POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Analysis and assessment of neural-network-based text comprehension quiz generation algorithms

Supervisors

Prof. Maurizio MORISIO

Prof. Simone LEONARDI

Candidate

Marco CASCHETTO

July 2022

Summary

Artificial neural networks allowed to achieve extraordinary results in diverse fields, but one where they made a real breakthrough in is natural language processing, allowing to build models that could help automatising processes related to natural language.
As part of a company project in Fluentify - online teaching platform -, this study has been conducted with two main goals: from one side, discovering and analysing the latest state-of-the-art technologies to deal with the problem of understanding the main topics of a given text, generating meaningful multiple-choice questions related to those topics and also identifying wrong answers automatically; on the other hand, applying and testing such models on a series of text paragraphs and evaluating the results in order to detect how well they perform and how they can be improved. Even though all these steps already have several possible state-of-the-art solutions, composing them together was the real challenge, as not every workflow proved to be effective in reaching the ultimate goal of obtaining meaningful exercises.
The first chapter of this thesis gives an overview about the context, explaining the issues and the needs related to natural language processing tasks and its main applications in the real world.
The second chapter goes through the neural networks development, with particular attention to sequence-to-sequence problems (translation, question answering, or, more generally, every question related to converting an input sequence to an output

one), with a focus on transformers models, which represent the current state-of-the-art architecture that deals with such tasks.
The third chapter describes the algorithms used for the experimental tests, and gives details on the specific models that underlie them.
The fourth chapter illustrates how these tests have been conducted, explaining the reasons behind the proposed evaluation criteria and the obtained results. More specifically, several texts, with completely different topics but clustered by their length - from short (less than 500 words) to long (more than 1000 words) - have been fed into three algorithms, which have then been assessed by means of custom human-based criteria that looked at both questions and answers to detect how they performed.
The fifth and last chapter shows the conclusions and the future developments of the work, with particular respect to the possible further optimisations that the suggested workflow could apply in order to maximise performance according to the specific company goals.

Table of Contents

List of Tables

List of Figures

Acronyms

AI

Artificial Intelligence

NLP

Natural Language Processing

FNN

Feedforward Neural Network

RNN

Recurrent Neural Network

Seq2Seq

Sequence-to-Sequence

LSTM

Long short-term memory

GRU

Gated Recurrent Units

BERT	Bidirectional Encoder Representations from Transformers
MLM	Masked Language Model
T5	Text-to-Text Transfer Transformer
SQUAD	Stanford Question Answering Dataset
MS MARCO	Microsoft MAchine Reading Comprehension
RACE	ReAding Comprehension dataset from Examinations
CoQA	Conversational Question Answering Challenge
BLEU	Bilingual Evaluation Understudy Score

Chapter 1

Introduction

1.1 General context

In the last years e-learning has increasingly developed, and it changed the way education is approached, being now an essential tool for either students, teachers or companies. As a matter of fact, cutting edge technologies have enriched the educational context with new possibilities and improvements, some of which can be listed as follows:

- interaction between the parties does no longer require physical presence, allowing people from all over the world to meet in virtual classrooms
- learners can access educational material anytime, needing just a connection to internet and very small devices to carry on huge amount of data
- learning paths can be structured according to the learner's needs to give them a fully personalised experience
- automatic tools like language translators or spelling and grammar checkers are life-saver when it comes to learning a new language
- online platforms make data collecting easier, thus resulting in a better understanding of potential deficiencies, in order to make improvements and increase

the course efficiency;

- scalable infrastructures allow contents to be simultaneously accessed by thousands of students, in order to face educators shortage issues

In this context, an important role is played by Artificial Intelligence (AI). As of today, lots of fields take advantage of AI, and it seems hard not to find areas where this is not used; however, despite its large affect on everyday life, it is undeniable that this process is at the very beginning and that there is, nevertheless, room for improvement.
Among the diverse fields Artificial Intelligence is related to, Natural Language Processing is certainly one of the most complex to work on, but, at the same time, extremely powerful and useful when it comes to e-learning.

1.2 NLP

Natural Language Processing (NLP) is used to refer to a field of research that involves computer science, AI and linguistic to let computers properly understand natural language, i.e. language used by humans, both written and spoken. This goal is achieved through algorithms that must be able to analyse and comprehend human language on every matter: words and their meaning with respect to the context they are used, phrases and their grammar structure, sentiment of the sentences, etc.

1.2.1 History

A recent work by P. Johri et al. [1] gives a good summary about the way NLP started and developed through the last century.
The first attempts on this matter can be found in mid-1930s, when some patents for translating machines made their appearance. The goal was to use bilingual

dictionaries to map words between languages, but, despite the effort on using innovating approaches that could not struggle with the grammar of the language, those systems did not have concrete use and remained just abstract concepts. It was during World War II that NLP was successfully used in a practical context: on one side, Germans built a machine called "Enigma" to encrypt messages and create a secret code to facilitate their communications, whereas, on the other one, British came up with a computer able to decrypt Enigma codes. Soon after, in 1950, in his article "Computing Machinery and Intelligence" [2], Alan Turing proposed a new criterion to assess level of intelligence of machines, called "Turing test": the test aimed to determine whether a computer could actually think as a human or not. Further studies and attempts were made during the following decades, with several and different approaches; however, they were all based on handwritten rules that struggled to deal with real-world situations due to the huge complexity and variety of human language structures; indeed, fixed rules were not able to understand the proper words meaning or grammar of the sentences, thus giving insufficient results. A real revolution only happened in late 1980s, with introduction of machine learning and focus on statistical and probabilistic models that allowed to overcome such ambiguity: algorithms did not have to match hand-written patterns anymore; instead, they learned from the context and made proper choices on a decision tree-like structure by taking advantage of the probabilistic model. Due to the successful results, research relied on that for years, until deep learning and neural networks came up at the end of the first decade of 2000, resulting in techniques that are still used today and provide the current state-of-the-art results. Deep learning based approach has really proved to be the right way to work with NLP, giving the possibility to reach a better understanding of the text through a deep awareness of the neighbouring objects.
It is not accidental that deep learning-based NLP unleashed its real power only in recent years. Indeed, there are mainly two reasons for that.
First of all, these algorithms proved to be extremely burdensome when it came to

hardware requirements. Processors did not use to be as high performance and fast as today, that is, computational cost and time to perform such tasks were not feasible. This was a real limit for NLP - and deep learning, more generally - development, and technological progress for sure helped on reaching big results. Not only was the hardware availability inadequate, but also, in order for these deep learning algorithms to work well, tons of data were needed. Before digital transformation and introduction of online services, the amount of information produced was scarce and insufficient to be effective in terms of Natural Language Processing development. However, today is different: first of all, with advent of social media and personal devices, average user has become an active player, constantly producing data by explicitly uploading new contents, generating transactions for purchase or sell activities, etc. At the same time, conversion into digital of lots of material that used to be stored only physically as papers - especially banks, private companies or public administration related - and computerisation of business processes have exponentially increased the available data. Last, but not least, Internet of Things. Thanks to that, billions of smart devices are connected to internet, allowing to massively generate and transmit data every second.

1.2.2 Applications

As previously stated, Natural Language Processing is widely used today, as several are the tools available to help accomplishing different tasks. They can find utility among both private users, who take advantage of these during everyday life, and business companies, that keep implementing such tools to gain valuable insights and enrich the product with features to excel among the competitors.
Some of the main techniques based on NLP and their applications are:

- Auto-correction & Auto-complete
- Chatbots & Virtual Assistants
- Machine Translation

- Sentiment Analysis
- Speech Recognition
- Targeted Advertising
- Text Classification
- Text Extraction & Named Entity Recognition
- Text Summarisation

Auto-correction & Auto-complete

Features like autocorrection and automplete are among the most used, and almost everyone takes advantage of them every day.
Autocomplete, also known as auto prediction, allows people to save time by guessing what they are going to write and suggesting words or sentences. This is widely used on search engines, but it founds application also on emails clients to speed up emails composition.
Autocorrection, on the other hand, is able to detect any grammar or spelling mistakes during writing and to give hints to the user in order to fix or rephrase what is pointed out as incorrect.

Chatbots & Virtual Assistants

Chatbots and virtual assistants give people the feeling to interact with real human beings thanks to their deep use of machine learning. Standard assistants use sets of pre-defined rules to answer questions; AI based, instead, improve their efficiency as the interact with the users: the more they are used, the more they learn and understand to detect the question and the right way to answer. Well-known Google Assistant, Amazon Alexa and Apple Siri are perfect examples of what virtual assistants can achieve, using Speech Recognition (1.2.2), Natural Language Understanding and Natural Language Processing to successfully perform and reach top level results. Chatbots are mainly used by companies to automatise most of the customer support requests, allowing them to reduce the cost for customer support representatives, to speed up the response times - chatbots can be available anytime -, and to make company processes easier.

Machine Translation

Machine Translation goal is to translate amounts of text from a source language to a different target one while keeping the meaning intact. As mentioned in 1.2.1, machine translation represents one of the first attempts to apply Natural Language Processing. Since then, translations systems have changed a lot in relation to the technology used, until current state-of-the-art results have been accomplished thanks to Neural Machine Translation.
Companies use this NLP application to reach users from all over the world and help their business processes.

Sentiment Analysis

Sentiment Analysis helps analysing and interpreting human language with relation to its sentiments and intentions. As a matter of fact, it can be tough for machines to fully understand natural language for several reasons: first of all, human expressions sometimes have meanings beyond their literal one, thus resulting in expressions that computers might find hard to recognise. Moreover, people tend to use irony or sarcasm, or they even give the sentence a certain tone that machines do not catch. Sentiment analysis, through Natural language Understanding, an NLP sub-field, allows to overcome these language barriers and to obtain optimal results in diverse areas; companies, above all, make the most of it by applying it to their customers' emotions and fully get their thinking. For instance, this can be used to measure how much a user is satisfied about a specific product to detect their feeling about the company and get insights about what can be improved. Another common usage of Sentiment Analysis is the monitoring of users interactions, especially to prevent disputes on social media by detecting and hiding the negative comments.

