telling the token: "Learn the relationship with the rest of the tokens, but don't forget what we already learned about yourself!"
Finally, a token-wise/row-wise normalization is computed with the mean and standard deviation of each row:

$$x' = \frac{x - \bar{x}}{\sigma} \tag{10}$$

- Where x is the original feature vector, $\bar{x} = \text{average}(x)$ is the mean of that feature vector, and σ is its standard deviation.
- This improves the stability of the network (gradient descent converges much faster).

The output of these layers is:

$$LayerNorm(x + Dropout(Sublayer(x))) \tag{11}$$

K Decoder block

K.1 Decoder block: inputs

Remember that the described Transformer architecture is processing both the input sentence and the target sentence to train the network.
The input sentence will be encoded as described in The Encoder's architecture.

Here we discuss how given a target sentence (e.g. “Hola, como estás ?” in translation if the target language is Spanish) we obtain a matrix representing the target sentence for the decoder blocks.
The process is exactly the same as the encoder one. It is also composed of two general steps:

1. Token embeddings
2. Encoding of the positions.

The main difference is that the target sentence is shifted. That is, before padding, the target sequence will be as follows: [“Hola”, “, “, “como”, “estás”, “?”] $\longrightarrow$[“<SS>”, “Hola”, “, “, “como”, “estás”, “?”]
The rest of the process to vectorize the target sequence will be exactly as the one described for input sentences in The Encoder’s architecture.

K.2 Decoder block: Training vs. Testing

During test time we don’t have the ground truth. The steps, in this case, will be as follows:

1. Compute the embedding representation of the input sequence.
2. Use a starting sequence token, for example ‘<SS>’ as the first target sequence: [<SS>]. The model gives as output the next token.
3. Add the last predicted token to the target sequence and use it to generate a new prediction [‘<SS>’, $Prediction_1$,...,$Prediction_n$]
4. Do step 3 until the predicted token is the one representing the End of the Sequence, for example <EOS>.

During training we have the ground truth, i.e. the tokens we would like the model to output for every iteration of the above process.
Since we have the target in advance, we will give the model the whole shifted target sequence at once and ask it to predict the non-shifted target.
Following up with our previous examples we would input:
[‘<SS>’,’Hola’, ‘,’, ‘ como’, ‘estas’, ‘?’] and the expected prediction would be:
[’Hola’, ‘,’, ‘ como’, ‘estas’, ‘?’,’<EOS>’]
However, there is a problem here. What if the model sees the expected token and uses it to predict itself? For example, it might see ‘estas’ at the right of ‘como’ and use it to predict ‘estas’.

That's not what we want because the model will not be able to do that a testing time as at prediction time we are just going to care about the prediction of the next word of the last token in the target/output sequence.

We need to modify some of the attention layers to prevent the model of seeing information on the right (or down in the matrix of vector representation) but allow it to use the already predicted words.

The output of the decoder block will be also a matrix of sizes $(target\_length) \times (emb\_dim)$.

After a row-wise linear (a linear layer in the form of matrix product on the right) and a Softmax per row this will result in a matrix for which the maximum element per row indicates the next word.

That means that the row assigned to "<SS>" is in charge of predicting "Hola", the row assigned to "Hola" is in charge of predicting "," and so on.

Hence, to predict "estas" we will allow that row to directly interact with the green region but not with the red region.

The problem will be in Multi-Head Attention and the input will need to be masked.

	$<$	$-$	d_{emb_dim}	$-$	$>$
$< SS >$	-3.521	23.625	$\cdots$	-3.041	21.150
$Hola$	28.256	-27.21	$\cdots$	-30.80	44.232
,	-25.34	-39.38	$\cdots$	44.016	18.240
$como$	22.126	14.527	$\cdots$	-18.22	48.169
$estás$	49.093	-48.61	$\cdots$	-17.78	26.766
?	11.692	-40.30	$\cdots$	-4.356	32.497

Figure 19: Output Sequence Matrix

K.3 Masked Multi-Head Attention

This will work exactly as the Multi-Head Attention mechanism but adding masking to our input.

The only Multi-Head Attention block where masking is required is the first one of each decoder block.

This is because the one in the middle is used to combine information between the encoded inputs and the outputs inherited from the previous layers.

There is no problem in combining every target token's representation with any of the input token's representations (since we will have all of them at test time).

The modification will take place after computing: $\frac{Q_i K_i^T}{\sqrt{d_k}}$

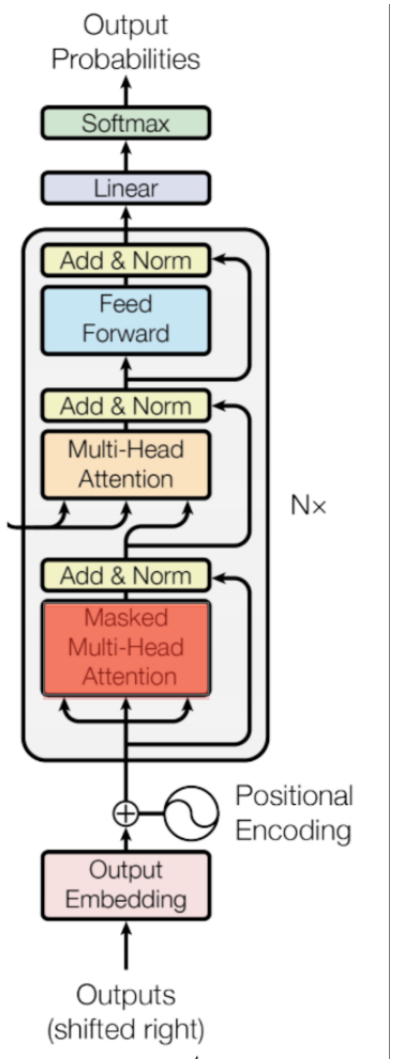


Figure 20: Transformer Decoder - Masked Multi-Head-Attention

Now, the masking step is just going to set to minus infinity all the entries in the strictly upper triangular part of the matrix (Figure 21). The rest of the process is identical as described in the Multi-Head Attention for the encoder.

Observe that if those entries are relative attention measures per each row, the larger they are, the more attention we need to pay to that token. So setting those elements to minus infinity is mainly saying: " For the row assigned to predicting "estás" (the one with input "como"), ignore "estás" and "?".

Our Softmax output would look like the matrix shown in Figure 22. The relative attention of those tokens that we were trying to ignore has indeed gone to zero. When multiplying this matrix with V_i the only elements that will be accounted for to predict the next word are the ones that the model will have access to during test time.

This time the output of the modified Multi-Head Attention layer will be a matrix of dimensions $(target\_length) \times (emb\_dim)$ because the sequence from which it has been

	$< SS >$	$Hola$	,	$como$	$estás$	?
$< SS >$	-29.59	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$Hola$	-15.26	46.884	$-\infty$	$-\infty$	$-\infty$	$-\infty$
,	30.225	-2.567	4.6751	$-\infty$	$-\infty$	$-\infty$
$como$	-8.656	-13.94	-29.00	48.459	$-\infty$	$-\infty$
$estás$	-5.156	48.210	8.5994	-20.29	-33.36	$-\infty$
?	-46.01	10.281	-39.73	28.344	25.826	20.824

Figure 21: Masked Attention Matrix

	$< SS >$	$Hola$	,	$como$	$estás$	?
$< SS >$	1.0	0	0	0	0	0
$Hola$	$1.026 * 10-27$	1.0	0	0	0	0
,	0.99	$5.73 * 10-15$	$8.013 * 10-12$	0	0	0
$como$	$1.56 * 10-25$	$7.95 * 10-28$	$2.29 * 10-34$	$1.12 * 10-39$	0	0
$estás$	$6.65 * 10-24$	1.0	$6.27 * 10-18$	$1.78 * 10-30$	$3.75 * 10-36$	0
?	$4.72 * 10-33$	$1.32 * 10-08$	$2.52 * 10-30$	0.92	0.07	$5 * 10-4$

Figure 22: Masked Attention Matrix after Softmax application

calculated has a sequence length of $target\_length$.

K.4 Multi-Head Attention — Encoder output and target

Observe that in this case we are using different inputs for that layer.
More specifically, instead of deriving Q_i, K_i and V_i from X as we have been doing in previous Multi-Head Attention layers, this layer will use both

- the Encoder's final output E (final result of all encoder blocks)
- and the Decoder's previous layer output D (the masked Multi-Head Attention after going through the Dropout, Add & Norm layer).

Let us first clarify the shape of those inputs and what they represent:

- E, the encoded input sequence, is a matrix of dimensions $(input\_length) \times (emb\_dim)$ which has encoded, by going through 6 encoder blocks, the relationships between the input tokens.
- D, the output from the masked Multi-Head Attention after going through the Add & Norm, is a matrix of dimensions $(target\_length) \times (emb\_dim)$.

Let us now dive into what to do with those matrices.
We will use weighted matrices with the same dimensions as before:

- W_i^K with dimensions $d_{model} \times d_k$
- W_i^Q with dimensions $d_{model} \times d_k$

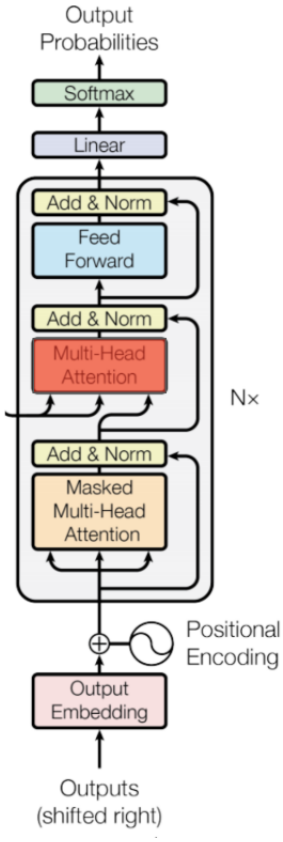


Figure 23: Transformer decoder - Multi-Head Attention

- W_i^V with dimensions $d_{model} \times d_v$

But this time the projection generating Q_i will be done using D (target information), while the ones generating K and V will be created using E (input information):

- $DW_i^Q = Q_i$ with dimensions $target\_length \times d_k$
- $EW_i^K = K_i$ with dimensions $target\_length \times d_k$
- $EW_i^V = V_i$ with dimensions $target\_length \times d_v$

for every head $i = 1, ..., h$
The matrix W_0 used after the concatenation of the heads will have dimensions $(d_v * h) \times (emb\_dim)$ just like the one used in the encoder block.
Apart from that, the Multi-Head Attention block is exactly the same as the one in the encoder.
In this case, the matrix resulting from $Softmax(\frac{Q_i K_i^T}{\sqrt{d_k}})$ is describing relevant relationships between "encoded input tokens" and "encoded target tokens".
Moreover notice that this matrix will have dimensions $(target\_length) \times (input\_length)$.

K.5 Linear and Softmax

This is the final step before being able to get the predicted token for every position in the target sequence.

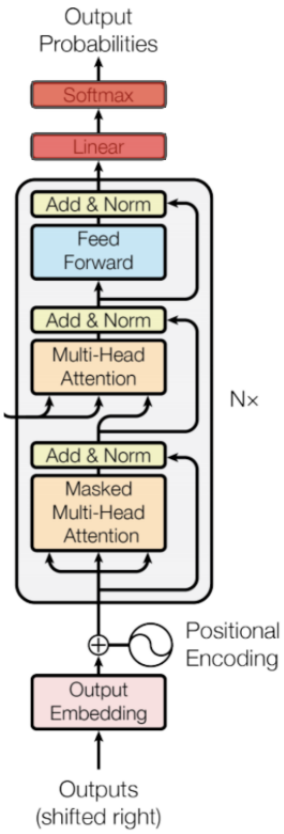


Figure 24: Transformer Decoder - Linear and Softmax

The output from the last Add & Norm layer of the last Decoder block is a matrix X of dimensions $(target\_length) \times (emb\_dim)$.
The idea of the linear layer is for every row in x of X to compute: xW_1 where W_1 is a matrix of learned weights of dimensions $(emb\_dim)x(vocab\_size)$. Therefore, for a specific row the result will be a vector of length $vocab\_size$.
Finally, a softmax is applied to this vector resulting in a vector describing the probability of the next token. Therefore, taking the position corresponding to the maximum probability returns the most likely next word according to the model.
In matrix form this looks like: XW_1 (and applying a Softmax in each resulting row).

The Illustrated GPT-2 Transformer

L The Illustrated GPT-2 (Visualizing Transformer Language Models)

This year, we saw a dazzling application of machine learning. The OpenAI GPT-2 exhibited impressive ability of writing coherent and passionate essays that exceed what we anticipated current language models are able to produce. The GPT-2 was not a particularly novel architecture – its architecture is very similar to the decoder-only transformer. The GPT2 was, however, a very large, transformer-based language model trained on a massive dataset. In this appendix adapted from a Jay Alammar post [106], we will look at the architecture that enabled the model to produce its results. We will go into the depths of its self-attention layer. And then we will look at applications for the decoder-only transformer beyond language modeling.

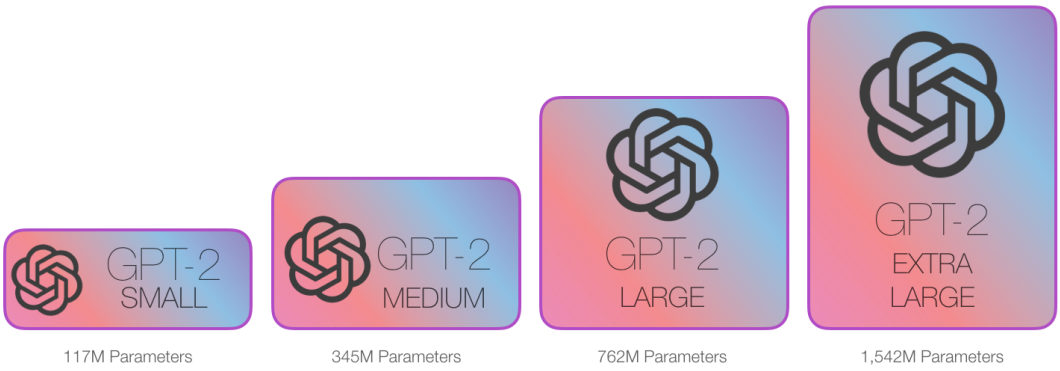
L.1 Part #1: GPT2 And Language Modeling

So what exactly is a language model?

L.1.1 What is a Language Model

A language model is basically a machine learning model that is able to look at part of a sentence and predict the next word. The most famous language models are smartphone keyboards that suggest the next word based on what you have currently typed.

