Between Extractive & Abstractive Document Summarization
- Project Report -

Andrew Zamai & Alessandro Manfè
Università degli Studi di Padova, Learning from Networks course 2022-23 final project
e-mails: andrew.zamai@studenti.unipd.it, alessandro.manfe.2@studenti.unipd.it

Abstract – **It is certainly exciting and challenging to find a way to automatically summarize long textual documents. Being able to produce a summary in a few seconds allows you to quickly understand the covered topics and immediately take the next steps (decide whether to read the complete article, make economical decisions, etc.). In this project report we present an approach we came up with to address these demands, trying to develop a complete software system that, given in input a (news) text article, outputs a short summary of it. The approach we will define is different from what current research papers propose, but we are optimistic about achieving some good results or, at least, demonstrating a new potential way of combining different DL models to deal with the task of document summarization.**

I. INTRODUCTION

A. What is it

Automatic summarization *is the task of producing a concise and fluent summary of a textual document while preserving key information content and overall meaning* (Allahyari et al., 2017). It is a well-noticed but still challenging problem. There are two main classic settings in this task: **extractive summarization** and **abstractive summarization**. The first one focuses on selecting sub-sentences from a given text to reduce redundancy (formulated as a classification problem). In contrast, the latter one follows the neural language generation task (it normally adopts the encoder-decoder architecture to generate a textual summary). In this project report we will present an approach that lies somewhere in between the two approaches, in short: we will try to find the most relevant sentences as in the extractive synthesis approach, then try to add new small pieces of information from other sentences, in order to produce new sentences that are similar to the directly extracted ones, but not necessarily identical.

B. Why should it be explored in a LFN course?

Many researchers **simply regard the documents to be summarized as sequences** and apply the encoder-decoder neural network models based, for example, on LSTM or Transformer architectures, to learn some latent representation, trying, either to perform sentence classification (*summarizing/not summarizing* sentence), or to produce a new summarizing text, e.g. using different expressions and paraphrasing direct speeches. As Wu et al. [1] claim, these approaches **fail to utilize the rich structural information** implicitly existing in the textual inputs and also makes difficult, if not impossible, to inject some **task-related knowledge** that may help in boosting performance. Especially when trying in summarizing long documents, as news articles (which may be composed of up to 50 sentences), RNN and Transformer based models may find difficult to discover correlations between sentences and topics that are far away between each other (a problem known as **long-dependency problem**). Moreover, as Jia et al. [3] report, these models, in the extractive setting, have no ability to consider the overlaps of the target selected summary, producing summary **sentences that are overlapped** (not orthogonal) and thus may be shortened more in one single sentence. Many researchers therefore believe that structural knowledge is useful in addressing these issues and propose using **Graph Neural Networks** as new tools to try to achieve a new state-of-the-art in synthesis systems.

II. APPROACHING THE PROBLEM

A. Dataset

Although automatic document summarization is typically approached as a supervised task (both in extractive and abstractive settings), one might be surprised at the lack of publicly available datasets. The time required by humans to produce *gold summaries* that can be used as target summaries is considerably expensive. More, if few but some datasets exist in English, the dataset in Italian are 2, and not even of good quality.

For our experiments we ended up in choosing the **CNN/Daily Mail dataset**. This dataset is made up of human generated abstractive summaries in bullets, generated from news stories in the CNN and Daily Mail websites as questions. In all, the corpus has 286,817 training pairs, 13,368 validation pairs and 11,487 test pairs. The source documents in the training set have 766 words spanning 29.74 sentences on average, while the summaries consist of 53 words and 3.72 sentences on average.

We remark the fact that the **summary sentences do not correspond perfectly to the source sentences** (thus not allowing a pure extractive setting) and neither can the sentences compose an abstract summary, since the speech phrases are reported as they are and not paraphrased. Therefore, for our system to achieve good metrics, it needs to be designed to **behave in an intermediate way** between extractive and abstracting settings. From here the title explanation.

B. Assumptions and task-related knowledge

When previously motivating the use of GNNs in automatic document summarization we emphasized how the injection of task-related knowledge into the model may potentially increase the performance of the model. Here we present the assumptions and the task-related knowledge we will try to encode in our system (these are not hard coded instruction about "how to summarize a document", but are just some considerations/hints we came up with, thinking about how a human approaches a synthesis task):

1. in news articles (but also more in general) **important things are at the top**: there are usually one or two concise starting sentences that present what the article is about and the rest of it are sentences in favor or against the presented thesis. This does not mean that the rest of the article is meaningless for the summarization purpose, contrarily **the rest of the article may help to inject more knowledge in the concise starting sentences**, with the result also of avoiding overlapped summary sentences;
2. **a concept is important** if it is **repeated many times** in the same or different paraphrased way or even by direct speeches reporting, **in any position** within the article (a sentence in the bottom may take up a concept that was briefly presented in the top part of the article, a long-dependency problem

difficult to cope with RNN/transformer models).