Speech Recognition

Speech recognition is a technology that allows to interpret voice input data and re-model it in a new format that can be read and understood by machines. .
Applications for such process are diverse. As already mentioned in 1.2.2, speech recognition is used by voice assistants to detect instructions in spoken language, but several other applications are used, especially in business, like speech-to-text that converts oral communications into text for writing emails, translating, language analysis, transcribing calls, etc.

Text Classification

Manually processing an unstructured text and categorising it can be a hard and time-consuming operation, since natural language is extremely rich in terms of content diversity. For this reason, text classification systems have been developed to automatise these processes and make it easier to complete some tasks. Thanks to text classification, language is ordered and classified into predefined categories through specific tags. Email clients use this to properly divide emails in sections and filter them.

Text Extraction & Named Entity Recognition

Especially used in hiring and recruitment departments, as it prevents human resources from manually going through hundreds of resumes, text extraction ease the job of extrapolating meaningful information from a text, avoiding people to scan all the data before finding the relevant ones. In order for it to work, entities must be mapped properly, together with the relationships between them, and this can be accomplished through automated processes like Keywords Extraction and Named Entity Recognition.
Keywords Extraction aims to seek the relevant (key words) inside a text, which

can be useful to detect the context. Named Entity Recognition, on the other hand, helps identifying the named entities and classifying them in predefined categories such as numbers, places, objects, names, etc.

Text Summarisation

This automation is used to condense a generic text into a shorter version: it reduces and simplifies the paragraph while keeping the key informational elements and the main topic. A double approach can be followed to accomplish this task: extraction based, which cuts down large amount of content by keeping key sentences without any further processing, and abstraction based, that, in order to summarise the text, rephrases expressions and alters the original text.
This is usually used when working with large contents, as in legal documentation or in scientific papers, or with other NLP tasks like question answering and text classification.

Targeted Advertising

Targeted Advertising is the process of showing advertisements to users based on their previous research, though Natural Language Processing techniques.
This NLP application works by looking at different properties. First of all, it uses keyword matching to compare keywords or phrases associated with ads with the keywords related to the text searched by the user. If there is a match, then there is a very high probability that the customer is interested in products shown in the advertisement. Secondly, the process takes in consideration also the websites and pages the users has interacted with.
This approach is extensively used by companies today, as it helps them saving money and showing products only to customers who are actually interested in.

Chapter 2

Architecture

2.1 Neural networks

As stated in 1.2.1, development of Natural Language Processing had a breakthrough when models used to represent the language began to make use of neural networks. Firstly proposed by McCulloch and Pitts in 1943 [3], neural networks aim to mimic the behaviour of human brain as a set of neurons interconnected between each other in a network that allows transit of information. In order for such representation to be possible, artificial neural networks are made of nodes, each designated to receive data as input, make some computation and forward the new output.

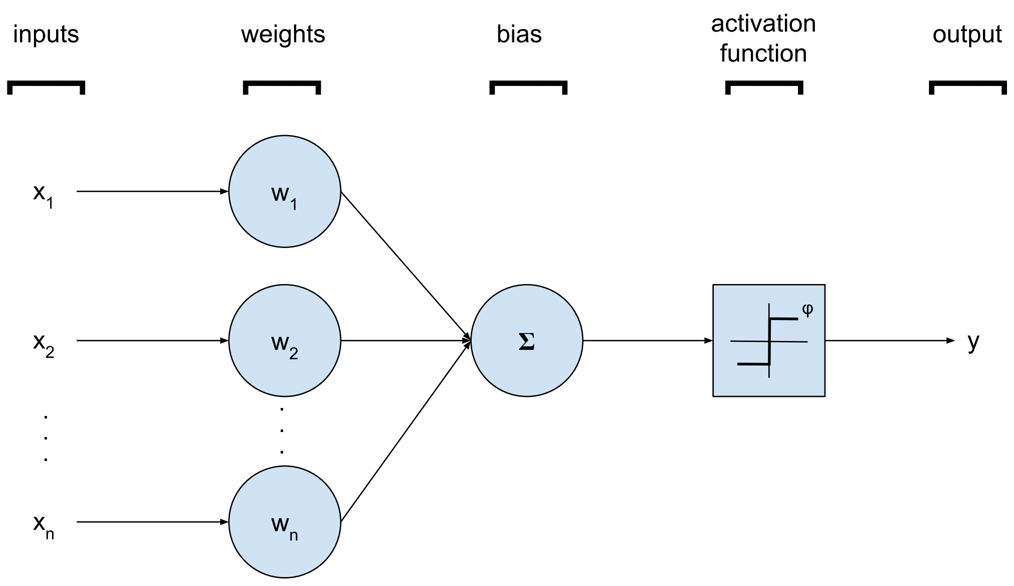

Figure 2.1: Artificial neuron

As visible in 2.1, a node is made of the following components:

- inputs: input data fed into the current node
- weights: coefficients combined together with the inputs to assign significance by either amplifying or dampening them
- bias: value that might be added to shift the result of the previous computation
- activation function: function to add non-linearity and help determining whether the signal should progress or not

Artificial neurons are organised in layers. A typical representation is the one shown in 2.2, where information transit from one layer to another. There always is an input layer, where input data are entered, some internal layers where actual computation is performed and a last one containing the final results (which is usually a single node, but it is not mandatory); therefore, each layer's input is given by the previous layer's output.

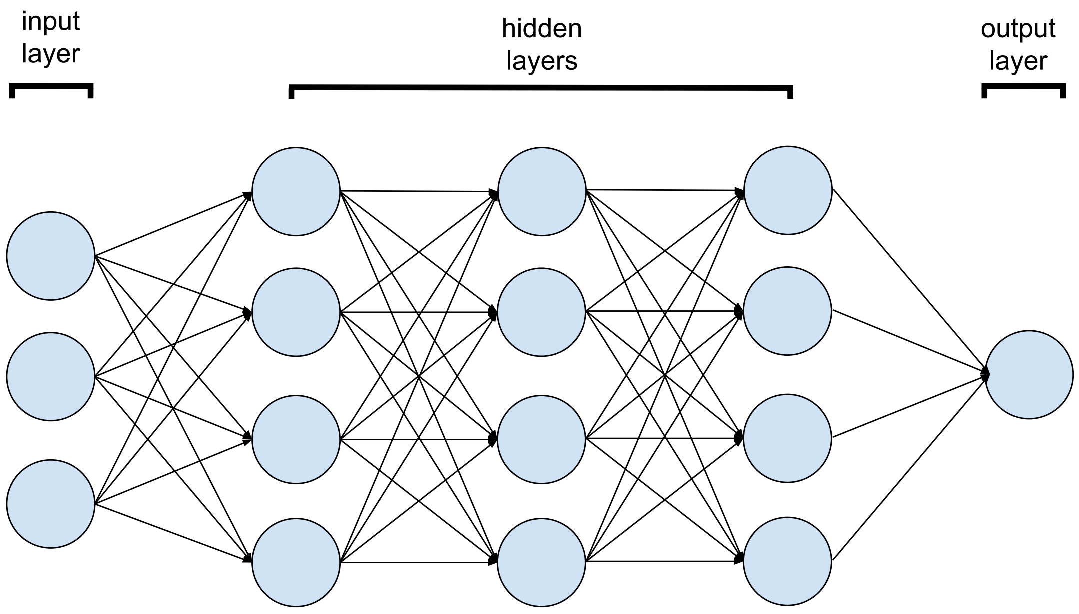

Figure 2.2: Neural network layers

An artificial network were information is channeled from one way to another is called Feedforward Neural Network (FNN). FNNs are pretty straightforward, having nodes that only analyse the current input and have no clue about what has already been computed: this means that information goes straight from the input layers to the hidden ones and from these to the output layer, and it never crosses a node more than once. This is not true for Recurrent Neural Networks. In a RNN, nodes work in a more effective way: not only do they consider the current input, but they also make use of the past evaluations to reach context awareness and make smarter predictions. This concept is represented in 2.3, where output for step k is not influenced by just weights applied at the current inputs, but also by the hidden state coming from the previous step. As a consequence, there will not be a fixed output for a given input, but, instead, this might change according to the hidden state.

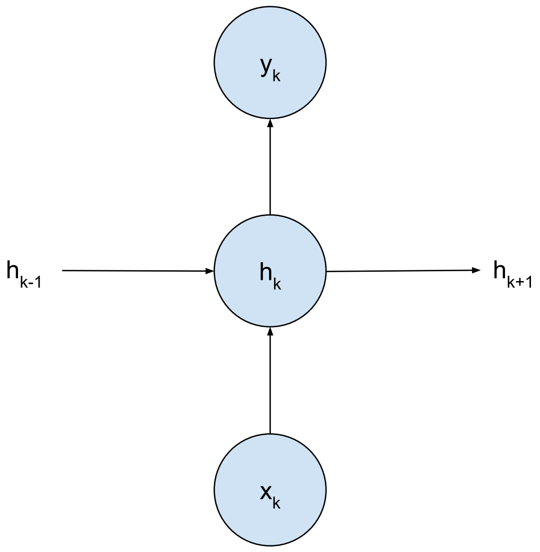

Figure 2.3: Recurrent neural network step

Recurrent Neural Network can vary depending on how many input or output layers are present:

- One-to-One, the simplest one, behaves as a normal neural network having a single input and a single output
- One-to-Many, with a single input and multiple outputs (e.g. image captioning, having an image as input and several data that describe that image as output)
- Many-to-One, when input has not a fixes size, but output does. Sentiment analysis is a perfect example of such structure, as it takes multiple input data (sequence of words) and always give a single output as a number that describes the sentiment of the sentence (usually from -1 to 1)
- Many-to-many, which can have input and output of the same length (like in Named Entity Recognition) or not (like in Machine Translation); these are usually referred as Sequence-to-Sequence (Seq2Seq), meaning that they convert sequences of inputs into sequences of outputs, regardless their size