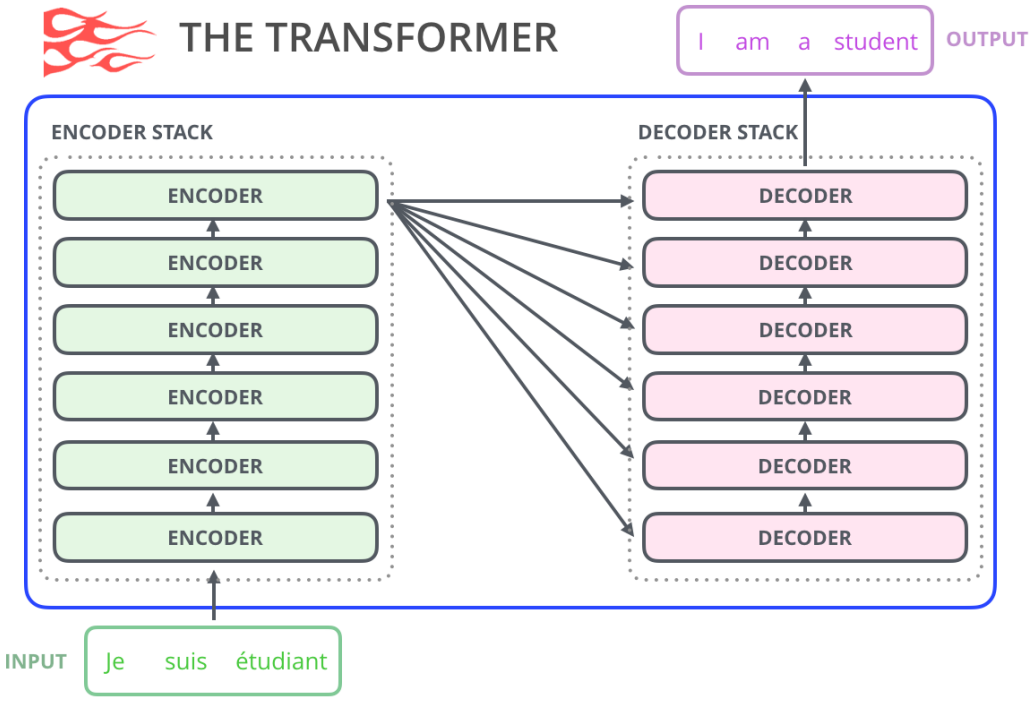
In this sense, we can say that the GPT-2 is basically the next word prediction feature of a keyboard app, but one that is much larger and more sophisticated than what your phone has. The GPT-2 was trained on a massive 40GB dataset called WebText that the OpenAI researchers crawled from the internet as part of the research effort. To compare in terms of storage size, the keyboard app I use, SwiftKey, takes up 78MBs of space. The smallest variant of the trained GPT-2, takes up 500MBs of storage to store all of its parameters. The largest GPT-2 variant is 13 times the size so it could take up more than 6.5 GBs of storage space.



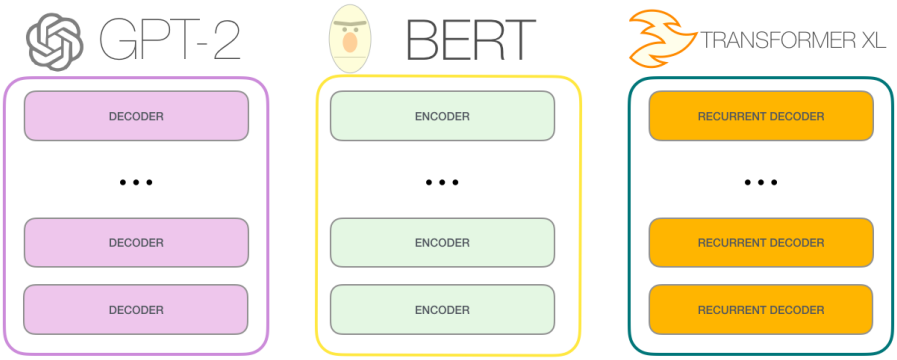
One great way to experiment with GPT-2 is using the AllenAI GPT-2 Explorer. It uses GPT-2 to display ten possible predictions for the next word (alongside their probability score). You can select a word then see the next list of predictions to continue writing the passage.

L.1.2 Transformers for Language Modeling

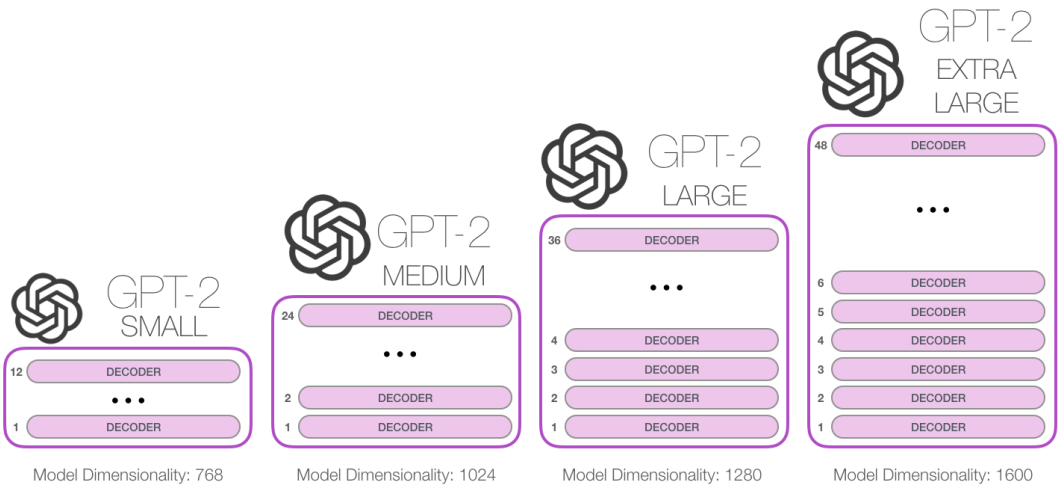
As it described in The Illustrated Transformer, the original transformer model is made up of an encoder and decoder – each is a stack of what we can call transformer blocks. That architecture was appropriate because the model tackled machine translation – a problem where encoder-decoder architectures have been successful in the past.



A lot of the subsequent research work saw the architecture shed either the encoder or decoder, and use just one stack of transformer blocks – stacking them up as high as practically possible, feeding them massive amounts of training text, and throwing vast amounts of compute at them (hundreds of thousands of dollars to train some of these language models, likely millions in the case of AlphaStar).



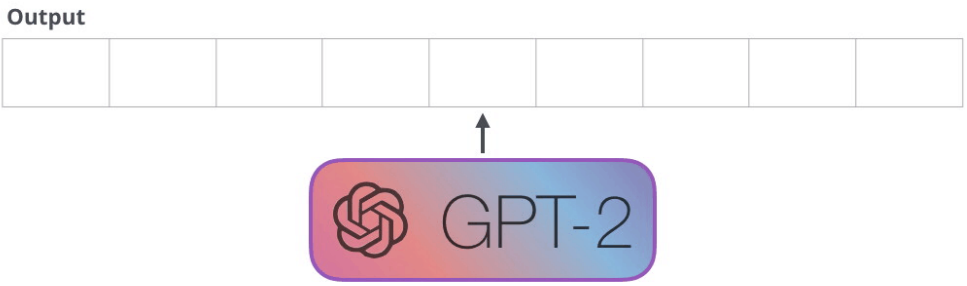
How high can we stack up these blocks? It turns out that is one of the main distinguishing factors between the different GPT2 model sizes:



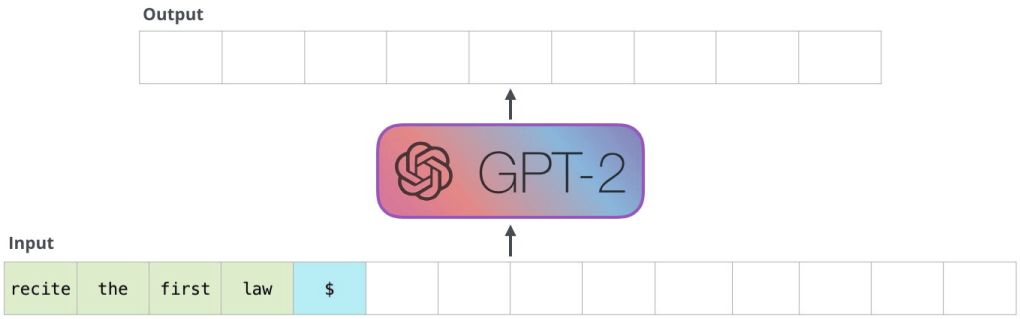
L.1.3 One Difference From BERT

> **First Law of Robotics**
> A robot may not injure a human being or, through inaction, allow a human being to come to harm.

The GPT-2 is built using transformer decoder blocks. BERT, on the other hand, uses transformer encoder blocks. We will examine the difference in a following section. But one key difference between the two is that GPT2, like traditional language models, outputs one token at a time. Let us for example prompt a well-trained GPT-2 to recite the first law of robotics:



The way these models actually work is that after each token is produced, that token is added to the sequence of inputs. And that new sequence becomes the input to the model in its next step. This is an idea called “auto-regression”. This is one of the ideas that made RNNs unreasonably effective.

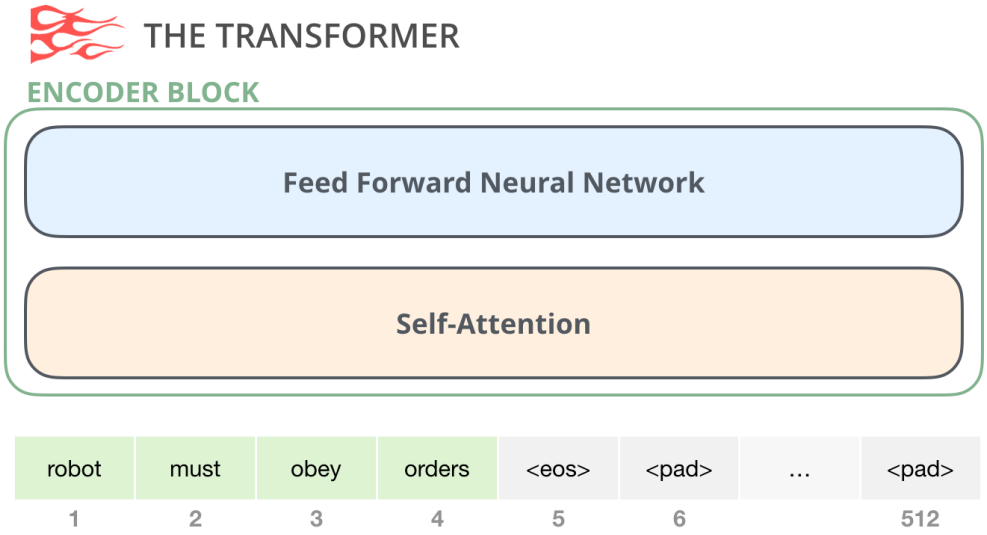


The GPT2, and some later models like Transformer-XL [107] and XLNet [42] are auto-regressive in nature. BERT is not. That is a trade off. In losing auto-regression, BERT gained the ability to incorporate the context on both sides of a word to gain better results. XLNet brings back autoregression while finding an alternative way to incorporate the context on both sides.

L.1.4 The Evolution of the Transformer Block

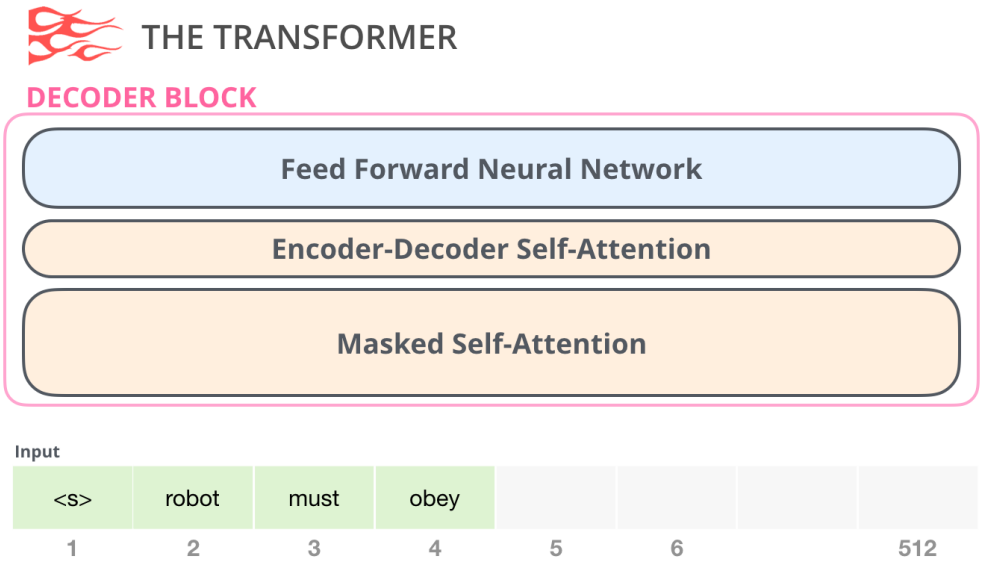
The initial transformer paper introduced two types of transformer blocks:

L.1.4.1 The Encoder Block First is the encoder block:



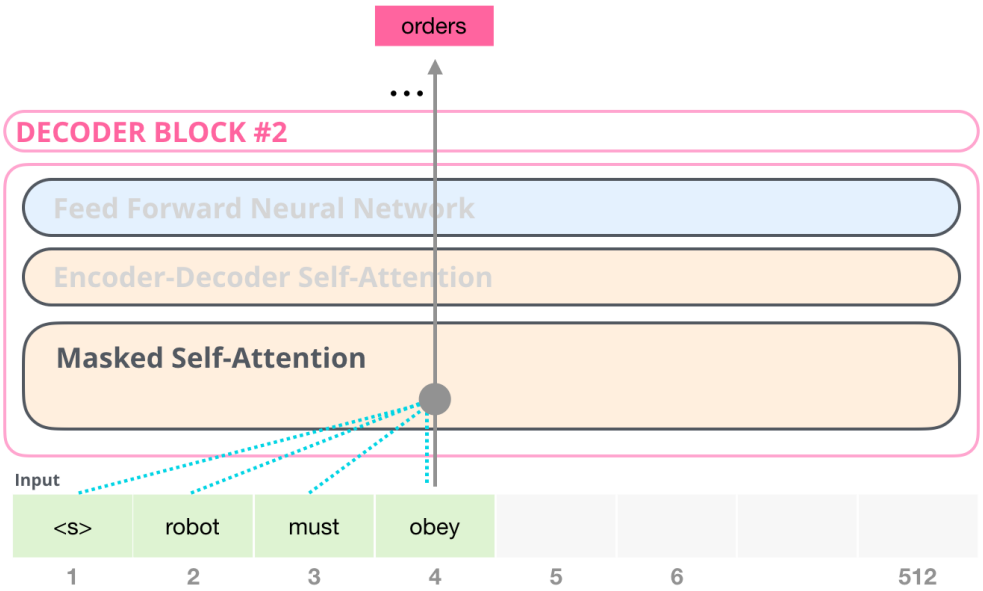
An encoder block from the original transformer paper can take inputs up until a certain max sequence length (e.g. 512 tokens). It is okay if an input sequence is shorter than this limit, we can just pad the rest of the sequence.

L.1.4.2 The Decoder Block Second, there is the decoder block which has a small architectural variation from the encoder block – a layer to allow it to pay attention to specific segments from the encoder:

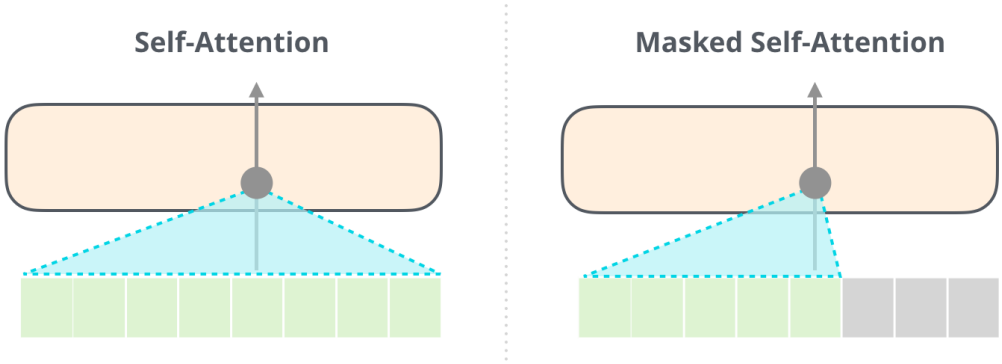


One key difference in the self-attention layer here, is that it masks future tokens – not by changing the word to [mask] like BERT, but by interfering in the self-attention calculation blocking information from tokens that are to the right of the position being calculated.