The take-home-message of the latter consideration is: **the system has to be able to discover, represent and later use semantic correlation between sentences within any position in the document**. The keypoint of the first consideration is: **the system has to discover the concise sentences which are more referred to** and **use them, together with other sentences, to produce summary sentences that overlap as little as possible**.

As example, if an article about skiing says that "Andrew is good at skiing" and, at the bottom of the article says "also Alessandro is good at skiing", we want our system to be able to discover the correlation between the two sentences based on the "skiing ability concept" and use them, not in a separated way to produce "Andrew is good at skiing", "Alessandro is good at skiing" (strongly overlapped sentences), but, hopefully, producing something like "Andrew and Alessandro are good at skiing".

III. DEFINING THE SYSTEM

With the above considerations and goals in mind, we came up with a system that we hereunder describe. It is made up of 2 sub-parts:

1. **Graph construction based on sentences correlation:** given in input a news article to be summarized, we construct a graph where **each node** represents **a sentence** and **direct weighted edges** are placed to represent correlation between the sentences (direction is used to keep sentences order, weights to quantify correlation);
2. **Summary generation part:** given the above manually designed graph, we use GNNs and Transformer Neural Networks to automatically learn how to use the graph topology and node sentences to generate a few summary sentences as output.

A. Graph construction based on sentences correlation

The idea of using a graph to represent relations between words or between sentences for summarization purposes can be already found in some recent research papers. For example, Jia et al. [3], propose to construct an heterogeneous graph which can be divided into three subgraphs (the word-level, word-sentence, and sentence-level subgraphs) and exploit Hierarchical Attentive Heterogeneous GNNs. The point is that the representation learning phase has to use the node level and topological structure of the graph: if **the graph is designed to help this phase**, probably it will perform better. With the knowledge-base considerations presented above we came up with **a novel graph design** aiming to represent correlations between sentences. The graph is constructed in the following way:

1. the input article is tokenized splitting it into sentences using "hard separators" like "."; Contractions are expanded ("won't" becomes "will not"). These operations are carried out using *nltk* library.
2. **each sentence a node**: each sentence is passed through a transformer encoder model (that we pre-trained in a way that we will later present), obtaining a vector of length 770 for each sentence. We call these vectors $CONCEPT(node_v)$.
3. Out Of Vocabulary Words must be managed, keeping a dictionary for each article and modifying both the source and target sentences.
4. the **edges** in the graph are **directed and weighted**. The direction allows to keep the sequences order within the document (and, in the later GNN representation learning phase, will **make the information flow from the bottom nodes to the top ones**, enriching them, as expressed in the consideration 1). The weights express the semantic similarity between two sentences. They are calculated by computing the **cosine similarity** between sentence embeddings obtained using the **SentenceTransformer model** provided by sbert.net. To reduce graph density an arc is retained only if its weight is greater than a threshold T (we identified a good value T to be the average of all correlation weights in a graph). Also, the max number of arcs in a graph is set to be equal to 3 times the number of sentences in the article.

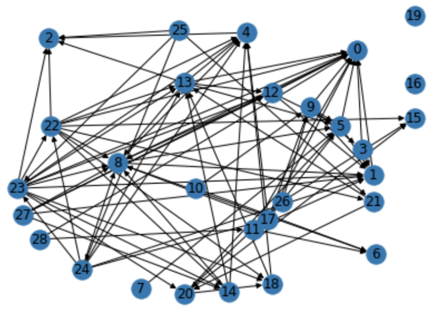

Fig. 1. Example of a graph constructed from a news article.

An example of a graph constructed from a news article is reported in Fig. 1.

B. Summary generation part

We have said that we designed and constructed the graph specifically to aid the summary generation part. Let us see how we intend to use this structured information to produce a summary.

The idea is to learn to extract the semantic concepts from each sentence-node and use the graph topology to learn how to aggregate and update these node representations, such to allow information to flow from the bottom nodes to the top ones, enriching them in information, as expressed in the consideration 1 in Assumptions and task-related knowledge section.

For **extracting the semantic concepts from each sentence-node** we use a **Transformer** neural network acting as encoder to produce a new vectorial representation for each node. We choose to use a transformer architecture over RNN models as they represent the current state-of-the-art in natural language tasks, allowing to better discover correlations within each sentence (self-attention mechanism), being more parallelizable during training and in practice dealing better with transfer learning. Being the sentence-node a length variable vector obtained by concatenating BERT word embeddings (+ position encoding) we will pad them to a fixed maximum sequence length and then obtain through the transformer encoder a fixed length vector of new dimension, that we call $CONCEPT(node_i)$.