2.2 Long short-term memory

Although these models are able to maintain information in memory over time, thanks to the feedback loops in the recurrent layer, they may still struggle when it comes to work with data that need long context dependencies, suffering from a problem known as "vanishing gradient" [4]. As a matter of fact, as the number of layers that use an activation function increases, gradient of the loss function - with respect to the weights - tends to approach values close to zero and to become working less effectively. In order to address this issue and be able to solve problems that require long-term temporal dependencies, Hochreiter and Schmidhuber [5] improved vanilla RNN units and made them more sophisticated by adding complexity to their structure. The basic concept of the newtorks that make use of such special components, called Long short-term memory, are pretty much the same as normal RNN, but their units are based on a memory and multiple gates that allow the flow of information to be controlled in a better and more accurate way. In contrast with standard RNN units, which have a very simple configuration with a single neural network layer, LSTM ones are based on four different layers, as it can be seen in the following image where:

- σ is a sigmoid function
- tanh is a tanh function
- + and x are operands that represent, respectively, sum and multiplication
- C are values in the cell state (long term memory)
- h are values in the hidden state (short term memory)
- x are input values
- W are the weights
- b are biases

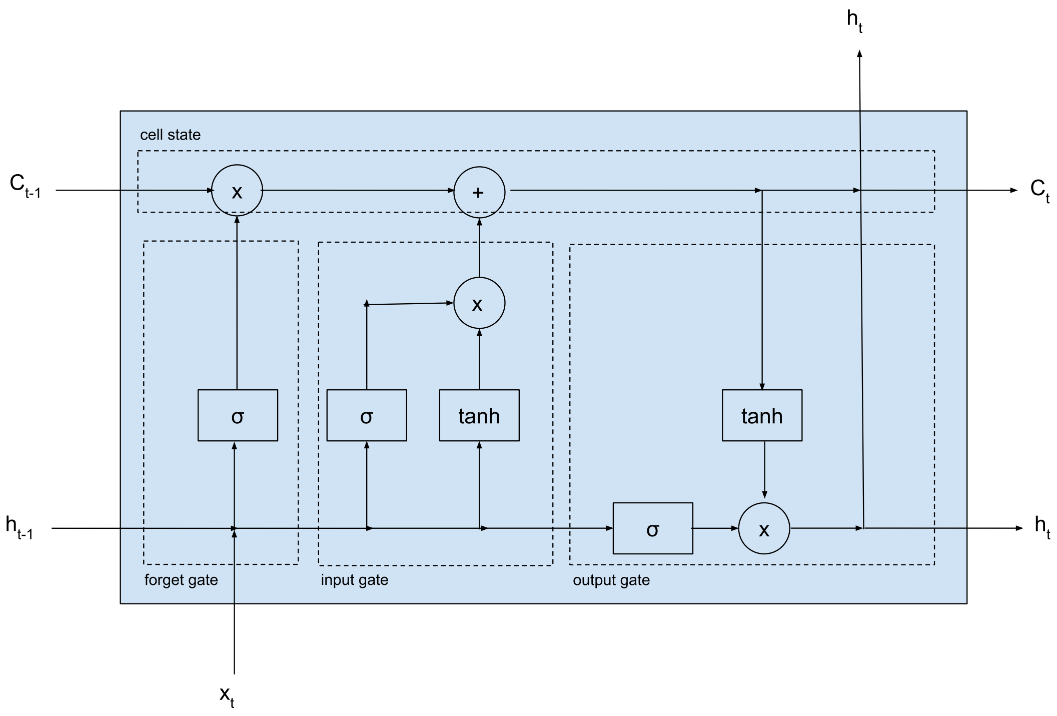

Figure 2.4: Long short-term memory unit

The module is made of several layers:

1. forget gate layer: the job of this first layer is to decide which information of the cell state must be kept or not, in order to remove the useless one. This procedure is done by looking at the current input x_t and the previous hidden state h_{t-1} and using a sigmoid function to return values between 0 and 1 that will be applied to the numbers from the cell state: a value close to 0 means that the information is going to be discarded, whilst one close to 1 means that it is going to remain in the cell state.
   The following formula summarises the process:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

2. input gate layer: this is the layer where cell state update is actually performed with the new values. It is made of two parts: the first one is a sigmoid similar to the forget layer's one, and decides which values will be updated; the second one is a tanh layer that defines candidate values to be added to the state. Outputs from the two will be combined together and will define the update on the state:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C_t} = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

3. cell state layer: with outputs coming from the previous steps, cell state can be computed. Its values are firstly multiplied by the result of the forget layer, and, after that, the remaining ones will be added to the new values computed in input gate as follows:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C_t}$$

4. output gate layer: this is the last layer, where the hidden state is computed. It takes the result of the previous computations and returns to the next step a filtered version after having manipulated it with a sigmoid function (applied to the previous hidden state and to the current input) and a tanh one (in order for the values to be included between -1 and 1). Formulas that describe this last process are the following:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * tanh(C_t)$$

2.3 Transformers

LSTM units (or variants like the Gated Recurrent Units) are extensively used in Sequence-to-Sequence prediction problems, like Machine Translation or Question

Generation ones. Sequence-to-Sequence models, which were introduced for the first time by Google in 2014 [6] can be built on different architectures, but one approach that proved to be very effective is the Encoder-Decoder.
Encoder-Decoder is a learning model made of two main RNN components, the encoder and the decoder, with an intermediate vector in between. As the name suggests, task of the encoder is to process each token of an input sequence and encode it in order to create a vector of values that will be the intermediate state. This vector, also known as context vector, is built in a way that helps encapsulating the whole meaning of all the input elements and gives the decoder as much information as possible to make accurate predictions; the decoder, as a matter of fact, after having been fed with the final states of the encoder, takes these values and processes them to generate the output sequence.

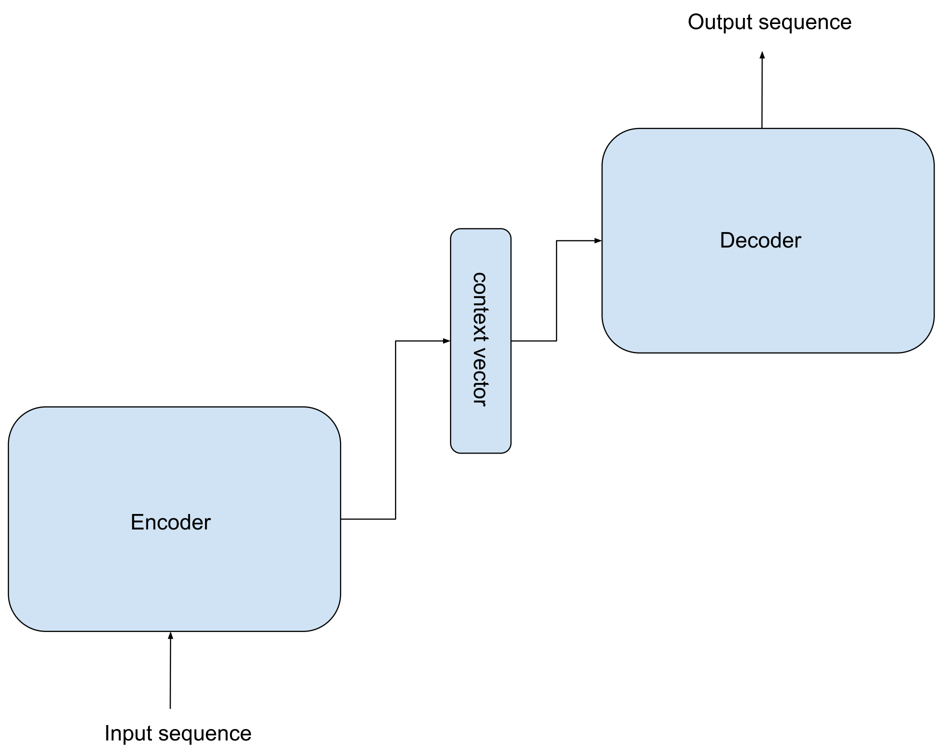

Figure 2.5: Encoder-decoder

The structure defined above really helps performing great with Sequence-to-Sequence problems where input and output lengths vary, but it still has some limitations: since the context vector contains all the information related to the input sequence, as the length of the latter increase, the more likely becomes for the model to forget or miss part of the information that was initially present. This results in encoder-decoder model struggling to perform well when sentences are too long, with intermediate vector representing a point of congestion. To overcome such issue, known as long-range dependency problem, Bahdanau et al. [7] proposed a new model based on an innovative mechanism, called attention, that allows to free the encoder-decoder architecture from the fixed-length internal representation. The goal is achieved in two ways: from one side, instead of just preserving the hidden state of the last step, all the intermediate outputs at each encoding stage are kept; on the other hand, the decoder performs an extra step assigning a relative importance to the various input parts in order to only consider the useful ones, otherwise there would be too much information to pick from.