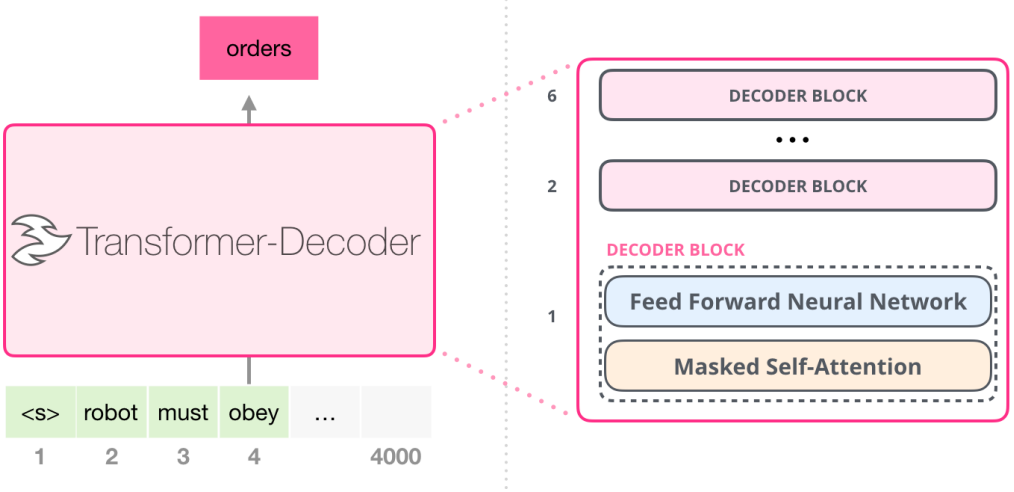
If, for example, we are to highlight the path of position #4, we can see that it is only allowed to attend to the present and previous tokens:



It is important that the distinction between self-attention (what BERT uses) and masked self-attention (what GPT-2 uses) is clear. A normal self-attention block allows a position to peak at tokens to its right. Masked self-attention prevents that from happening:



L.1.4.3 The Decoder-Only Block Subsequent to the original paper, Generating Wikipedia by Summarizing Long Sequences proposed another arrangement of the transformer block that is capable of doing language modeling. This model threw away the Transformer encoder. For that reason, let us call the model the "Transformer-Decoder". This early transformer-based language model was made up of a stack of six transformer decoder blocks:



The decoder blocks are identical. I have expanded the first one so you can see its self-attention layer is the masked variant. Notice that the model now can address up to 4,000 tokens in a certain segment -- a massive upgrade from the 512 in the original transformer.

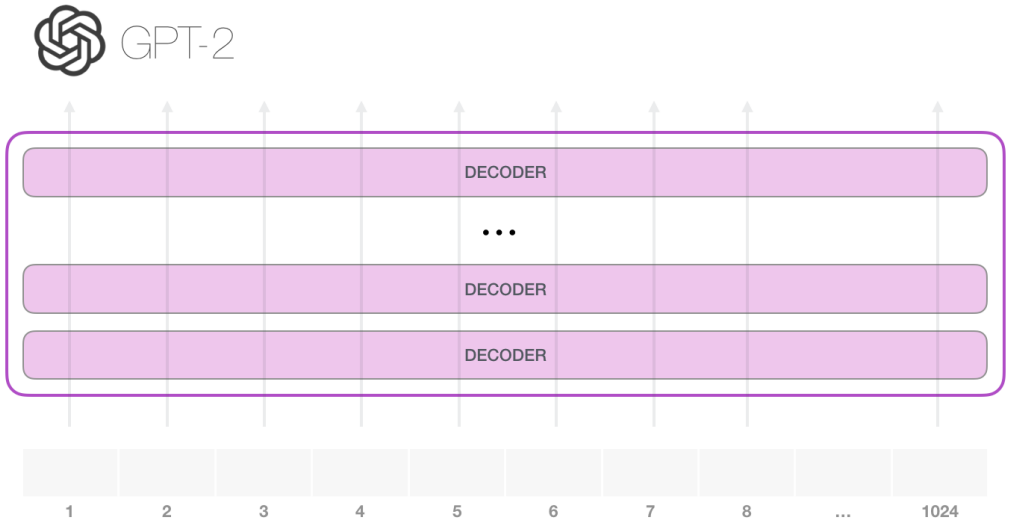
These blocks were very similar to the original decoder blocks, except they did away with that second self-attention layer. A similar architecture was examined in Character-Level Language Modeling with Deeper Self-Attention to create a language model that predicts one letter/character at a time.

The OpenAI GPT-2 model uses these decoder-only blocks.

L.1.5 Crash Course in Brain Surgery: Looking Inside GPT-2

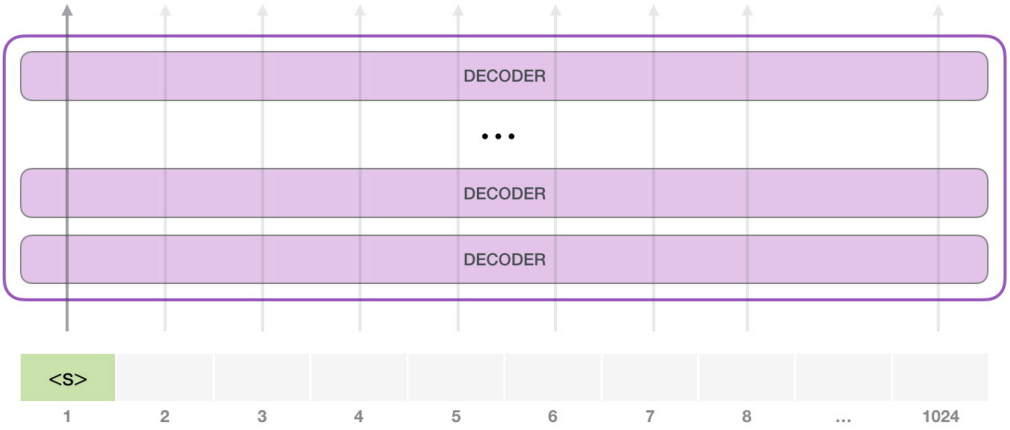
> Look inside and you will see, The words are cutting deep inside my brain. Thunder burning, quickly burning, Knife of words is driving me insane, insane yeah. ~**Budgie**

Let us lay a trained GPT-2 on our surgery table and look at how it works.



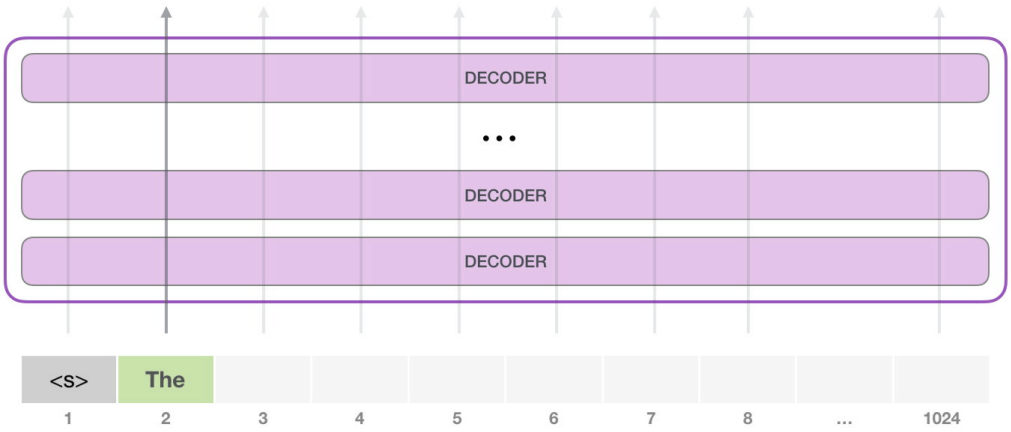
The GPT-2 can process 1024 tokens. Each token flows through all the decoder blocks along its own path.

The simplest way to run a trained GPT-2 is to allow it to ramble on its own (which is technically called *generating unconditional samples*) – alternatively, we can give it a prompt to have it speak about a certain topic (a.k.a generating *interactive conditional samples*). In the rambling case, we can simply hand it the start token and have it start generating words (the trained model uses `<|endoftext|>` as its start token. Let us call it `<s>` instead).



The model only has one input token, so that path would be the only active one. The token is processed successively through all the layers, then a vector is produced along that path. That vector can be scored against the model's vocabulary (all the words the model knows, 50,000 words in the case of GPT-2). In this case we selected the token with the highest probability, 'the'. But we can certainly mix things up – you know how if you keep clicking the suggested word in your keyboard app, it sometimes can stuck in repetitive loops where the only way out is if you click the second or third suggested word. The same can happen here. GPT-2 has a parameter called top-k that we can use to have the model consider sampling words other than the top word (which is the case when top-k = 1).

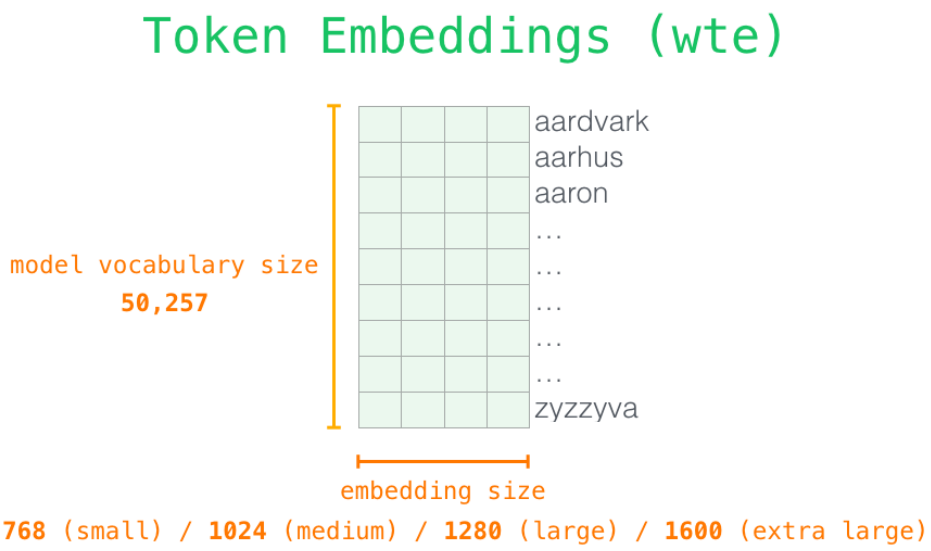
In the next step, we add the output from the first step to our input sequence, and have the model make its next prediction:



Notice that the second path is the only that is active in this calculation. Each layer of GPT-2 has retained its own interpretation of the first token and will use it in processing the second token (we will get into more detail about this in the following section about self-attention). GPT-2 does not re-interpret the first token in light of the second token.

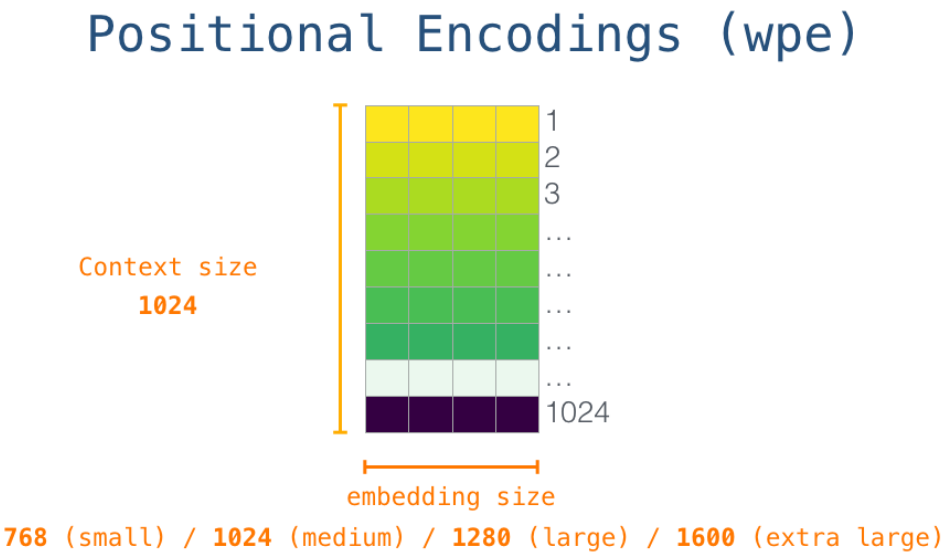
L.1.6 A Deeper Look Inside

L.1.6.1 Input Encoding Let us look at more details to get to know the model more intimately. Let us start from the input. As in other NLP models we have discussed before, the model looks up the embedding of the input word in its embedding matrix – one of the components we get as part of a trained model.



Each row is a word embedding: a list of numbers representing a word and capturing some of its meaning. The size of that list is different in different GPT2 model sizes. The smallest model uses an embedding size of 768 per word/token.

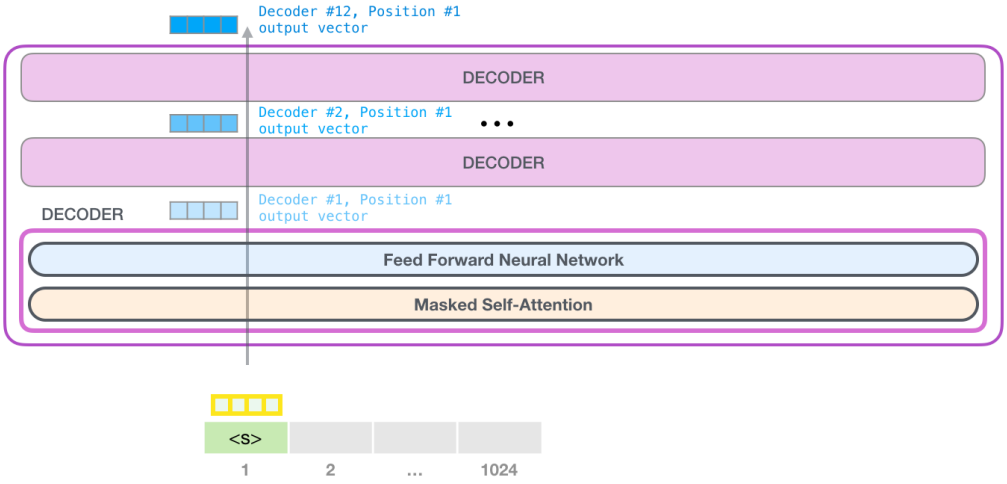
So in the beginning, we look up the embedding of the start token <s> in the embedding matrix. Before handing that to the first block in the model, we need to incorporate positional encoding – a signal that indicates the order of the words in the sequence to the transformer blocks. Part of the trained model is a matrix that contains a positional encoding vector for each of the 1024 positions in the input.



With this, we have covered how input words are processed before being handed to the first transformer block. We also know two of the weight matrices that constitute the trained GPT-2.

Sending a word to the first transformer block means looking up its embedding and adding up the positional encoding vector for position #1.

L.1.6.2 A journey up the Stack The first block can now process the token by first passing it through the self-attention process, then passing it through its neural network layer. Once the first transformer block processes the token, it sends its resulting vector up the stack to be processed by the next block. The process is identical in each block, but each block has its own weights in both self-attention and the neural network sublayers.