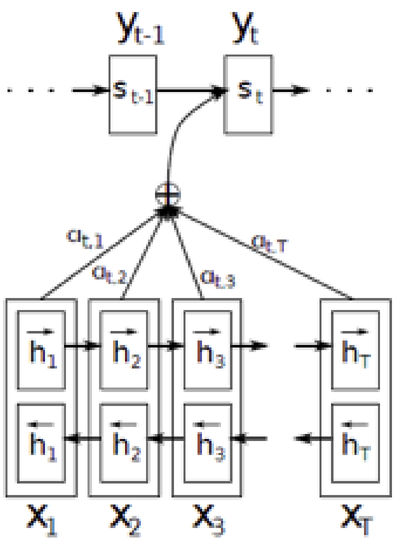
y_{t-1}
y_t
s_{t-1}
s_t
$a_{t,1}$
$a_{t,2}$
$a_{t,3}$
$a_{t,T}$
$\overrightarrow{h}_1$ $\overrightarrow{h}_2$ $\overrightarrow{h}_3$ $\overrightarrow{h}_T$
$\overleftarrow{h}_1$ $\overleftarrow{h}_2$ $\overleftarrow{h}_3$ $\overleftarrow{h}_T$
x_1 x_2 x_3 x_T

Figure 2.6: Attention model [7]

Figure 2.6 illustrates the model described before, where sequential RNNs/LSTMs of the encoder are replaced by bidirectional LSTMs to both include preceding and following data in current annotation and further steps are performed in order to let the context vector reaching full awareness about connections between output and input. The context vector is built taking in consideration the following elements:

- alignment scores: a score is computed to define how well the input at position i matches the current output at position t and tells how much of each source hidden state should be considered for each output; the alignment model is represented by a function a implemented through a FNN and takes in consideration both the hidden state h_i and the previous output s_{t-1}:

$$e_{t,i} = a(s_{t-1}, h_i)$$

- weights: a softmax operation is applied to the alignment scores to normalise them and computing the weights used for the context vector:

$$\alpha_{t,i} = softmax(e_{t,i})$$

Therefore, the context vector is computed through a weighted sum of all the encoder hidden states as shown in the following formula:

$$c_t = \sum_{i=1}^{T} a_{t,i} h_i$$

To sum up, the proposed architecture, which takes advantage of attention mechanism, allows to deal with the encoder-decoder limitation of long sequences by looking at all the states of the encoder, each one accordingly weighted, to access information about all the elements of the input sequence, instead of just paying attention to the last one.
Attention concept has been used a lot in NLP tasks, especially translation or question answering, and several studies have been conducted to improve its performance. The biggest flaw of such model is that it is very high demanding when it comes to long input sequences, as it handles them word-by-word in a sequential fashion, thus requiring *t-1* steps before performing the *t-th* one. This is obviously an obstacle towards parallelisation, being not only time consuming, but also highly computationally inefficient. A solution for this problem has been offered by Vaswani et al. in a paper called "Attention is all you need" [8], where they propose a new infrastructure for the attention model, called "transformer". This novel architecture implements different solutions to overcome the discussed issue, achieving current state-of-the-art results in several NLP applications, and its model is depicted below:

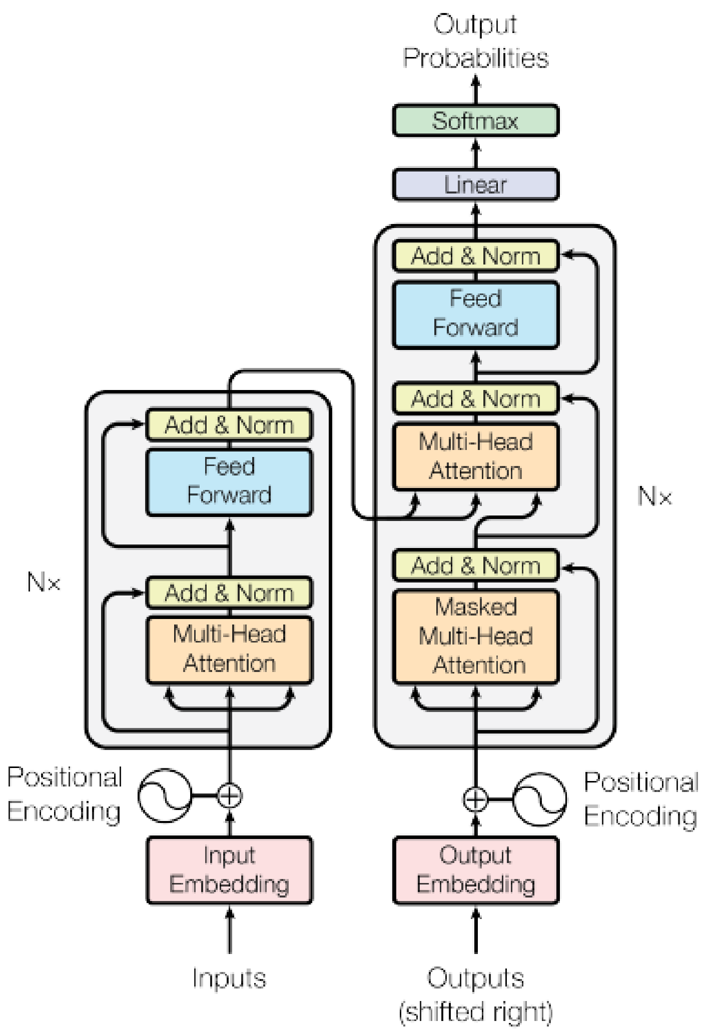

Figure 2.7: Transformer architecture [8]

Transformers are still based on encoder-decoder architecture, having an encoding and a decoding components, connected in some way between each other. The differences are related to how the components are structured inside, as well as the transformations applied to the data before access them.
First thing to notice is that input data is conveniently transformed and mapped before accessing encoder and decoder; as a matter of fact, as visible in the picture above, two different modules precede these blocks: the embedding and the positional

encoding. The embedding block is widely used in NLP applications and it is used to map each token into embeddings of the same size, as this helps representation and understanding the similar ones. Positional encoding, instead, is used to account for the words order in the input sequence; without that, the model would not be able to perform differently for permutation of the same words, thus the need to add to the embedding vectors a representation of the position of the words. An example of function to compute positional encoding is the following, which applies sine and cosine for, respectively, even and odd positions:

$$PE_{(pos,2i)} = \sin\left(pos/10000^{2i/dmodel}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(pos/10000^{2i/dmodel}\right)$$

where *pos* is position of the word inside the sequence, d the size of the word embedding and i refers to each of the d individual dimensions of the embedding. This specific choice allows to both prevent the numbers exploding for long sentences if positions were simply increasing integers, as well as keeping consistent meaning across different sentences if positions are in range [0,1].
After embedding and positional encoding blocks, data reaches encoder, which differs from the one described before for several reasons.
First of all, the transformer encoder is internally stacked and composed of multiple encoder layers (6 in the original paper); each of them is identical in structure and it is broken down into two sub-layers: a self-attention one and a Feedforward Neural Network. A self-attention model allows inputs to interact with each other in order to find out who they should pay more attention to and discover clues that can help lead to a better encoding of the current one. The process to compute self-attention layer output can be divided in some steps:

- creation of query (Q), key (K) and value (V) vectors by multiplication of embedding with 3 pre-trained matrices
- calculation of a score: given a certain word the module is trying to compute the self-attention for, a score is calculated for each token of the input sequence

with respect to that specific word to express how much attention must be put on the former when analysing the latter. This is done through the dot product of the query vector with the key one

- division of the score by the the square root of the dimension of the key vectors to stabilise the gradients
- application of a softmax function to the value computed by the previous steps to normalise it
- multiplication of the softmaxed attention score for each input by its corresponding value (from value vector)
- summation of the weighted values

These operations can be condensed in the following formula:

$$Attention(Q, K, V) = softmax(QK^T/\sqrt{d_k})V$$

See figure 2.8 for a visual representation of the process.

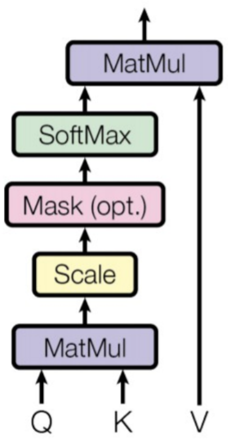

Figure 2.8: Scaled dot-product attention [8]

The result of such process would then be fed to a Feedforward Neural Network that applies two linear transformations, but in practice, before doing that, the attention process is improved with the innovative concept of "multi-headed attention", which consists on creating N different attention heads (N > 1, usually 8), each with different weight matrices, to allow the process to jointly use different representation subspaces of queries, keys, and values. Since the Feedforward Neural Network is expecting a single matrix, the obtained matrices only need to be concatenated and multiplied by an additional weight matrix.

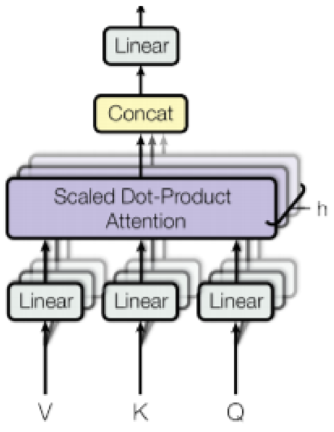

Figure 2.9: Multi-head attention [8]

Both the multi-head attention layer and the feedforward one have also a residual connection around them, and are followed by a layer normalisation step.
The second main component of the transformer architecture is the decoder, which shares several similarities with the encoder. As a matter of fact, it also consists of a stack of multiple layers with three sub-layers inside: the first one receives as input the output of the decoder stack at the previous step, accordingly embedded and added with positional encoding as the inputs of the encoder stack. Nevertheless, this first sub-layer operates in a slightly different way than the one in the encoder, as it is only allowed to attend to earlier positions in the output sequence. This is achieved with a mask over the future positions by setting their matrix values to $-\infty$ before the softmax step. The second sub-layer is called "encoder decoder attention layer" and differs from the encoder corresponding one, too, since it takes the keys and values from the output of the encoder stack, but it creates the query matrix from the previous decoder sub-layer. Along with these two sub-layers there

is a third one, which is a fully connected Feedforward Neural Network, and residual connections are employed around them, together with a normalisation layer just right after.
Finally, the output of the decoder stack is received by the last block, the "Linear Layer", which projects the vector of floats obtained from the previous computations into a larger one ("logits vector") to generate, together with the following softmax layer, a prediction for the next word.

Chapter 3

Libraries and models

In order to generate multiple-choice questions, a research on transformer-based models that could handle such task has been conducted, leveraging state-of-the-art Natural Language Processing techniques that are going to be properly explained. Three different sets of algorithms have been used to run the tests and they will be identified, respectively, as *lib_bert*_1, *lib_t*5_2 and *lib_t*5_3. All of them are based on transformer models, even though they differ from they way these models are structured or trained.

3.1 First library

The first analised and tested library was build by Ramsri Goutham Golla [9]. The entry point of this algorithm is a text paragraph and the following steps are performed in order to generate multiple-choice questions:

1. summarising the text
2. extracting keywords from the summarised text
3. mapping keywords with sentences from summarised text
4. generating distractors

An important note about this first approach is that questions are not real questions, but only sentences picked from the text with a blank space instead of the missing keyword.