L.1.6.3 Self-Attention Recap Language heavily relies on context. For example, look at the second law:

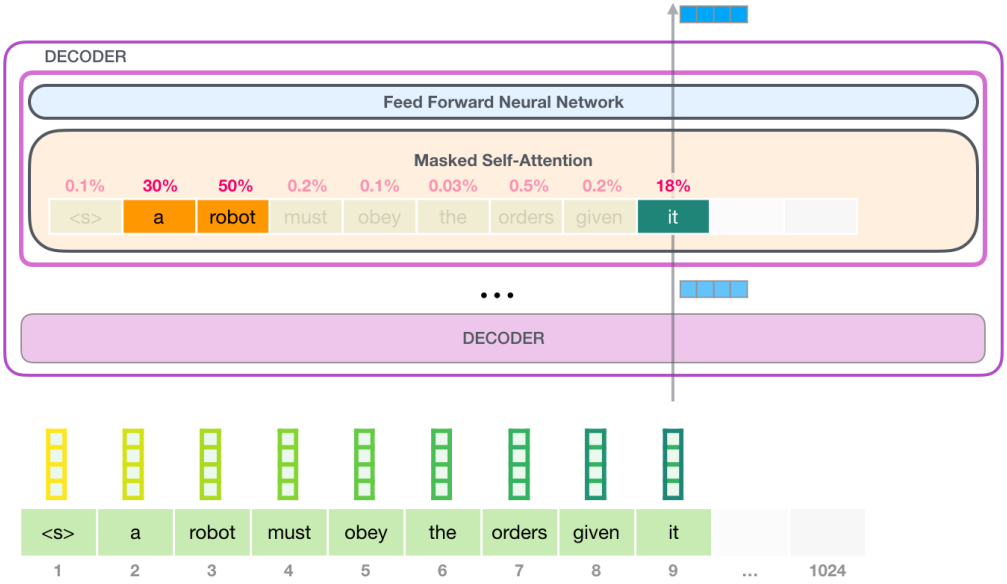
> **Second Law of Robotics**
> A robot must obey the orders given **it** by human beings except where **such orders** would conflict with the **First Law**.

I have highlighted three places in the sentence where the words are referring to other words. There is no way to understand or process these words without incorporating the context

they are referring to. When a model processes this sentence, it has to be able to know that:

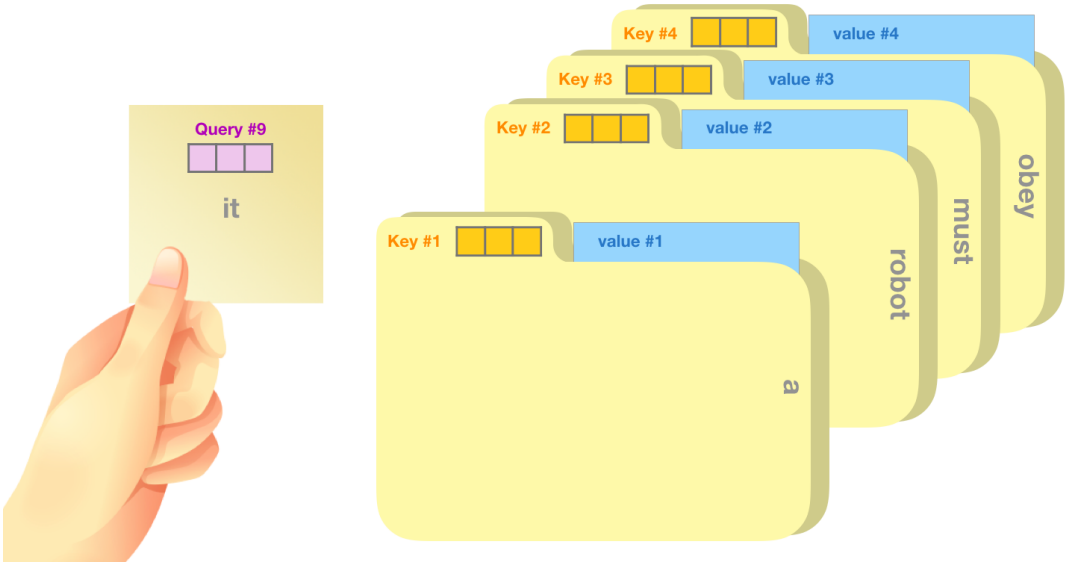
- **it** refers to the robot
- **such orders** refers to the earlier part of the law, namely "the orders given it by human beings"
- **The First Law** refers to the entire First Law

This is what self-attention does. It bakes in the model's understanding of relevant and associated words that explain the context of a certain word before processing that word (passing it through a neural network). It does that by assigning scores to how relevant each word in the segment is, and adding up their vector representation.
As an example, this self-attention layer in the top block is paying attention to "a robot" when it processes the word "it". The vector it will pass to its neural network is a sum of the vectors for each of the three words multiplied by their scores.



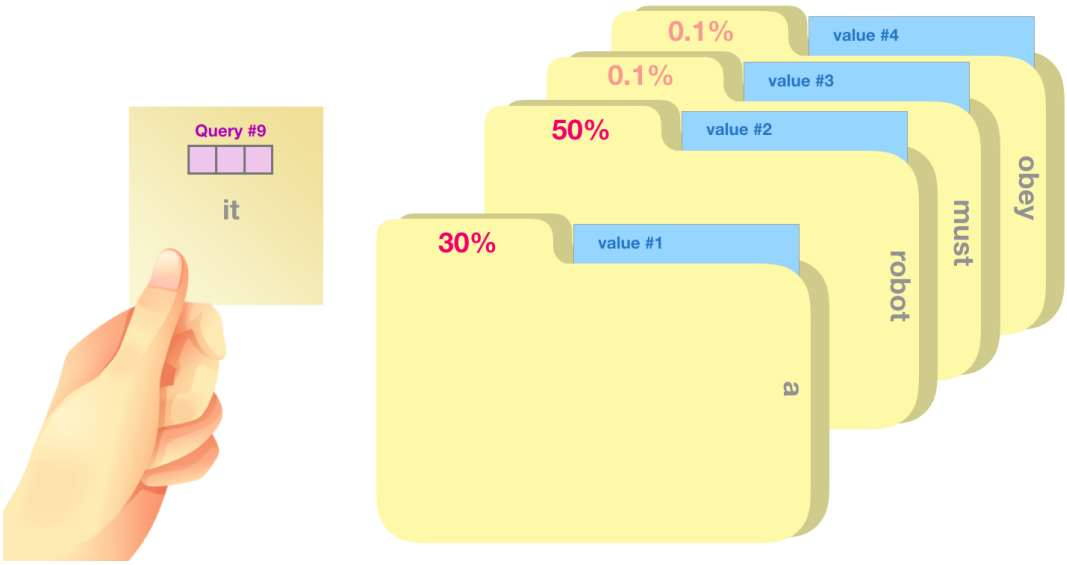
L.1.6.4 Self-Attention Process Self-attention is processed along the path of each token in the segment. The significant components are three vectors:

- Query: The query is a representation of the current word used to score against all the other words (using their keys). We only care about the query of the token we are currently processing.
- Key: Key vectors are like labels for all the words in the segment. They are what we match against in our search for relevant words.
- Value: Value vectors are actual word representations, once we have scored how relevant each word is, these are the values we add up to represent the current word.



A crude analogy is to think of it like searching through a filing cabinet. The query is like a sticky note with the topic you are researching. The keys are like the labels of the folders inside the cabinet. When you match the tag with a sticky note, we take out the contents of that folder, these contents are the value vector. Except you are not only looking for one value, but a blend of values from a blend of folders.

Multiplying the query vector by each key vector produces a score for each folder (technically: dot product followed by softmax).

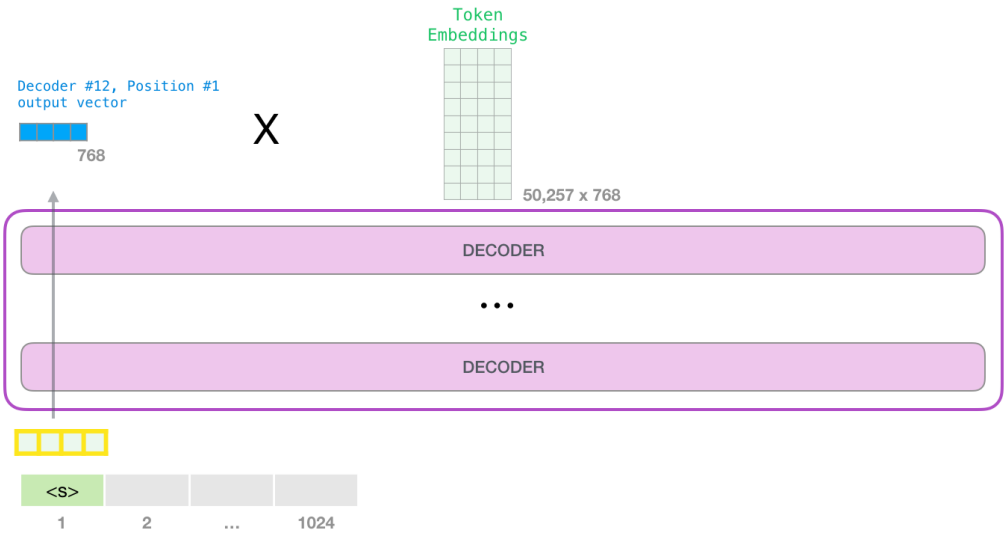


We multiply each value by its score and sum up – resulting in our self-attention outcome.

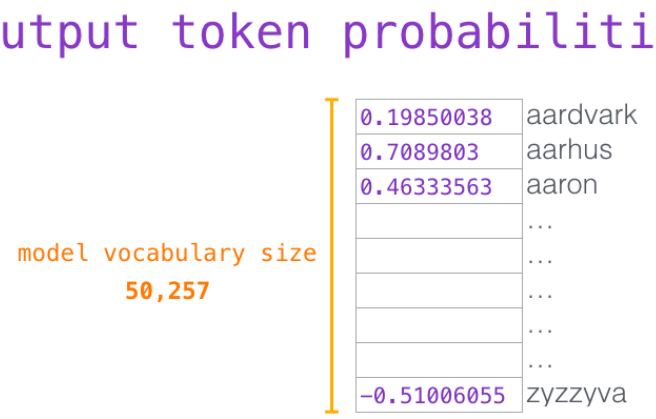
Word	Value vector	Score	Value X Score
<s>		0.001	
a		0.3	
robot		0.5	
must		0.002	
obey		0.001	
the		0.0003	
orders		0.005	
given		0.002	
it		0.19	
		Sum:	

This weighted blend of value vectors results in a vector that paid 50% of its "attention" to the word `robot`, 30% to the word `a`, and 19% to the word `it`. Later in the post, we will got deeper into self-attention. But first, let us continue our journey up the stack towards the output of the model.

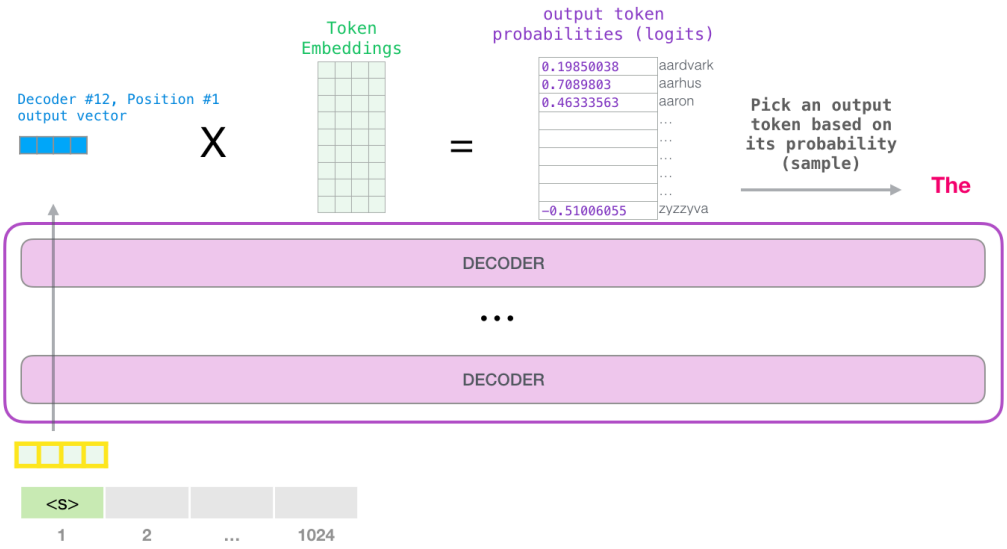
L.1.6.5 Model Output When the top block in the model produces its output vector (the result of its own self-attention followed by its own neural network), the model multiplies that vector by the embedding matrix.



Recall that each row in the embedding matrix corresponds to the embedding of a word in the model's vocabulary. The result of this multiplication is interpreted as a score for each word in the model's vocabulary.



We can simply select the token with the highest score (top_k = 1). But better results are achieved if the model considers other words as well. So a better strategy is to sample a word from the entire list using the score as the probability of selecting that word (so words with a higher score have a higher chance of being selected). A middle ground is setting top_k to 40, and having the model consider the 40 words with the highest scores.



With that, the model has completed an iteration resulting in outputting a single word. The model continues iterating until the entire context is generated (1024 tokens) or until an end-of-sequence token is produced.

L.1.7 End of part #1: The GPT-2, Ladies and Gentlemen

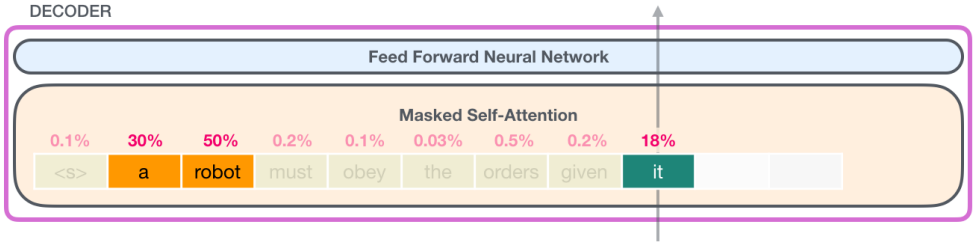
And there we have it. A run down of how the GPT2 works. If you are curious to know exactly what happens inside the self-attention layer, then the following bonus section is for you. I created it to introduce more visual language to describe self-attention in order to make describing later transformer models easier to examine and describe (looking at you, TransformerXL and XLNet).

Below are a few oversimplifications in this section:

- I used "words" and "tokens" interchangeably. But in reality, GPT2 uses Byte Pair Encoding to create the tokens in its vocabulary. This means the tokens are usually parts of words.
- The example we showed runs GPT2 in its inference/evaluation mode. That is why it is only processing one word at a time. At training time, the model would be trained against longer sequences of text and processing multiple tokens at once. Also at training time, the model would process larger batch sizes (512) vs. the batch size of one that evaluation uses.
- I took liberties in rotating/transposing vectors to better manage the spaces in the images. At implementation time, one has to be more precise.
- Transformers use a lot of layer normalization, which is pretty important. We have noted a few of these in the Illustrated Transformer, but focused more on self-attentionin this post.
- There are times when I needed to show more boxes to represent a vector. I indicate those as "zooming in". For example:

L.2 Part #2: The Illustrated Self-Attention

Earlier in the post we showed this image to showcase self-attention being applied in a layer that is processing the word `it`:



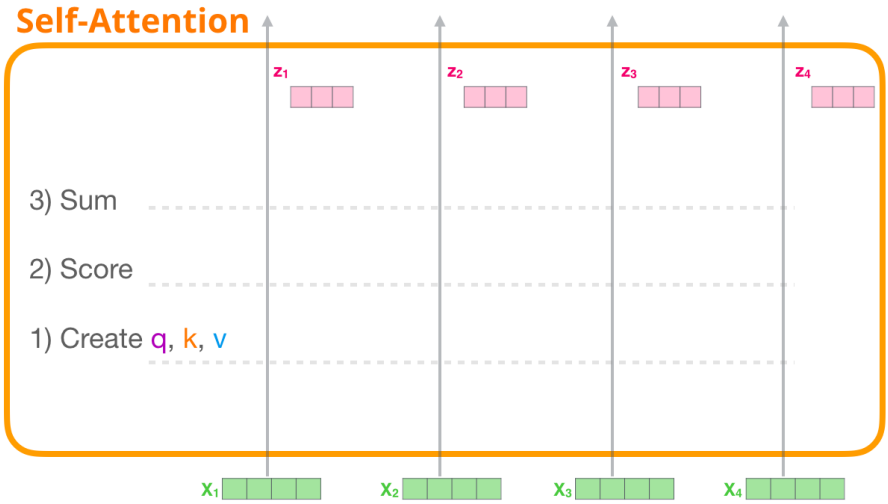
In this section, we will look at the details of how that is done. Note that we will look at it in a way to try to make sense of what happens to individual words. That is why we will be showing many single vectors. The actual implementations are done by multiplying giant matrices together. But I want to focus on the intuition of what happens on a word-level here.

L.2.1 Self-Attention (without masking)

Let us start by looking at the original self-attention as it is calculated in an encoder block. Let us look at a toy transformer block that can only process four tokens at a time.

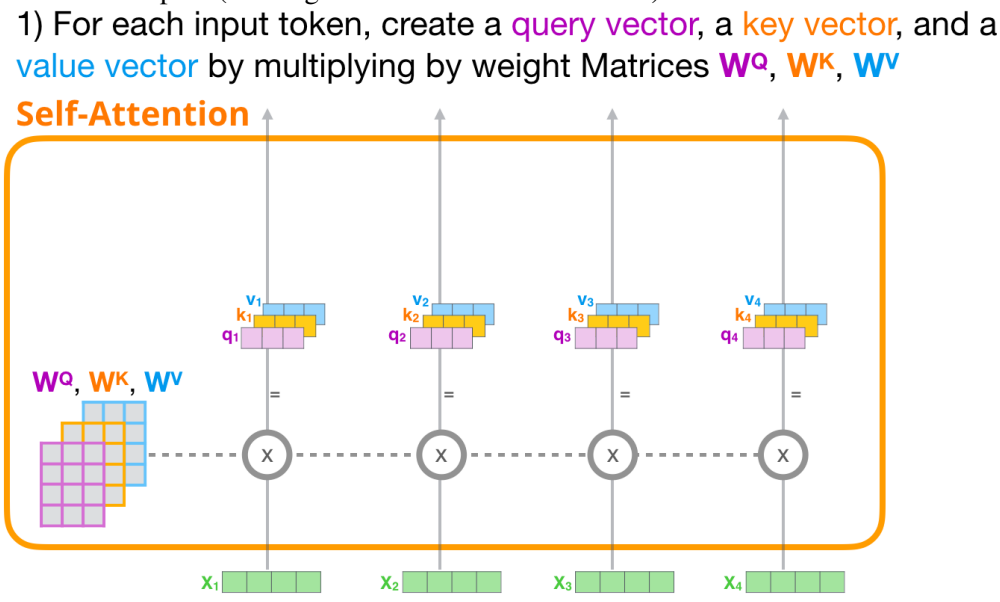
Self-attention is applied through three main steps:

1. Create the Query, Key, and Value vectors for each path.
2. For each input token, use its query vector to score against all the other key vectors
3. Sum up the value vectors after multiplying them by their associated scores.



L.2.2 1- Create Query, Key, and Value Vectors

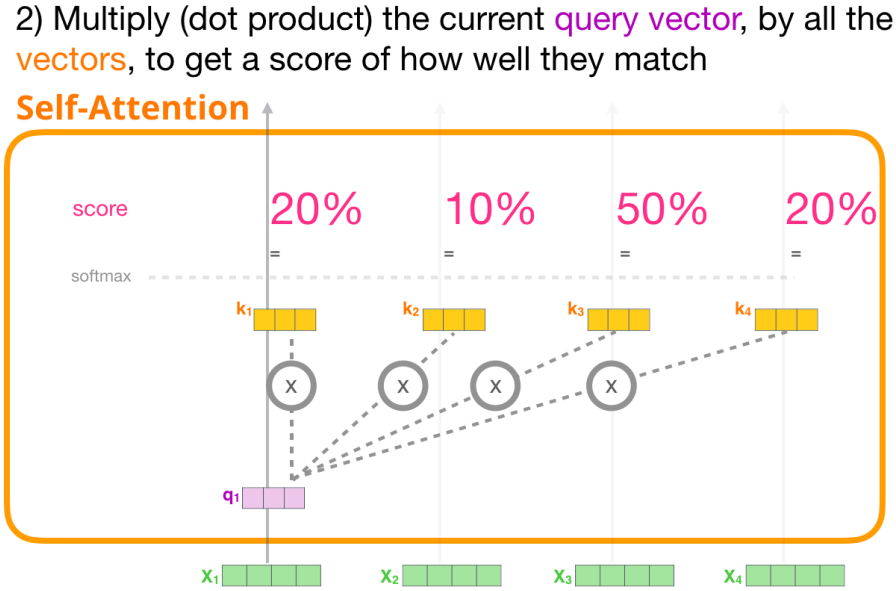
Let us focus on the first path. We will take its query, and compare against all the keys. That produces a score for each key. The first step in self-attention is to calculate the three vectors for each token path (let us ignore attention heads for now):



L.2.3 2- Score

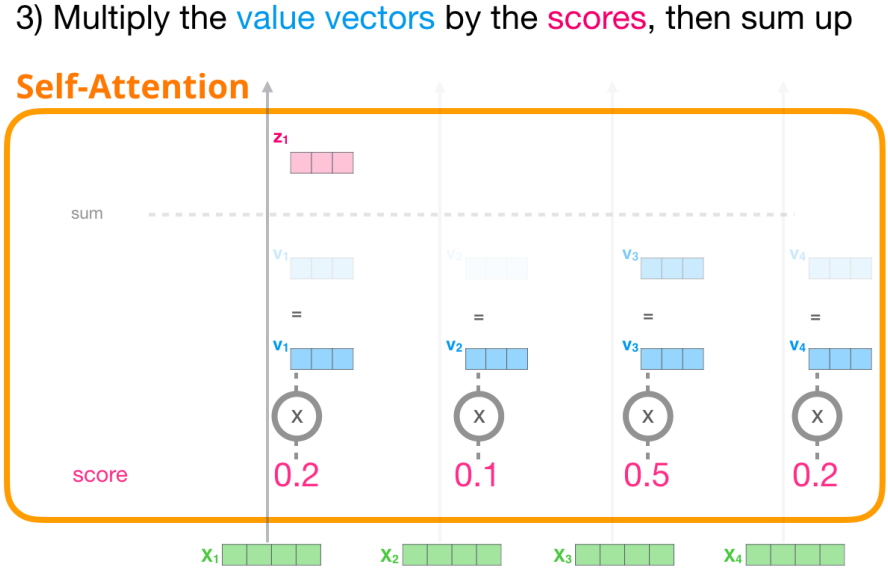
Now that we have the vectors, we use the query and key vectors only for step #2. Since we are focused on the first token, we multiply its query by all the other key vectors resulting in

a score for each of the four tokens.



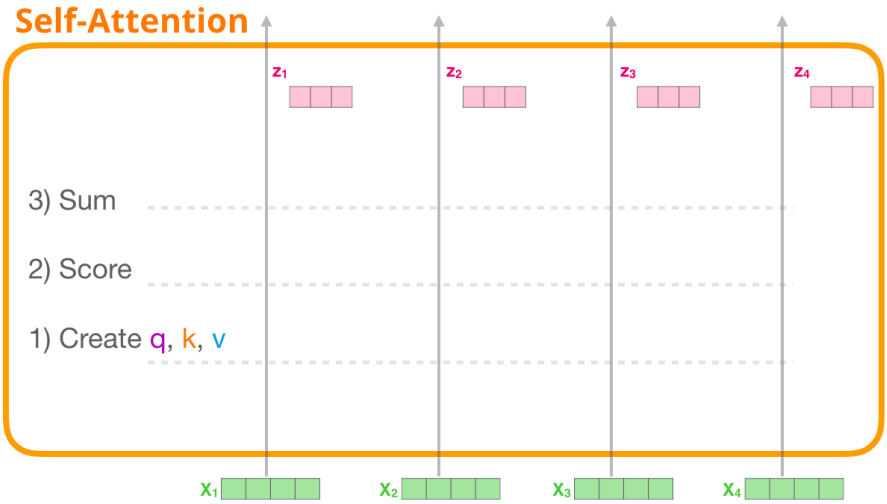
L.2.4 3- Sum

We can now multiply the scores by the value vectors. A value with a high score will constitute a large portion of the resulting vector after we sum them up.



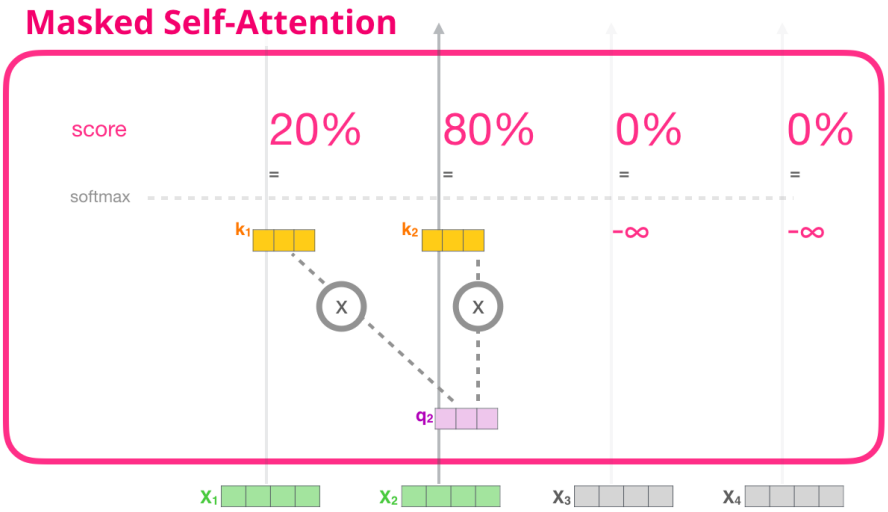
The lower the score, the more transparent we are showing the value vector. That is to indicate how multiplying by a small number dilutes the values of the vector.

If we do the same operation for each path, we end up with a vector representing each token containing the appropriate context of that token. Those are then presented to the next sublayer in the transformer block (the feed-forward neural network):



L.2.5 The Illustrated Masked Self-Attention

Now that we have looked inside a transformer's self-attention step, let us proceed to look at masked self-attention. Masked self-attention is identical to self-attention except when it comes to step #2. Assuming the model only has two tokens as input and we are observing the second token. In this case, the last two tokens are masked. So the model interferes in the scoring step. It basically always scores the future tokens as 0 so the model cannot peak to future words:



This masking is often implemented as a matrix called an attention mask. Think of a sequence of four words ("robot must obey orders", for example). In a language modeling scenario, this sequence is absorbed in four steps – one per word (assuming for now that every word is a token). As these models work in batches, we can assume a batch size of 4 for this toy model that will process the entire sequence (with its four steps) as one batch.

	Features				Labels
	position: 1	2	3	4	
Example: 1	robot	must	obey	orders	must
2	robot	must	obey	orders	obey
3	robot	must	obey	orders	orders
4	robot	must	obey	orders	<eos>

In matrix form, we calculate the scores by multiplying a queries matrix by a keys matrix. Let us visualize it as follows, except instead of the word, there would be the query (or key) vector associated with that word in that cell:

Queries					Keys					Scores (before softmax)			
robot	must	obey	orders	X	robot	must	obey	orders	=	0.11	0.00	0.81	0.79
					robot	must	obey	orders		0.19	0.50	0.30	0.48
					robot	must	obey	orders		0.53	0.98	0.95	0.14
					robot	must	obey	orders		0.81	0.86	0.38	0.90

After the multiplication, we slap on our attention mask triangle. It set the cells we want to mask to -infinity or a very large negative number (e.g. -1 billion in GPT2):

Scores (before softmax)				Apply Attention Mask	Masked Scores (before softmax)			
0.11	0.00	0.81	0.79	→	0.11	-inf	-inf	-inf
0.19	0.50	0.30	0.48		0.19	0.50	-inf	-inf
0.53	0.98	0.95	0.14		0.53	0.98	0.95	-inf
0.81	0.86	0.38	0.90		0.81	0.86	0.38	0.90

Then, applying softmax on each row produces the actual scores we use for self-attention:

Masked Scores (before softmax)				Softmax (along rows)	Scores			
0.11	-inf	-inf	-inf	→	1	0	0	0
0.19	0.50	-inf	-inf		0.48	0.52	0	0
0.53	0.98	0.95	-inf		0.31	0.35	0.34	0
0.81	0.86	0.38	0.90		0.25	0.26	0.23	0.26

What this scores table means is the following:

- When the model processes the first example in the dataset (row #1), which contains only one word (“robot”), 100% of its attention will be on that word.

- When the model processes the second example in the dataset (row #2), which contains the words (“robot must”), when it processes the word “must”, 48% of its attention will be on “robot”, and 52% of its attention will be on “must”.

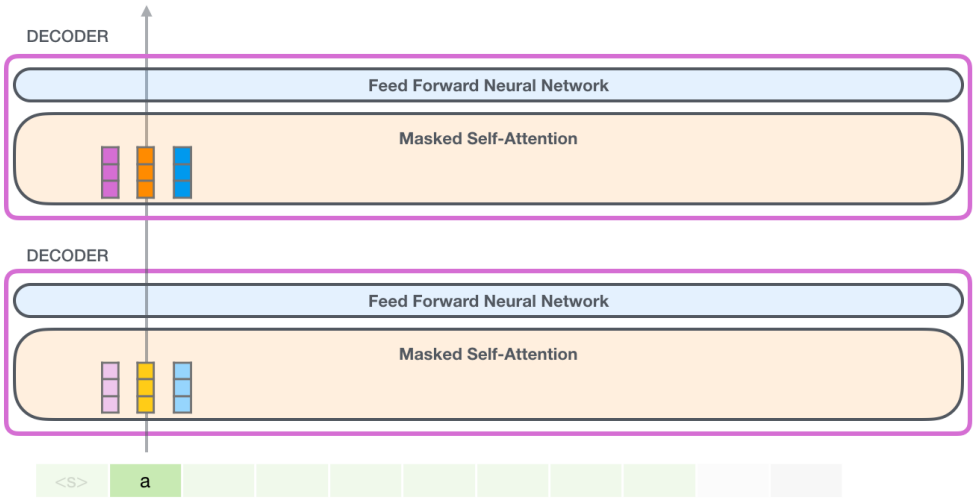
- And so on

L.2.6 GPT-2 Masked Self-Attention

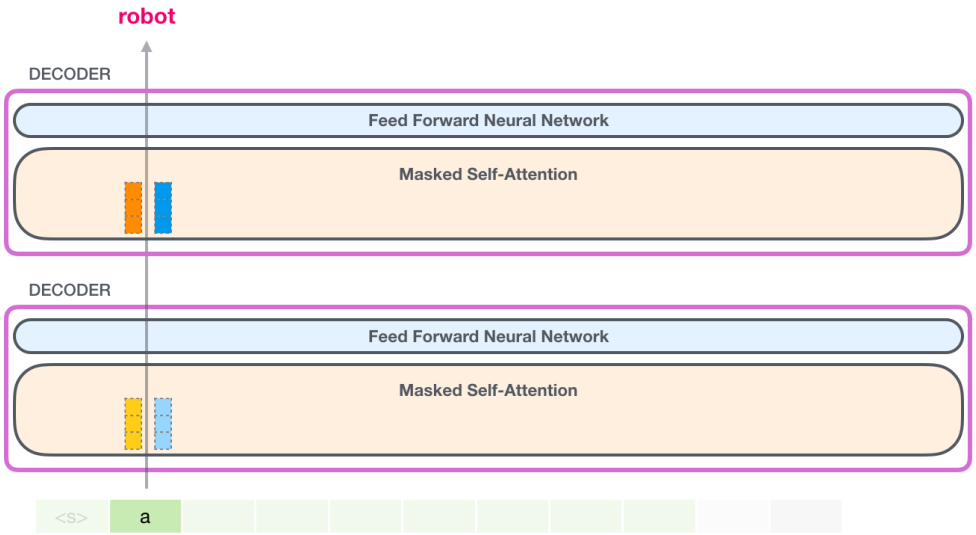
Let us get into more detail on GPT-2’s masked attention.