3.1.1 Text summarising

To summarise the text, the process leverages "bert-extractive-summarizer", which is based on BERT.
BERT, acronym for Bidirectional Encoder Representations from Transformers, is an open-source machine learning framework for NLP, designed upon transformers by Jacob Devlin, Ming-Wei Chang, Kenton Lee and Kristina Toutanova [10]; however, it differs from traditional transformers for several reasons.
First of all, BERT offers bidirectional capability by only taking advantage of the encoder stack, which allows to attend to tokens on both left and right (traditional transformers decoder are uni-directional). Also, BERT is based on two main steps: pre-training and fine-tuning. During pre-training, the model is trained on unlabeled data over two different NLP tasks:

- Masked Language Model (MLM), that aims to hide a word in a sentence and then have the program predict what word has been hidden (masked) based on the hidden word's context to avoid misleading connection between words
- Next Sentence Prediction, to let the program predict whether two given sentences have a logical, sequential connection or whether their relationship is simply random.

BERT has been pre-trained on a large corpus comprising the Toronto Book Corpus and Wikipedia.
For fine-tuning, the BERT model is first initialised with the pre-trained parameters, and all of the parameters are fine-tuned using labeled data from the downstream tasks. Each downstream task has separate fine-tuned models, even though they are initialized with the same pre-trained parameters.

Here follows the procedure for the text summarising:

Listing 3.1: text summarising

```python
from summarizer import Summarizer
model = Summarizer()
result = model(original_text , min_length=60, max_length = 500 , ratio
    = 0.4)
summarized_text = ''.join(result)
```

3.1.2 Keywords extracting

Keywords extracting is performed through python keywords extractor. Only nouns are taken by the algorithm, with a cap of 20, and only keywords that are present in the summarised text are kept, as visible in the following code snippet:

Listing 3.2: keyword extractor

```python
import pprint
import itertools
import re
import pke
import string
from nltk.corpus import stopwords

def get_nouns_multipartite(text):
    out=[]

    extractor = pke.unsupervised.MultipartiteRank()
    extractor.load_document(input=text)
   #   not contain punctuation marks or stopwords as candidates.
    pos = {'PROPN'}
   #pos = {'VERB', 'ADJ', 'NOUN'}
    stoplist = list(string.punctuation)
    stoplist += ['-lrb-', '-rrb-', '-lcb-', '-rcb-', '-lsb-', '-rsb-'
   ]
    stoplist += stopwords.words('english')
    extractor.candidate_selection(pos=pos, stoplist=stoplist)
    # 4. build the Multipartite graph and rank candidates using
   random walk,
    #    alpha controls the weight adjustment mechanism, see
   TopicRank for
    #    threshold/method parameters.
    extractor.candidate_weighting(alpha=1.1,
                                  threshold=0.75,
                                  method='average')
    keyphrases = extractor.get_n_best(n=20)

    for key in keyphrases:
        out.append(key[0])

    return out

keywords = get_nouns_multipartite(full_text)
print (keywords)
filtered_keys=[]
for keyword in keywords:
    if keyword.lower() in summarized_text.lower():
        filtered_keys.append(keyword)

print (filtered_keys)
```

3.1.3 Sentence Mapping

Third step is a sentence mapping to map each selected keyword from the previous step to a sentence from the summarised text, keeping only the ones with a length lower than 20 letters. The following python code will describe the process:

Listing 3.3: sentence mapping

```
from nltk.tokenize import sent_tokenize
from flashtext import KeywordProcessor
```

```
def tokenize_sentences(text):
    sentences = [sent_tokenize(text)]
    sentences = [y for x in sentences for y in x]
    # Remove any short sentences less than 20 letters.
    sentences = [sentence.strip() for sentence in sentences if len(
    sentence) > 20]
    return sentences
def get_sentences_for_keyword(keywords, sentences):
    keyword_processor = KeywordProcessor()
    keyword_sentences = {}
    for word in keywords:
        keyword_sentences[word] = []
        keyword_processor.add_keyword(word)
    for sentence in sentences:
        keywords_found = keyword_processor.extract_keywords(sentence)
        for key in keywords_found:
            keyword_sentences[key].append(sentence)
for key in keyword_sentences.keys():
        values = keyword_sentences[key]
        values = sorted(values, key=len, reverse=True)
        keyword_sentences[key] = values
    return keyword_sentences
sentences = tokenize_sentences(summarized_text)
keyword_sentence_mapping = get_sentences_for_keyword(filtered_keys,
    sentences)

print (keyword_sentence_mapping)
```

3.1.4 Generating distractors

The last step consists of generating wrong answers (distractors) to choose from. In order to achieve this goal, two different approaches are used: Wordnet and Conceptnet.

Wordnet [11] is a database of English words used by the algorithm to get the sense of a word and, if successful, to find its hypernym to look for hyponyms related to that. In case Wordnet fails, the process relies on Conceptnet, a multilingual knowledge base, representing words and phrases that people use and the common-sense relationships between them. Even though Conceptnet does not have provision to disambiguate between different word sense, it will try to get distractors for the given word with the picked one. The high level algorithm can be seen in the following lines:

Listing 3.4: distractors generation

```
key_distractor_list = {}

for keyword in keyword_sentence_mapping:
    wordsense = get_wordsense(keyword_sentence_mapping[keyword][0],
   keyword)
    if wordsense:
        distractors = get_distractors_wordnet(wordsense,keyword)
        if len(distractors) ==0:
            distractors = get_distractors_conceptnet(keyword)
        if len(distractors) != 0:
            key_distractor_list[keyword] = distractors
    else:

        distractors = get_distractors_conceptnet(keyword)
        if len(distractors) != 0:
            key_distractor_list[keyword] = distractors
```

3.2 Second library

Library *lib_t5_2* has been developed by the same author of the previous one along with some other contributors [12], but works differently. Transformers play still a main role in it, but in a new variant called T5 (Text-to-Text Transfer Transformer) [13], which has proved to achieve state-of-the-art results on multiple NLP tasks

like question answering, summarisation or machin translation.
T5 is an encoder-decoder model whose main feature is conversion of all NLP problems into a text-to-text format where the input and output are always strings; this allows for the use of the same model, loss function and hyperparameters etc. across diverse set of tasks.
Another important tool this library takes also advantage of is Sense2vec [14], a neural network model that generates vector space representations of words from large corpora. It does this by creating embeddings for "senses", given that a sense is a word combined with a label i.e. the information that represents the context in which the word is used. This label can be a Entity Name, POS Tag, Polarity, etc. In this context, it is used to generate distractors.
Regarding the training, the model, which actually is an extended tool that deals not only with generation of multiple-choice questions, but also boolean ones, FAQs and paraphrasing, has been trained on different datasets:

- Quora Question pairs [15], a dataset of duplicate question pairs used to train and test models of semantic equivalence
- BoolQ [16], a question answering dataset for yes/no questions (irrelevant in this context)
- SQUAD [17], acronym for Stanford Question Answering Dataset, a reading comprehension dataset, consisting of questions posed by crowdworkers on Wikipedia articles, where the answer to every question is a segment of text, or span, from the corresponding reading passage, or the question might be unanswerable
- MS MARCO (Microsoft MAchine Reading Comprehension) [18], a collection of datasets focused on deep learning in search

The first step that this algorithm performs is tokenising the text into sentences through the *tokenize_sentences* function, which, together with that, takes care of filtering the sentences by keeping only the ones with a length greater than 20. This first step is shown in the code below:

Listing 3.5: tokenise sentences

```
def tokenize_sentences(text):
    sentences = [sent_tokenize(text)]
    sentences = [y for x in sentences for y in x]
    # Remove any short sentences less than 20 letters.
    sentences = [sentence.strip() for sentence in sentences if
len(sentence) > 20]
    return sentences
```

After that, as in the library *lib_bert*_1, the process continues by extracting the keywords from a reduced version of the original text, obtained by concatenating the sentences retrieved in the previous step. Here is the *get_keywords* function that achieves that:

Listing 3.6: tokenise sentences

```
def get_keywords(nlp,text,max_keywords,s2v,fdist,
normalized_levenshtein,no_of_sentences):
    doc = nlp(text)
    max_keywords = int(max_keywords)

    keywords = get_nouns_multipartite(text)
    keywords = sorted(keywords, key=lambda x: fdist[x])
    keywords = filter_phrases(keywords, max_keywords,
normalized_levenshtein )

    phrase_keys = get_phrases(doc)
    filtered_phrases = filter_phrases(phrase_keys, max_keywords,
normalized_levenshtein )

    total_phrases = keywords + filtered_phrases

```

```
        total_phrases_filtered = filter_phrases(total_phrases, min(max_keywords, 2*no_of_sentences),normalized_levenshtein )


        answers = []
        for answer in total_phrases_filtered:
            if answer not in answers and MCQs_available(answer,s2v):
                answers.append(answer)

        answers = answers[:max_keywords]
        return answers
```

Later, such keywords are simply mapped with the sentences:

Listing 3.7: tokenise sentences

```
    def get_sentences_for_keyword(keywords, sentences):
        keyword_processor = KeywordProcessor()
        keyword_sentences = {}
        for word in keywords:
            word = word.strip()
            keyword_sentences[word] = []
            keyword_processor.add_keyword(word)
        for sentence in sentences:
            keywords_found = keyword_processor.extract_keywords(sentence)
            for key in keywords_found:
                keyword_sentences[key].append(sentence)

        for key in keyword_sentences.keys():
            values = keyword_sentences[key]
            values = sorted(values, key=len, reverse=True)
            keyword_sentences[key] = values

        delete_keys = []
        for k in keyword_sentences.keys():
            if len(keyword_sentences[k]) == 0:
                delete_keys.append(k)
        for del_key in delete_keys:
            del keyword_sentences[del_key]

        return keyword_sentences
```

Eventually, the questions are generated from the computed keyword-sentence mapping

Listing 3.8: tokenise sentences

```
     def generate_questions_mcq(keyword_sent_mapping,device,tokenizer,
    model,sense2vec,normalized_levenshtein):
        batch_text = []
        answers = keyword_sent_mapping.keys()
        for answer in answers:
            txt = keyword_sent_mapping[answer]
            context = "context: " + txt
            text = context + " " + "answer: " + answer + " </s>"
            batch_text.append(text)

        encoding = tokenizer.batch_encode_plus(batch_text,
    pad_to_max_length=True, return_tensors="pt")


        print ("Running model for generation")
        input_ids, attention_masks = encoding["input_ids"].to(device)
    , encoding["attention_mask"].to(device)

        with torch.no_grad():
            outs = model.generate(input_ids=input_ids,
                                  attention_mask=attention_masks,
                                  max_length=150)

        output_array ={}
        output_array["questions"] =[]
```

```
#      print(outs)
    for index, val in enumerate(answers):
        individual_question ={}
        out = outs[index, :]
        dec = tokenizer.decode(out, skip_special_tokens=True, clean_up_tokenization_spaces=True)

        Question = dec.replace("question:", "")
        Question = Question.strip()
        individual_question["question_statement"] = Question
        individual_question["question_type"] = "MCQ"
        individual_question["answer"] = val
        individual_question["id"] = index+1
        individual_question["options"], individual_question["options_algorithm"] = get_options(val, sense2vec)

        individual_question["options"] =  filter_phrases(individual_question["options"], 10,normalized_levenshtein)
        index = 3
        individual_question["extra_options"]= individual_question["options"][index:]
        individual_question["options"] = individual_question["options"][:index]
        individual_question["context"] = keyword_sent_mapping[val]

        if len(individual_question["options"])>0:
            output_array["questions"].append(individual_question)

    return output_array
```

The following code describes the main function that starts with an input text and performs the calls to the previous methods to achieve the goal of generating multiple-choice questions:

Listing 3.9: main algorithm

```
    def predict_mcq(self, payload):
        start = time.time()
        inp = {
            "input_text": payload.get("input_text"),
            "max_questions": payload.get("max_questions", 4)
        }

        text = inp['input_text']
        sentences = tokenize_sentences(text)
        joiner = " "
        modified_text = joiner.join(sentences)


        keywords = get_keywords(self.nlp,modified_text,inp['max_questions'],self.s2v,self.fdist,self.normalized_levenshtein,len(sentences) )


        keyword_sentence_mapping = get_sentences_for_keyword(keywords, sentences)

        for k in keyword_sentence_mapping.keys():
            text_snippet = " ".join(keyword_sentence_mapping[k][:3])
            keyword_sentence_mapping[k] = text_snippet


        final_output = {}

        if len(keyword_sentence_mapping.keys()) == 0:
            return final_output
        else:
            try:
                    generated_questions = generate_questions_mcq(
    keyword_sentence_mapping, self.device, self.tokenizer, self.model,
    self.s2v, self.normalized_levenshtein)

            except:
                return final_output
            end = time.time()

            final_output["statement"] = modified_text
            final_output["questions"] = generated_questions["
    questions"]
            final_output["time_taken"] = end-start

            if torch.device=='cuda':
                torch.cuda.empty_cache()

            return final_output
```

3.3 Third library

The third and last tested library, *lib_t5_3*, is based on the same technology that underlies the previous one, as the core of this implementation still relies on T5 transformers. There are some differences in this approach, though, as it makes use of a QA evaluator to rank the computed question-answer pairs and it also finetunes the model with two different datasets. As a matter of fact, together with SQUAD and MS MARCO that are used for library *lib_t5_2* too, the model has been pretrained with the following datasets:

- RACE (ReAding Comprehension dataset from Examinations) [19], a machine reading comprehension dataset consisting of nearly 28000 passages and 100000 questions from English exams in China, designed for middle school and high school students

- CoQA, a large-scale dataset to build Conversational Question Answering Challenge systems containing 127,000+ questions with answers collected from 8000+ conversations

The process starts by generating a list of model inputs from the input text, in the following form: "answer_token <answer text> context_token <context text>", where the answer is a string extracted from the text, and the context is the wider text surrounding the context. This is done through the *_prepare_qg_inputs_MC* function, that, by taking advantage of the Spacy library [20], performs Named Entity Recognition (1.2.2) on the text and uses extracted entities as candidate answers for multiple-choice questions. Sentences are used as context, and entities as answers. Returns a tuple of (model inputs, answers).

Listing 3.10: preparing answers

```python
def _prepare_qg_inputs_MC(self, sentences: List[str]) -> Tuple[List[str], List[str]]:
    spacy_nlp = en_core_web_sm.load()
    docs = list(spacy_nlp.pipe(sentences, disable=["parser"]))
    inputs_from_text = []
    answers_from_text = []

    for doc, sentence in zip(docs, sentences):
        entities = doc.ents
        if entities:

            for entity in entities:
                qg_input = f"{self.ANSWER_TOKEN} {entity} {self.CONTEXT_TOKEN} {sentence}"
                answers = self._get_MC_answers(entity, docs)
                inputs_from_text.append(qg_input)
                answers_from_text.append(answers)

    return inputs_from_text, answers_from_text

def _get_MC_answers(self, correct_answer: Any, docs: Any) -> List[Mapping[str, Any]]:
    """Finds a set of alternative answers for a multiple-choice question. Will attempt to find
    alternatives of the same entity type as correct_answer if possible.
    """
    entities = []

    for doc in docs:
        entities.extend([{"text": e.text, "label_": e.label_}
                          for e in doc.ents])

    # remove duplicate elements
    entities_json = [json.dumps(kv) for kv in entities]
    pool = set(entities_json)
    num_choices = (
        min(4, len(pool)) - 1
    )  # -1 because we already have the correct answer

    # add the correct answer
    final_choices = []
    correct_label = correct_answer.label_
    final_choices.append({"answer": correct_answer.text, "correct": True})
    pool.remove(
        json.dumps({"text": correct_answer.text,
                    "label_": correct_answer.label_})
    )

    # find answers with the same NER label
    matches = [e for e in pool if correct_label in e]

    # if we don't have enough then add some other random answers
    if len(matches) < num_choices:
        choices = matches
        pool = pool.difference(set(choices))
        choices.extend(random.sample(pool, num_choices - len(choices)))
    else:
        choices = random.sample(matches, num_choices)

    choices = [json.loads(s) for s in choices]

    for choice in choices:
        final_choices.append({"answer": choice["text"], "correct": False})

    random.shuffle(final_choices)
    return
```

The answers and context pairs obtained so far are then used to generate the questions, as shown in the following lines of code:

Listing 3.11: preparing questions

```
    def generate_questions_from_inputs(self, qg_inputs: List) -> List[str]:
        generated_questions = []

        for qg_input in qg_inputs:
            question = self._generate_question(qg_input)
            generated_questions.append(question)

        return generated_questions
```

This implementation also provides a QA evaluator, which evaluates the quality of question-answer pairs by assigning them a score and ranking and filtering them based on their quality.
The following script shows how the main algorithm works and how it combines together the functions described above:

Listing 3.12: main algorithm

```
 def generate(
        self,
        article: str,
        use_evaluator: bool = True,
        num_questions: bool = None,
        answer_style: str = "all"
 ) -> List:
        qg_inputs, qg_answers = self.generate_qg_inputs(article, answer_style)
        generated_questions = self.generate_questions_from_inputs(qg_inputs)

        message = "{} questions doesn't match {} answers".format(
            len(generated_questions), len(qg_answers)
        )
        assert len(generated_questions) == len(qg_answers), message

        if use_evaluator:
            print("Evaluating QA pairs...\n")
            encoded_qa_pairs = self.qa_evaluator.encode_qa_pairs(
                generated_questions, qg_answers
            )
            scores = self.qa_evaluator.get_scores(encoded_qa_pairs)

            if num_questions:
                qa_list = self._get_ranked_qa_pairs(
                    generated_questions, qg_answers, scores, num_questions
                )
            else:
                qa_list = self._get_ranked_qa_pairs(
                    generated_questions, qg_answers, scores
                )

        else:
            print("Skipping evaluation step.\n")
            qa_list = self._get_all_qa_pairs(generated_questions,
 qg_answers)

        return qa_list
```

Chapter 4

Tests

The libraries explored in chapter 3 work differently from each other, but all of them are supposed to give a similar output when run: a question-answers pair, with one correct answer and some other wrong to mislead the choice.
In order to test and evaluate these tools, a series of text paragraphs has been fed into them and the result has been manually evaluated assigning a score to determine how well they performed and possibly detect what they lacked.
The testing process has been conducted with paragraphs of different length. The idea behind this choice was to understand whether the models performed differently depending on their length, as the initial though was that the longer the text, the more data was available for the language understanding process. Three different thresholds have been established for such purpose, searching text paragraphs with:

- length greater than 1000 words; text with such length will be identified from now on as "long texts"
- length between 500 and 1000 words; these will be called "medium texts"
- length lower than 500 words; the expression "short texts" will be used from now on to refer to them

The three libraries have been tested with a total of thirty paragraphs, ten for each length. These texts have been chosen without any specific filter on the topic, and they have been taken from public online sources like BBC, CNN, National Geographic, etc.