L.2.6.1 Evaluation Time: Processing One Token at a Time We can make the GPT-2 operate exactly as masked self-attention works. But during evaluation, when our model is only adding one new word after each iteration, it would be inefficient to recalculate self-attention along earlier paths for tokens which have already been processed.

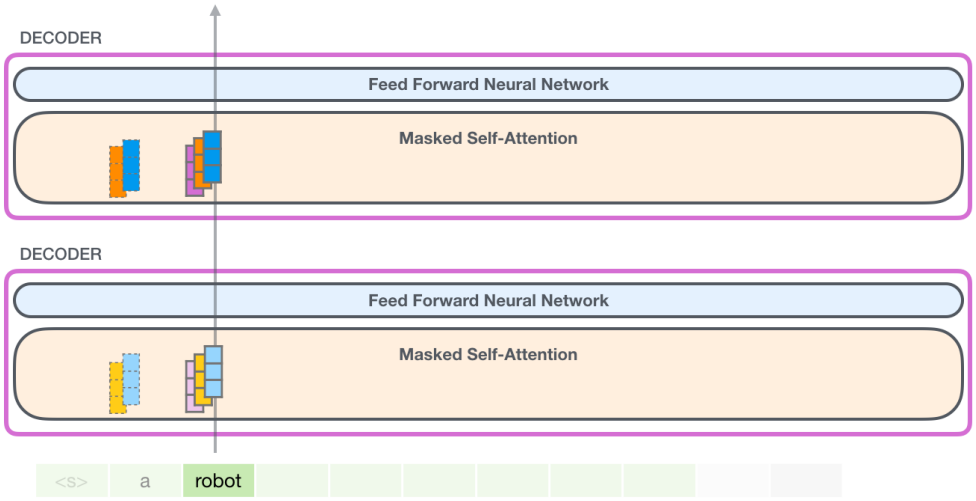
In this case, we process the first token (ignoring `<s>` for now).



GPT-2 holds on to the key and value vectors of the the `a` token. Every self-attention layer holds on to its respective key and value vectors for that token:

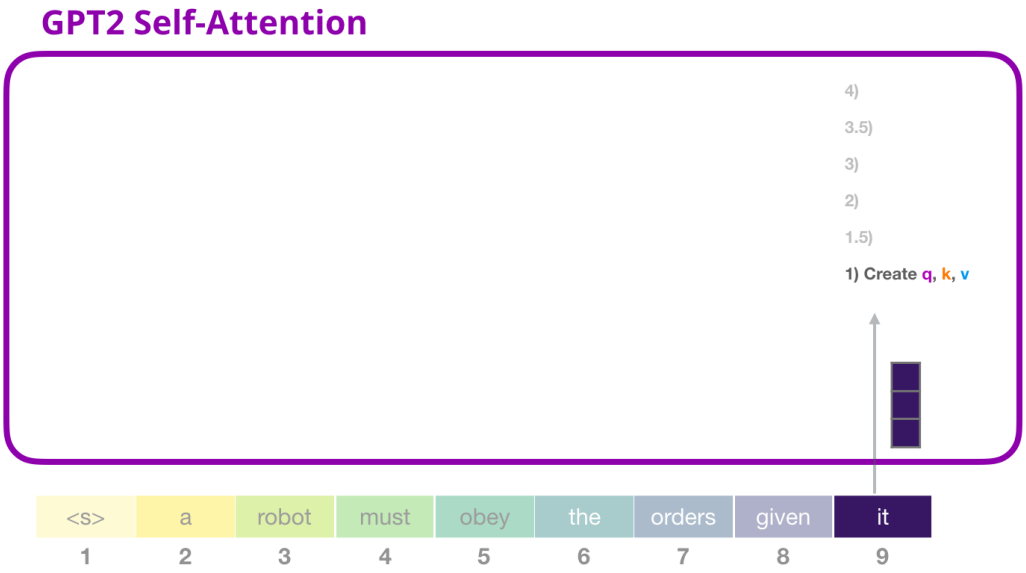


Now in the next iteration, when the model processes the word `robot`, it does not need to generate query, key, and value queries for the `a` token. It just reuses the ones it saved from the first iteration:

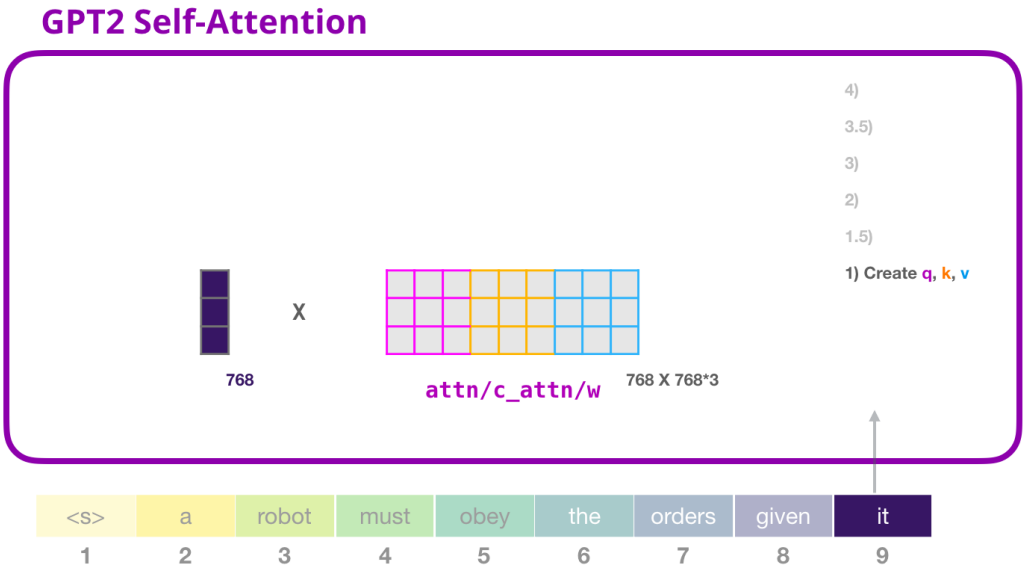


L.2.6.2 GPT-2 Self-attention: 1- Creating queries, keys, and values

Let us assume the model is processing the word `it`. If we are talking about the bottom block, then its input for that token would be the embedding of `it` + the positional encoding for slot #9:

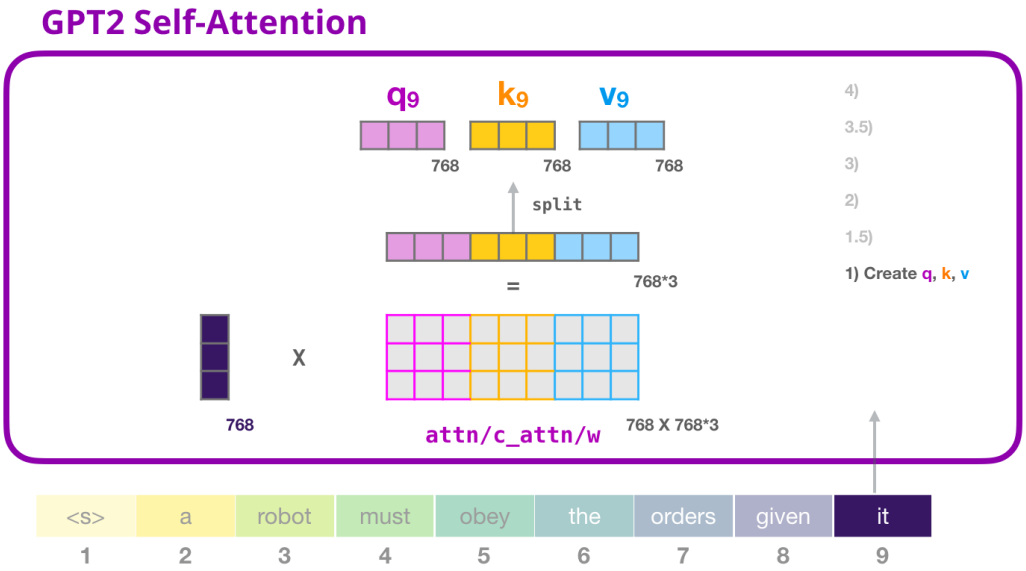


Every block in a transformer has its own weights (broken down later in the post). The first we encounter is the weight matrix that we use to create the queries, keys, and values.



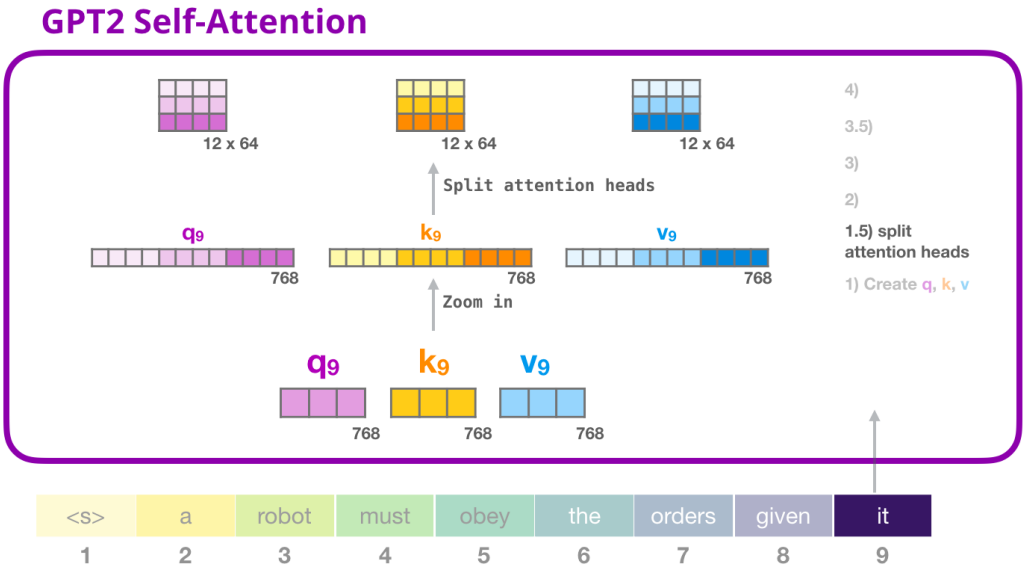
Self-attention multiplies its input by its weight matrix (and adds a bias vector, not illustrated here).

The multiplication results in a vector that is basically a concatenation of the query, key, and value vectors for the word `it`.

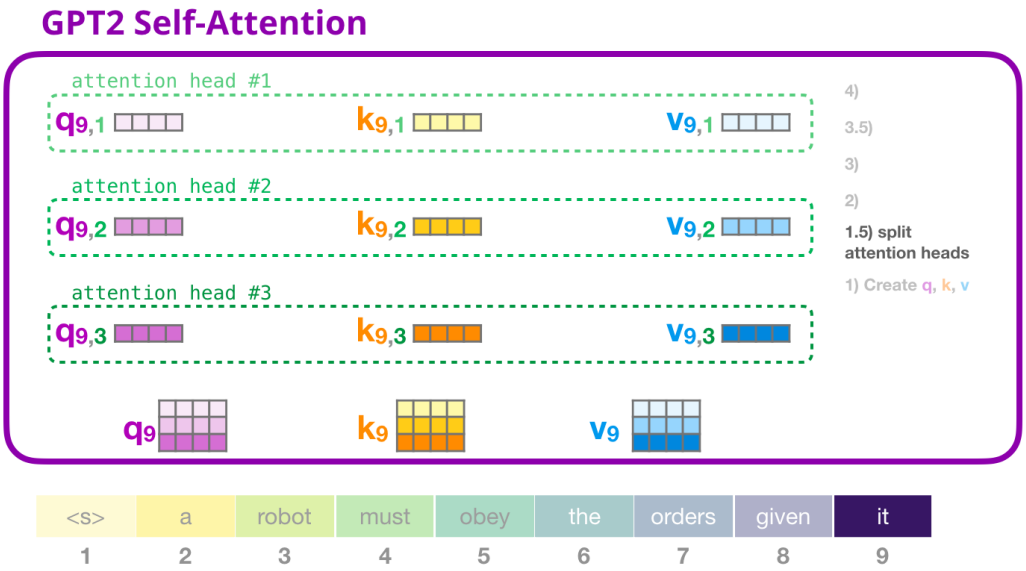


Multiplying the input vector by the attention weights vector (and adding a bias vector aftwards) results in the key, value, and query vectors for this token.