4.1 Evaluation

As to the evaluation process, two possibilities have been taken in consideration: using an automated system or a human-based one. There are, actually, several algorithms that automatise the process of rating sentence that has machine-translated from one natural language to another, giving a score that represents the similarity of that sentence to a human one (the higher the similarity, the higher the quality of the generated text). Among those, on of the most used is BLEU (Bilingual Evaluation Understudy Score) [21]. Many question generation systems take advantage of BLEU metric to assess the quality of their output, but some studies show that these metrics are ill-suited for such purposes (e.g. Nema and Khapra's study [22]). For this reason a human-based evaluation system has been proposed to give a score to the models output. In particular, a double metric has been chosen to separately evaluate questions and answers, starting with a score of ten and decreasing it according to whether some specific conditions were respected or not. The idea behind this choice was to help understanding which areas were more critical, in order to focus and improve them.
For the questions, grammar, semantic and general sense of the phrase are the criteria used to assess them. Specifically, these are the details about the points:

- -3 for missing answer: three points have been deducted if the question does not really have an answer from the text
- 0 to -3 for grammar: up to three points subtracted from the final score if the sentence is grammatically incorrect

- 0 to -4 for general sense: up to four points deducted if the question does not make sense, either related to the context or not

As to the answers, criteria followed to give a score were based on looking at the presence of the correct answer, the number of distractors and their sense. Here follow the details for score assignment:

- -3 for missing correct answer: if, among the given answers, the correct one was missing, the total score was reduced by three points
- -1 for each missing distractor: given that distractors are the wrong answers to choose from, one point for each missing distractor has been removed (up to three points, as the questions were meant to have a total of four answers, the right one and three distractors)
- 0 to -4 for distractors sense: up to four points deducted from the total according to how the distractors make sense; ideally, one point for each distractor plus an extra one in case of bad result, but there was not a real strict univocal relation between each distractor and the deducted points

4.2 Results

Results for the tests on the three libraries are shown in the following sections. Each section displays a table with the obtained data, distinguishing the results for each category of texts (long, medium and short) and showing the following data:

- *Q_Num_Avg* will refer to the average number of questions obtained per text. Parameters have been set to generate ten questions per text
- *Q_Score_Avg* will refer to the average score assigned to the generated questions
- *A_Score_Avg* will refer to the average score assigned to the generated answers
- *Q_Duplicated* will refer to the average number of duplicated questions

4.2.1 First library

	Long texts	Medium texts	Short texts
Q_Num_Avg	5.8	3.20	3.20
Q_Score_Avg	10	10	10
A_Score_Avg	7.85	7.99	7.94
Q_Duplicated_Avg	0,40	0.50	1.60

Table 4.1: *lib_bert*_1 results

The table 4.1 shows the results obtained by the library *lib_bert*_1 with different length texts and gives interesting points for reflection.
First of all, it can be noticed that the average number of questions drastically decreased when moving from long texts to medium and short ones. Since the number of questions is strongly related to the number of keywords - as questions are simply sentences containing the extracted keywords -, such values represent the difficulty of the algorithm to find valuable keywords from texts with a length below a certain threshold - 1000 words in this case. One important fact related to the number of questions is also that some text paragraphs gave no result when running the algorithm: in particular, the text that had no output was exactly one among the medium ones, and there were two among the short ones, which significantly lowered the final computed average.
Another information that can be retrieved from this data is the extremely high values for the average questions score. This is actually no surprising, as the questions were phrases taken form the text, so no generation had been performed in that case, thus resulting in no processing for the model.

The third analised parameter is the average answers score. The table shows more than sufficient results for the answers, meaning that the system was able to return good answers and distractors for the given question. Values slightly varies from long to short texts, almost reaching a score of 8 out of 10; no trend can be found in this case.
Regarding the duplicated questions, short texts showed a lot of duplicated questions with respect to the other types, proving how challenge is for this algorithm to find keywords and sentences for the questions.

The following examples give more details on how the model performed, showing both successful attempts and sentences the model struggled with.

Successful attempts:

Context:
After his parents divorced, Obama's mother married another foreign student at the University of Hawaii, Lolo Soetoro of Indonesia.

Generated MC-question:
1) After his parents divorced, Obama's mother married another foreign student at the University of Hawaii, Lolo Soetoro of __________ .

a Philippines

b East Timor

c Indochina

d Indonesia

Question score: 10
Answers score: 10

Context:

Evidence from paleontology indicates that the penguins and the order Procellariiformes (albatrosses, shearwaters, and petrels) had a common origin.

Generated MC-question:

1) Some, such as the Adélie penguin, carry in their bodies trace amounts of pollutants, albeit in lower quantities than are found in many birds that live closer to humans. Evidence from paleontology indicates that the penguins and the order __________ (albatrosses, shearwaters, and petrels) had a common origin.

a Procellariiformes

b Acanthopterygii

c Actiniaria

d Acarina

Question score: 10

Answers score: 10

These multiple-choice questions obtained the maximum score for the answers (as well as the questions), being able to understand and offer reasonable alternatives to them: a country for the first one, and an animal group for the second one.

Failing attempts:

Context:
Otherwise, Etruscan literature is evidenced only in references by later Roman authors. The Architecture of the ancient Etruscans was derived from that of the Greeks, and went on to influence that of early Rome.

Generated MC-question:
1) Otherwise, Etruscan literature is evidenced only in references by later Roman authors. The Architecture of the ancient Etruscans was derived from that of the Greeks, and went on to influence that of early __________.

a High Command

b Rome

c Rome

d

Question score: 10
Answers score: 5 (-1 for missing distractor, -4 for distractors)

The question above has several issues. First of all, the algorithm managed to produce only three distractors (they were supposed to be four). Also, the generated answers show that the system failed to detect the right sense for the selected keyword: not only was one answer - the correct one - duplicated, but also the other one was out of context and with no relation with the topic of the text.

Another problem encountered in the results is the repetition of the same sentences with different hidden keywords:

Context:

On a Thursday night in December, most of the audience was from a black or ethnic minority background.

Generated MC-questions:

1) On a __________ night in December, most of the audience was from a black or ethnic minority background.

a Friday

b Thursday

c Feria

d Monday

Question score: 10

Answers score: 9 (-1 for distractors)

2) On a Thursday night in __________ , most of the audience was from a black or ethnic minority background.

a December

b December

c April

d August

Question score: 10

Answers score: 9 (-1 for distractors)

Such behaviour is not correct and it has to be prevented for two main reasons: in first case, obviously, showing the same question twice (or more) means that the total amount of useful questions is lower than the expected one; secondly, given the nature of these questions - which, as already said, are an exact copy of the sentences inside the text-, having duplicated questions also means that the missing keyword from one of them can be understood from the others and vice-versa, thus making the purpose of the quiz useless.

4.2.2 Second library

	Long texts	Medium texts	Short texts
Q_Num_Avg	7.2	4.8	5.8
Q_Score_Avg	7.97	8.21	8.17
A_Score_Avg	5.25	6.15	5.89
Q_Duplicated_Avg	0	0	0.4

Table 4.2: *lib_t*5_2 results

The table 4.2 with results from library *lib_t*5_2 does really present different data with respect to the ones of *lib_bert*_1.
Firstly, the average number of questions increased significantly, especially for medium and short texts, which now presents higher values by far. However, the trend already seen for the previous library is maintained, with fewer generated questions for medium and short texts (even though the latter are higher than the former). Going into details, this combination of algorithm and architecture did not present any outputs with no data, giving always at least one result even for short texts, which are usually harder to be understood by NLP applications as they offer less context.

While the number of questions showed better values, on the other hand the average questions score in this case is lower. As a matter of fact, the generated questions, which are not phrases taken from the text anymore, sometimes proved to lack general sense or to not have an answer that could have been guessed from the text. The values for this parameter are still sufficient and high enough to be considered good, ensuring a score of 8 out of 10.
One more parameter to take in consideration is the average score for the answers. This is actually the worst for this set of tests, as it presents values slightly above the sufficiency, with scores assessed around 5-6 out of 10. The reasons behind this low score are several, but among them there are:

- the algorithm struggled to detect the proper sense for the keyword, thus giving distractors not really related to what the question was asking

- an upstream issue on the question side, as sometimes the question was expressed in a way that it did not have a real answer that could have been detected from the available text

- the selected keyword was too specific and so the model was not able to find enough alternative answers

Lastly, as to the average number of duplicated questions, it must be said that it is optimal, with no duplicated questions for both long and medium texts, and very few for the short ones.

Here follow some outputs from this library.

Successful attempts:

Context:

Legend has it that Annie Oakley was such a skilled sharpshooter that she singlehandedly foiled train robberies, shot bears and panthers, and killed a wolf that already had her in its grip—or so claimed one 1887 novel based on her life titled The Rifle Queen.

Generated MC-question:

1) What animal did Annie kill that already had her in its grip?

   a Wolf

   b Bear

   c Coyote

   d Lion

Question score: 10

Answers score: 10

Context:
Dolphins are regarded as the friendliest creatures in the sea and stories of them helping drowning sailors have been common since Roman times.

Generated MC-question:
1) What are the friendliest creatures in the sea?

   a Dolphins

   b Killer whales

   c Orcas

   d Sharks

Question score: 10
Answers score: 10

Again, as for the first library results, these generated pairs show how the system took advantage of Named Entity Recognition to assign an entity to the keyword and select alternative ones as possible choices for the answer.

Failing attempts:

Context:

In 1853, a young tailor from Germany, called Levi Strauss, began working in San Francisco; Levi sold thick canvas to miners; the miners used the canvas to make tents.

Generated MC-question:

1) Who did Levi sell canvas to?

   a Mining power

   b Miners

   c Block rewards

   d Hashrate

Question score: 10

Answers score: 6 (-4 for distractors)

The question just illustrated is a clear example of misleading keywords, for which the system might detect the wrong sense. In the context of the original text, the word "Miners" refers to people that used to work in the mines, but the model actually retrieved the wrong meaning (cryptocurrency related). As a consequence, all the distractors were linked to that meaning, thus offering answers out of context.

Context:

Right now, apps such as VRchat work on a kind of self-regulating report system when it comes to moderation: players who continually receive complaints about their behaviour gradually lose privileges, and you can mute or block other users if they're annoying or harassing you. [...]
Given how endemic toxicity is online, from Twitter to Facebook to Call of Duty, I do not have high confidence that Meta or indeed anyone in the big tech world has either the will or the means to make the online world universally pleasant and safe. [...]
Once the headset is on you can browse from a few hundred apps and games designed for VR, many of which are social spaces designed for chatting to other people.

Generated MC-question:

1) What are VRchat?

   a Apps

   b Single App

   c

   d

Question score: 10

Answers score: 6 (-2 for missing distractors, -4 for distractors)

In this case the correct answer - apps - is present, but the library could not produce sufficient alternative answers.

4.2.3 Third library

	Long texts	Medium texts	Short texts
Q_Num_Avg	10	10	7
Q_Score_Avg	9.02	8.06	7.68
A_Score_Avg	5.72	5.54	4.58
Q_Duplicated_Avg	1	1	0.4

Table 4.3: lib_t5_3 results

Finally, the data about the tests conducted with the third library (lib_t5_3) are visible in table 4.3.
This approach proved to be the best for the number of generated questions. As a matter of fact, the library has been able to produce the maximum amount of required questions for both long and medium texts, and an average of 7 for the short ones, which, by far, outdoes the first two tested systems.
Despite the big amount of generated questions, the average score representing their quality assessed on optimal values, reaching a value of 9 for the long texts, and 8.06 and 7.68 for, respectively, the medium and short ones. Considering that these questions have been automatically produced and formulated, these values are good news and coherent to the ones already seen in 4.2.2. The results, though, seem to show a direct proportionality between the length of the text and the score of the generated questions.
The similarity with the second library can be seen also in the third row of the same table, where the scores about the answers are shown. These values are significantly lower than the others of the same table, with a trend already seen in the previous

section. The only difference is related to the scores of tests run on the short texts, which in this case could not overpass 4.58.

Successful attempts:

Context:

In 1853, a young tailor from Germany, called Levi Strauss, began working in San Francisco

Generated MC-question:

1) Where did Levi Strauss come from?

    a San Francisco

    b USA

    c Germany

    d France

Question score: 10

Answers score: 10

Context:

His Walden 7 housing project, which stands as a monumental terracotta termite mound on the outskirts of Barcelona, seems as radical today as when it was built in 1975.

Generated MC-question:

1) When was the Walden 7 housing project built?

   a Today

   b 1975

   c 80s

   d 2017

Question score: 10

Answers score: 9 (-1 for distractors)

Failing attempts:

Context:

They range from about 35 cm (14 inches) in height and approximately 1 kg (about 2 pounds) in weight in the blue, or fairy, penguin (Eudyptula minor) to 115 cm (45 inches) and 25 to 40 kg (55 to 90 pounds) in the emperor penguin (Aptenodytes forsteri).

Generated MC-question:

1) How much weight is the emperor penguin?

   a 55 to 90 pounds

   b approximately 1 kg (correct)

   c 14 inches

   d 25 to 40 kg

Question score: 10 Answers score: 5 (-3 for wrong correct answer, -2 for distractors)

2) How much weight is the emperor penguin?

   a 5 metres

   b 160 km

   c 55 to 90 pounds (correct)

   d 6 feet

Question score: 10 Answers score: 6 (-4 for distractors)

3) How much weight is the emperor penguin?

a 45 inches

b 5 metres

c 2 pounds (correct)

d 110 km

Question score: 10 Answers score: 3 (-3 for wrong correct answer, -4 for distractors)

4) How much weight is the emperor penguin?

a 25 to 40 kg (correct)

b 160 km

c 55 to 90 pounds

d approximately 1 kg

Question score: 10 Answers score: 7 (-3 for distractors)

In this case the algorithm failed multiple times, generating four questions asking the same thing but with different answers. The question itself is fine, but not only is the same question repeated more than once, but also the answers were incorrect: in two cases the picked correct answer is not the right one (questions 1 and 3), and some of the distractors were senseless (the question is asking about weight, but the answers contain length measurements). Also, as it can be noticed in questions 1 and 4, the same question contained multiple correct answers, as the text provided a double measurement and the model took both. This is a behaviour that also happened with the previous libraries, as often the process showed alternative choices that were equally right.

Chapter 5

Conclusion

The goal of this project was to explore the architecture behind Natural Language Processing tasks, with particular focus on the ones to automatically generate multiple-choice questions, and to test them in order to understand how well hey performed.
The study helped understanding the development of neural networks with respect to Natural Language Processing applications, with a continuous evolution from Feedforward Neural Network to Encoder-Decoder model, until reaching the transformers architecture that underlies state-of-the-art results for such subject.
Furthermore, several variants of these models have been tested on text paragraphs of different lengths, in order to asses their performance and highlight their strengths and weaknesses. More specifically, what arose from such analysis is that the process of generating questions from given input texts works pretty well, despite sometimes formulating them in a way that could bring up ambiguity or asking for answers that can not really be guessed from the original context.
Concerning the generated answers, instead, the algorithm still encounters issues and struggles with complex sentences, proving not to be reliable enough to always guarantee a sufficient number of meaningful and reasonable answers. In particular, the analised systems worked well with a specific set of keywords (months, places,

animals, etc.), but struggled with the ones that could be misconceived as they presented more than a single meaning. In order to improve this task, more test should be performed with different models and approaches (e.g., question answering techniques could be combined with the ones used in this project).
Additionally, the tests showed that some of the generated questions were sometimes duplicated, which can be solved by improving and refining the process that selects the keywords (avoiding the ones that lead to the same sentences), as well as some post-processing to filter out the results that do not match the expected constraints. The project will continue with further attempts in this direction, as the final goal would be for the process to be able to work with conversational texts in order to automatically produce quizzes for students after their language lessons.

References

[1] Prashant Johri, Sunil K. Khatri, Ahmad T. Al-Taani, Munish Sabharwal, Shakhzod Suvanov, and Avneesh Kumar. «Natural Language Processing: History, Evolution, Application, and Future Work». In: *Proceedings of 3rd International Conference on Computing Informatics and Networks*. Ed. by Ajith Abraham, Oscar Castillo, and Deepali Virmani. Singapore: Springer Singapore, 2021, pp. 365–375. ISBN: 978-981-15-9712-1 (cit. on p. 2).

[2] A. M. Turing. «Computing machinery and intelligence». In: *Mind* LIX (Oct. 1950), pp. 433–460. DOI: `https://academic.oup.com/mind/article/LIX/236/433/986238` (cit. on p. 3).

[3] Warren S McCulloch and Walter Pitts. «A logical calculus of the ideas immanent in nervous activity». In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133 (cit. on p. 9).

[4] Sepp Hochreiter. «The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions.» In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.2 (1998), pp. 107–116. URL: `http://dblp.uni-trier.de/db/journals/ijufks/ijufks6.html#Hochreiter98` (cit. on p. 13).

[5] Sepp Hochreiter and Jürgen Schmidhuber. «Long Short-Term Memory». In: *Neural Computation* 9.8 (1997), pp. 1735–1780 (cit. on p. 13).

[6] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. «Sequence to Sequence Learning with Neural Networks». In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger. Vol. 27. Curran Associates, Inc., 2014. URL: `https://proceedings.neurips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf` (cit. on p. 16).

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. «Neural machine translation by jointly learning to align and translate». In: *arXiv preprint arXiv:1409.0473* (2014) (cit. on pp. 17, 18).

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. «Attention Is All You Need». In: *CoRR* abs/1706.03762 (2017). arXiv: `1706.03762`. URL: `http://arxiv.org/abs/1706.03762` (cit. on pp. 19, 20, 22, 23).

[9] Ramsri Goutham Golla. $Generate_MCQ_BERT_Wordnet_Conceptnet$. URL: `https://github.com/ramsrigouthamg/Generate_MCQ_BERT_Wordnet_Conceptnet/pulls` (cit. on p. 25).

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding». In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: `10.18653/v1/N19-1423`. URL: `https://aclanthology.org/N19-1423` (cit. on p. 26).

[11] George A. Miller. «WordNet: A Lexical Database for English». In: *Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*. 1994. URL: `https://aclanthology.org/H94-1111` (cit. on p. 29).

[12] Ramsri Goutham Golla, Parth Chokhra, and Vaibhav Tiwari. *Questgen AI*. URL: `https://github.com/ramsrigouthamg/Questgen.ai` (cit. on p. 30).

[13] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. «Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer». In: *CoRR* abs/1910.10683 (2019). arXiv: `1910.10683`. URL: `http://arxiv.org/abs/1910.10683` (cit. on p. 30).

[14] Andrew Trask, Phil Michalak, and John Liu. «sense2vec - A Fast and Accurate Method for Word Sense Disambiguation In Neural Word Embeddings». In: *CoRR* abs/1511.06388 (2015). arXiv: `1511.06388`. URL: `http://arxiv.org/abs/1511.06388` (cit. on p. 31).

[15] Lakshay Sharma, Laura Graesser, Nikita Nangia, and Utku Evci. «Natural Language Understanding with the Quora Question Pairs Dataset». In: *CoRR* abs/1907.01041 (2019). arXiv: `1907.01041`. URL: `http://arxiv.org/abs/1907.01041` (cit. on p. 31).

[16] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. «BoolQ: Exploring the Surprising Difficulty of Natural Yes/No Questions». In: *CoRR* abs/1905.10044 (2019). arXiv: `1905.10044`. URL: `http://arxiv.org/abs/1905.10044` (cit. on p. 31).

[17] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. «SQuAD: 100,000+ Questions for Machine Comprehension of Text». In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 2383–2392. DOI: `10.18653/v1/D16-1264`. URL: `https://aclanthology.org/D16-1264` (cit. on p. 31).

[18] Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. «MS MARCO: A Human Generated MAchine

Reading COmprehension Dataset». In: *CoRR* abs/1611.09268 (2016). arXiv: `1611.09268`. URL: `http://arxiv.org/abs/1611.09268` (cit. on p. 31).

[19] Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. «RACE: Large-scale ReAding Comprehension Dataset From Examinations». In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark: Association for Computational Linguistics, Sept. 2017, pp. 785–794. DOI: `10.18653/v1/D17-1082`. URL: `https://aclanthology.org/D17-1082` (cit. on p. 37).

[20] Matthew Honnibal and Ines Montani. «spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing». To appear. 2017 (cit. on p. 38).

[21] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. «Bleu: a Method for Automatic Evaluation of Machine Translation». In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, July 2002, pp. 311–318. DOI: `10.3115/1073083.1073135`. URL: `https://aclanthology.org/P02-1040` (cit. on p. 44).

[22] Preksha Nema and Mitesh M. Khapra. «Towards a Better Metric for Evaluating Question Generation Systems». In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 3950–3959. DOI: `10.18653/v1/D18-1429`. URL: `https://aclanthology.org/D18-1429` (cit. on p. 44).