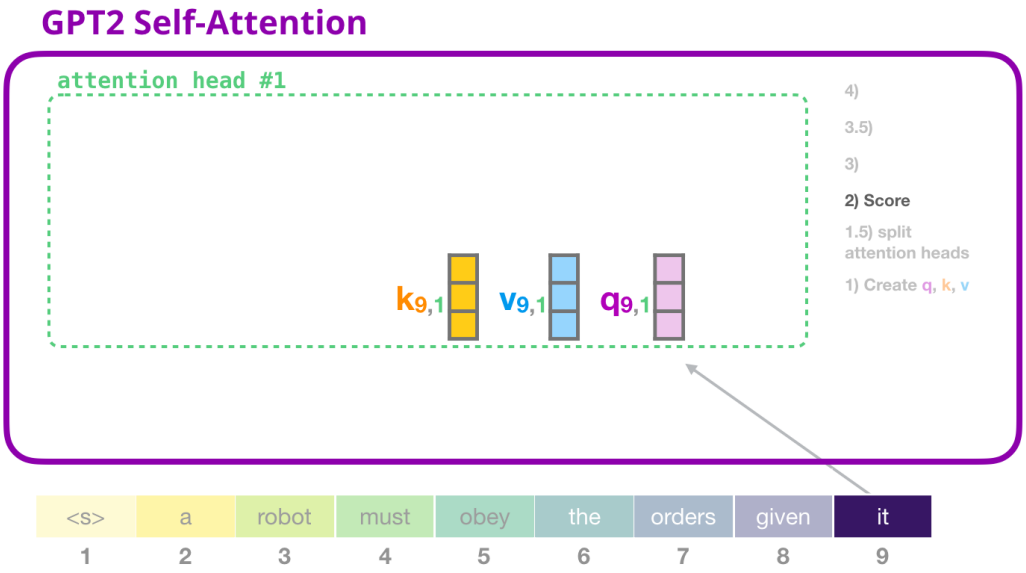
L.2.6.3 GPT-2 Self-attention: 1.5- Splitting into attention heads In the previous examples, we dove straight into self-attention ignoring the "multi-head" part. It would be useful to shed some light on that concept now. Self attention is conducted multiple times on different parts of the Q,K,V vectors. "Splitting" attention heads is simply reshaping the long vector into a matrix. The small GPT2 has 12 attention heads, so that would be the first dimension of the reshaped matrix:



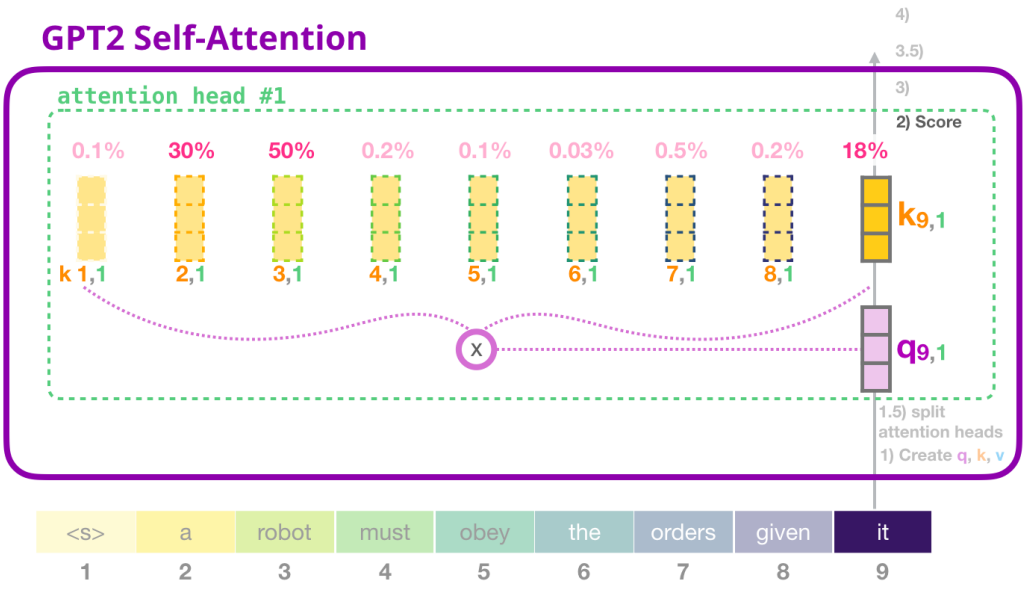
In the previous examples, we have looked at what happens inside one attention head. One way to think of multiple attention-heads is like this (if we are to only visualize three of the twelve attention heads):



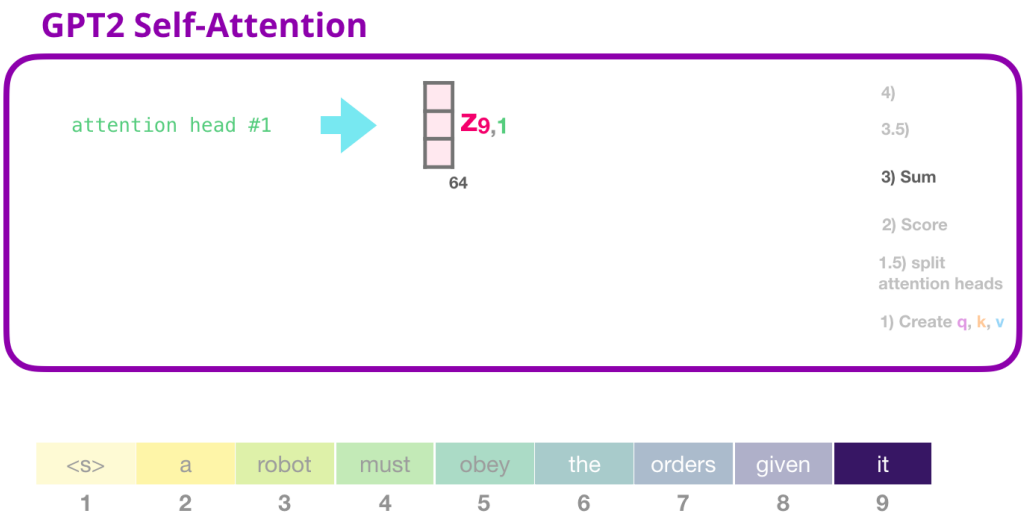
L.2.6.4 GPT-2 Self-attention: 2- Scoring We can now proceed to scoring – knowing that we are only looking at one attention head (and that all the others are conducting a similar operation):



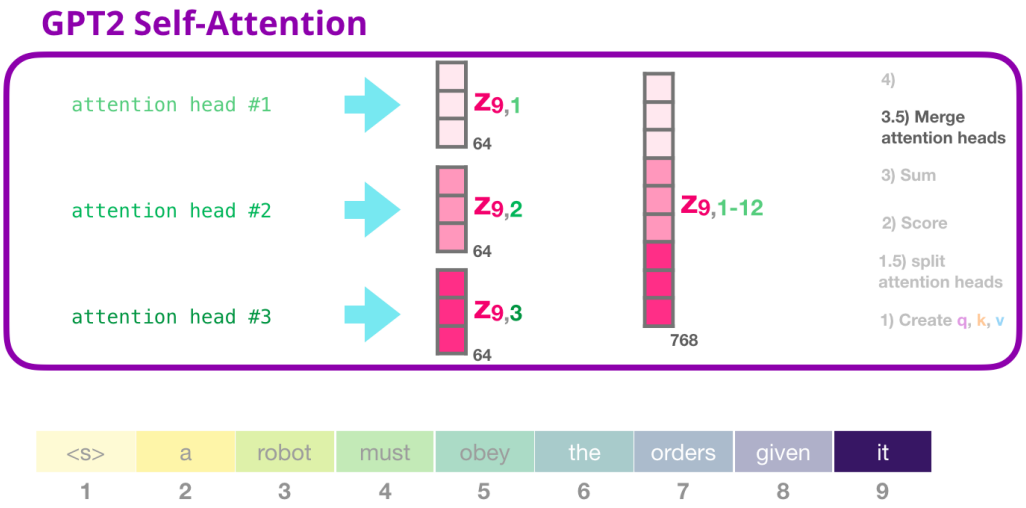
Now the token can get scored against all of keys of the other tokens (that were calculated in attention head #1 in previous iterations):



L.2.6.5 GPT-2 Self-attention: 3- Sum As we have seen before, we now multiply each value with its score, then sum them up, producing the result of self-attention for attention-head #1:

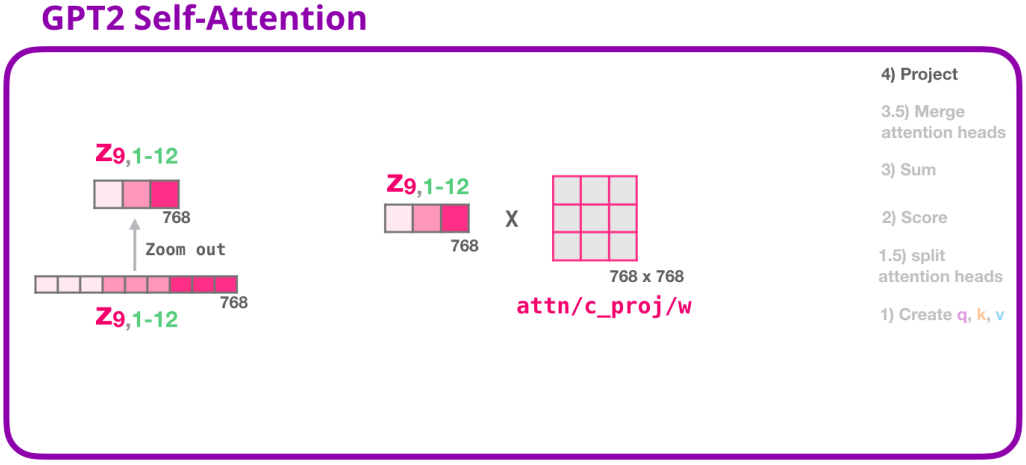


L.2.6.6 GPT-2 Self-attention: 3.5- Merge attention heads The way we deal with the various attention heads is that we first concatenate them into one vector:

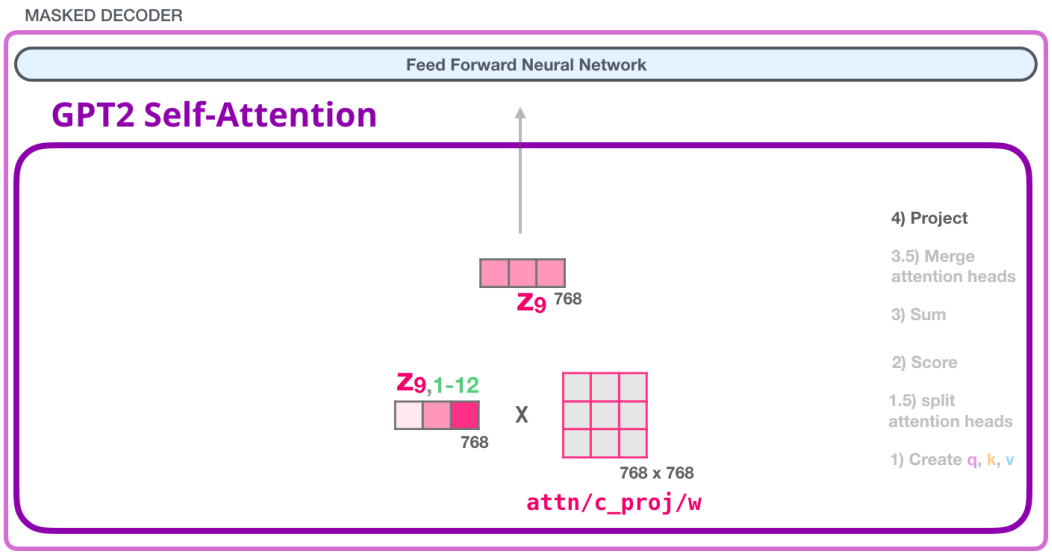


But the vector is not ready to be sent to the next sublayer just yet. We need to first turn this Frankenstein's-monster of hidden states into a homogenous representation.

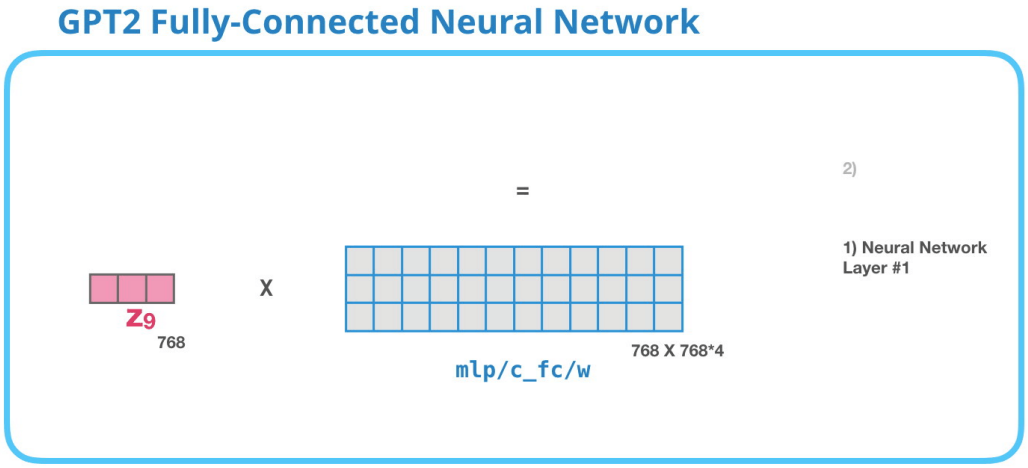
L.2.6.7 GPT-2 Self-attention: 4- Projecting We will let the model learn how to best map concatenated self-attention results into a vector that the feed-forward neural network can deal with. Here comes our second large weight matrix that projects the results of the attention heads into the output vector of the self-attention sublayer:



And with this, we have produced the vector we can send along to the next layer:

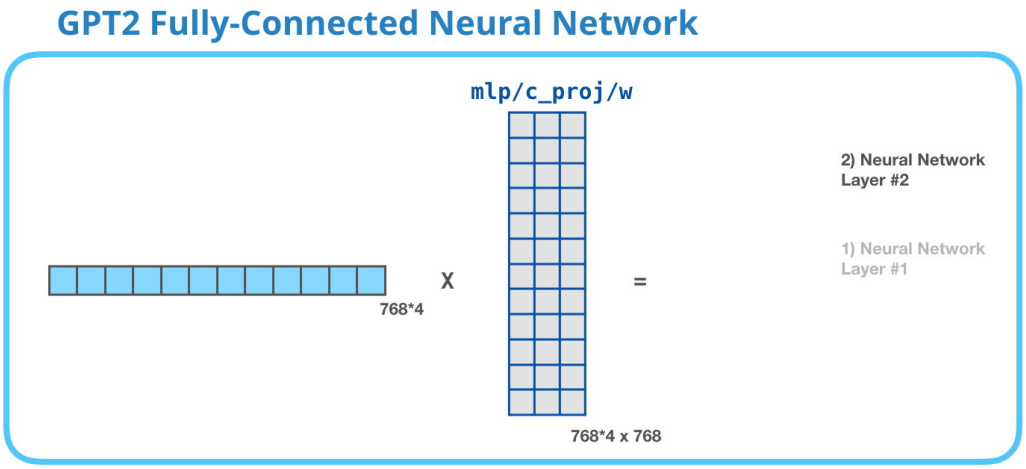


L.2.6.8 GPT-2 Fully-Connected Neural Network: Layer #1 The fully-connected neural network is where the block processes its input token after self-attention has included the appropriate context in its representation. It is made up of two layers. The first layer is four times the size of the model (Since GPT2 small is 768, this network would have 768*4 = 3072 units). Why four times? That is just the size the original transformer rolled with (model dimension was 512 and layer #1 in that model was 2048). This seems to give transformer models enough representational capacity to handle the tasks that have been thrown at them so far.



(Not shown: A bias vector)

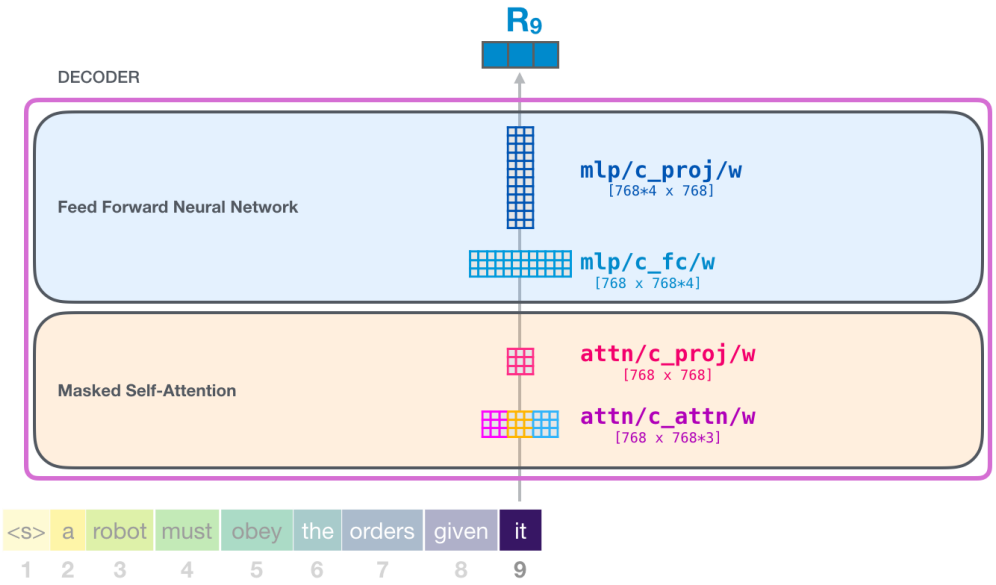
L.2.6.9 GPT-2 Fully-Connected Neural Network: Layer #2 - Projecting to model dimension The second layer projects the result from the first layer back into model dimension (768 for the small GPT2). The result of this multiplication is the result of the transformer block for this token.



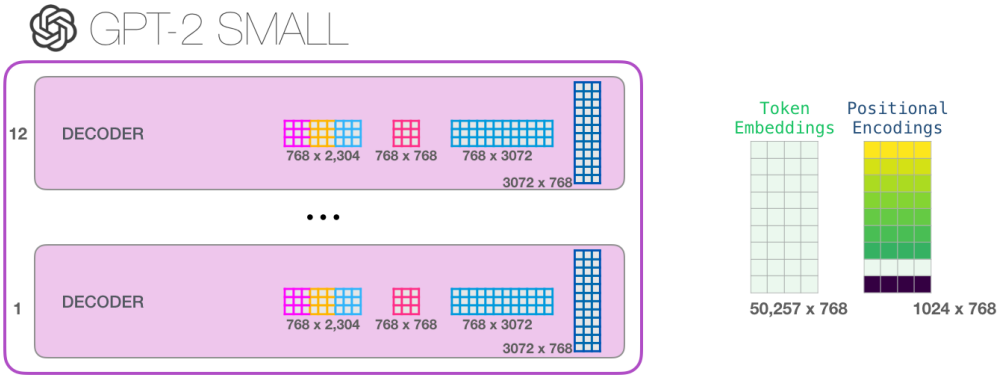
(Not shown: A bias vector)

L.2.7 Final recap

That is the most detailed version of the transformer block we will get into! You now pretty much have the vast majority of the picture of what happens inside of a transformer language model. To recap, our brave input vector encounters these weight matrices:



And each block has its own set of these weights. On the other hand, the model has only one token embedding matrix and one positional encoding matrix:



If you want to see all the parameters of the model, then I have tallied them here:

				Dimensions		Parameters	
Single Transformer Block	Conv1d	**attn/c_attn**	w	768	2,304	1,769,472	
			b		2,304	2,304	
		attn/c_proj	w	768	768	589,824	
			b		768	768	
		mlp/c_fc	w	768	3,072	2,359,296	
			b		768	768	
		mlp/c_proj	w	3,072	768	2,359,296	
			b		768	768	
	Norm	**ln_1**	g		768	768	
			b		768	768	
		ln_2	g		768	768	
			b		768	768	
					Total	7,085,568	per block
					X 12 blocks	85,026,816	In all blocks
		Embeddings		50,257	768	38,597,376	
		Positional Encoding		1,024	768	786,432	
					Grand Total	**124,410,624**	

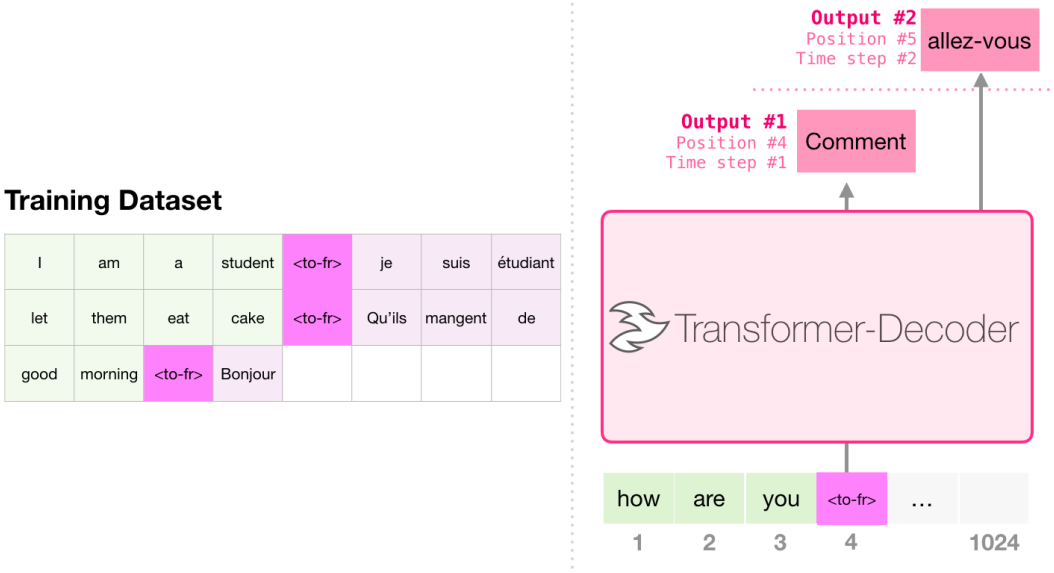
They add up to 124M parameters instead of 117M for some reason.

L.3 Part 3: Beyond Language Modeling

The decoder-only transformer keeps showing promise beyond language modeling. There are plenty of applications where it has shown success which can be described by similar visuals as the above. Let us close this post by looking at some of these applications

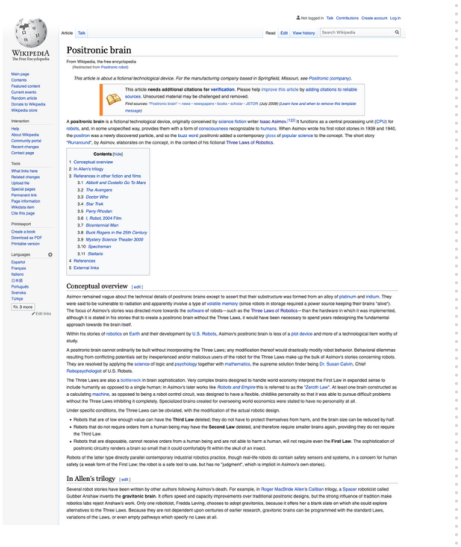
L.3.1 Machine Translation

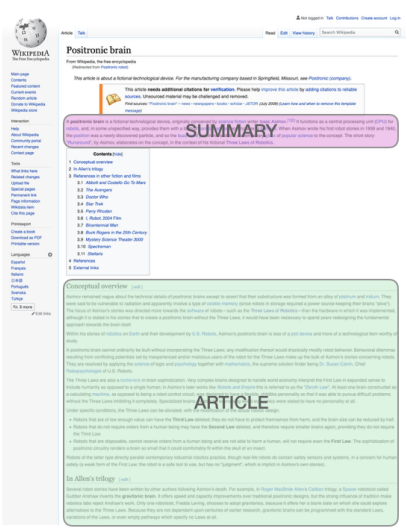
An encoder is not required to conduct translation. The same task can be addressed by a decoder-only transformer:



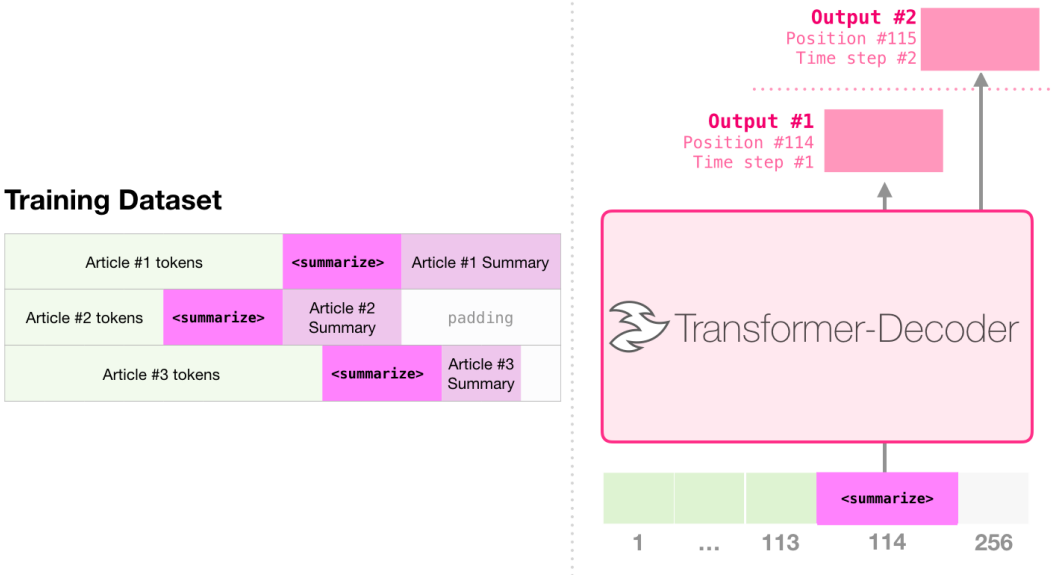
L.3.2 Summarization

This is the task that the first decoder-only transformer was trained on. Namely, it was trained to read a wikipedia article (without the opening section before the table of contents), and to summarize it. The actual opening sections of the articles were used as the labels in the training datasest:





The paper trained the model against wikipedia articles, and thus the trained model was able to summarize articles:



L.3.3 Transfer Learning

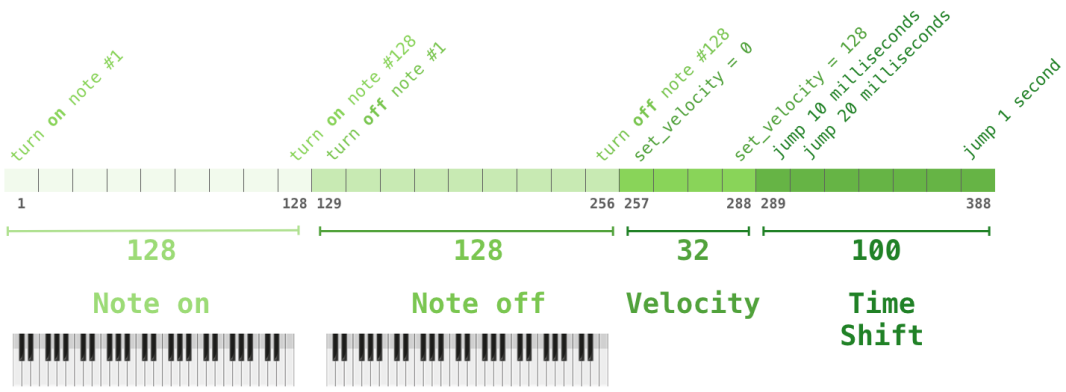
In Sample Efficient Text Summarization Using a Single Pre-Trained Transformer, a decoder-only transformer is first pre-trained on language modeling, then finetuned to do summarization. It turns out to achieve better results than a pre-trained encoder-decoder transformer in limited data settings.

The GPT2 paper also shows results of summarization after pre-training the model on language modeling.

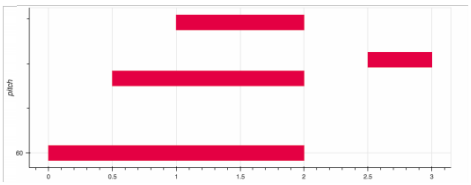
L.3.4 Music Generation

The Music Transformer uses a decoder-only transformer to generate music with expressive timing and dynamics. "Music Modeling" is just like language modeling – just let the model learn music in an unsupervised way, then have it sample outputs (what we called "rambling", earlier).

You might be curious as to how music is represented in this scenario. Remember that language modeling can be done through vector representations of either characters, words, or tokens that are parts of words. With a musical performance (let us think about the piano for now), we have to represent the notes, but also velocity – a measure of how hard the piano key is pressed.

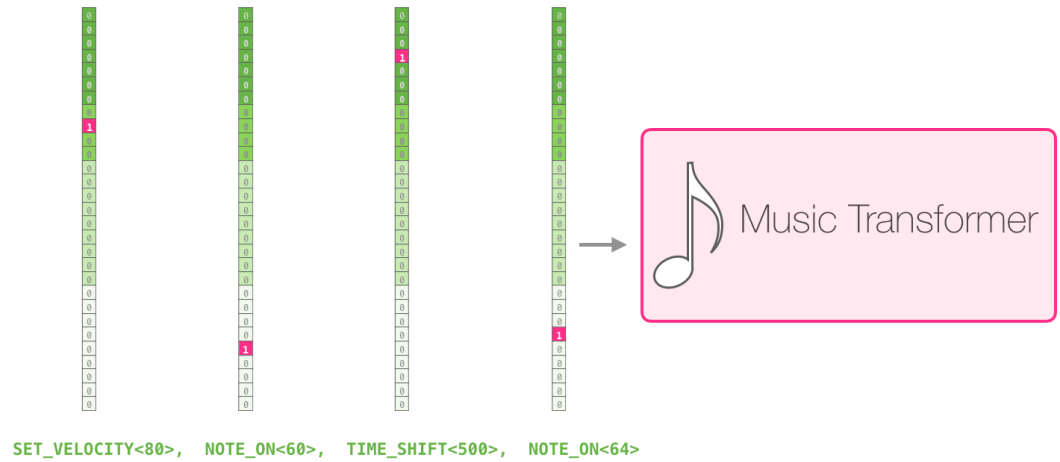


A performance is just a series of these one-hot vectors. A midi file can be converted into such a format. The paper has the following example input sequence:

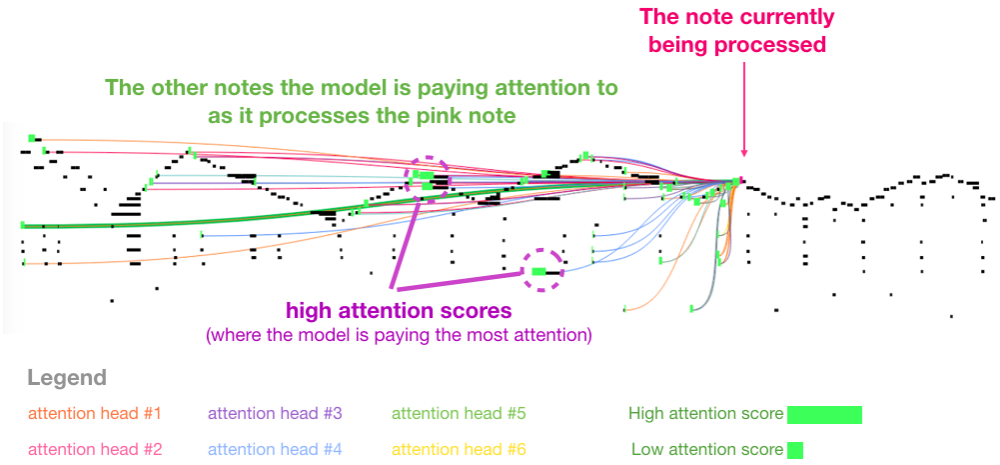


```
SET_VELOCITY<80>, NOTE_ON<60>
TIME_SHIFT<500>, NOTE_ON<64>
TIME_SHIFT<500>, NOTE_ON<67>
TIME_SHIFT<1000>, NOTE_OFF<60>, NOTE_OFF<64>,
NOTE_OFF<67>
TIME_SHIFT<500>, SET_VELOCITY<100>, NOTE_ON<65>
TIME_SHIFT<500>, NOTE_OFF<65>
```

The one-hot vector representation for this input sequence would look like this:



I love a visual in the paper that showcases self-attention in the Music Transformer. I have added some annotations to it here:



This piece has a recurring triangular contour. The query is at one of the latter peaks and it attends to all of the previous high notes on the peak, all the way to beginning of the piece. The figure shows a query (the source of all the attention lines) and previous memories being attended to (the notes that are receiving more softmax probabiliy is highlighted in). The coloring of the attention lines correspond to different heads and the width to the weight of the softmax probability.

If you are unclear on this representation of musical notes, check out this video.

L.4 Conclusion

This concludes our journey into the GPT2, and our exploration of its parent model, the decoder-only transformer. I hope that you come out of this appendix with a better understanding of self-attention and more comfort that you understand more of what goes on inside a transformer.