Obtained the concept representation for each sentence-node we now need a way to **learn how to aggregate and update these node representations** such to allow information to flow from the bottom nodes to the top ones. For this task we think **GNNs are perfectly suited**. We have decided to define the AGGREGATE operator as:

$$\mathbf{m}_{N(u)}^{(k)} = \sum_{v \in N(u)} \frac{correlation_{u,v} \times CONCEPT(node_v)}{\sqrt{|N(u)| \times |N(v)|}}$$

and the UPDATE operator as:

$$\mathbf{h}_u^{(k+1)} = \sigma(W_{self} h_u^{(k)} + W_{neigh} m_{N(u)}^{(k)} + b)$$

The direction of the arcs (embedding sequence ordering within the document) should have allowed to bring information from the bottom nodes to the top ones (as desired from Consideration 1).

Notice: the training of the complete GNN model is still under development: the above AGGREGATE AND UPDATE functions are the first one we came up with and tried. In the section about the GNN complete model definition and training we will present several new AGGREGATE and UPDATE operators we came up with and tried.

Back to the system presentation, according to Consideration 2, we now can take the **top nodes with higher in-degree** to be the nodes from which we can generate the summary (we have somehow performed the extractive task in an unsupervised way). Let's call these nodes {CondensedRawConcept1, ... , CondensedRawConceptN}.

We have now to define **a way to combine** these CondensedRawConcept vectors to input them to a **decoder transformer**, which will finally output the summary sentences. Having the target summary sentences in the dataset an average of 3.75 sentences we have decided, for now, to pick as top nodes the 5 nodes in each article with higher in-degree score and stack them vertically in input to the decoder. The decoder transformer will now have to produce in output the target summary sentences using a new vocabulary (of eg. dimension 10k) and one-hot encoding. The transformer will be auto-regressive, with masked self-attention mechanism and the loss will be defined as the sum of cross-entropy at each time step between the predicted word and the target word.

Although this solution theoretically allowing for CondensedRawConcept vectors to construct the summary sentences order independent, the number of summarizing sentences is always fixed to 5, which is not true for all the articles in the dataset, leading us to believe that this top 5 nodes approach is not good enough. Later, we'll talk about other factors to think about and potential fixes for this problem.

Despite these factors, the above defined system seemed to be a reasonable candidate for implementation. Hereafter we are going to outline the ideas and trials over the past months that lead us to an almost working system.

IV. ENCODER AND DECODER PRE-TRAINING

We believe that an effective **pre-training of the encoder and decoder transformer modules** will be of great benefit when later training the complete model working on the articles structured as graphs. If the encoder and decoder are already "familiar" with the encodings of the initial and final vocabulary, the most part of the training will only concern the GNN weights, which will have to learn how to manipulate the representations of node-sentences, i.e. CONCEPT(node). We therefore started the project looking forward to all the steps that were needed to perform such pre-training task, with in mind what also would have been needed after in the complete model. We could not employ any pre-trained transformer from Hugging-face library as they all works with their dedicated tokenization and also the transformer we had to design was particularly handcrafted. Moreover **our belief is that our model working on an article structured as graph will need less parameters** than the big transformer models typical of big companies, of order of 150Billion parameters, like GPT-3 (our encoder will only need to encode sentences of a max length e.g setted to 50, while the big summarizing models somehow must work on much more longer sentences, if not at all the article level). Why did we call our encoder-decoder transformer 'handcrafted'? The idea is to pre-train the encoder and decoder providing in input, one by one **separately**, 5 random sentences (encoded through BERT) and requiring in output the same sentences concatenated in one single sentence, but in a new embedding (one-hot vectors of our defined Final Vocabulary). The tricky part is that the input sentences need to be forwarded through the encoder one by one, their vectorial representations stacked at the bottleneck and provided all together to the decoder to produce the final sentence as concatenation of all the input sentences. Such an architecture **allows exactly what we later will need:**

1. an encoder that takes one single sentence at the time and produce a 'concept level' vectorial representation for a node, i.e. CONCEPT(nodei);
2. a bottleneck representation that is obtained stacking 5 CONCEPT(nodei). In the final model these will be the CONCEPT(nodei) of the topk degree nodes in output from the GNN. Note that this will require some fine-tuning of the decoder as its inputs will be slightly modified by the GNN working on the CONCEPT(nodei).
3. a decoder that works on a stack of node representations {CONCEPT(nodei)} and produces in output a final sentence made up of more sub-sentences.

Two were the major difficulties in this encoder-decoder pre-training task:

1. **deciding which embeddings to use in input and output to the encoder and decoder respectively** and **how to deal with OOV words** (OOV words in news articles are pretty common since there are many proper nouns and cannot be simply neglected). Why 2 different input and output vocabularies? With BERT, we use a tokenizer at the sub-word level that has been pre-trained on a huge number of textual examples and thus we are able to cover almost every word by giving it a context-specific embedding in the sentence. Why not using BERT also as final vocabulary? We cannot use a subword level tokenizer as this would not allow us to keep track of the OOV words (as we will later present) and also we need to be able to go back from the embedding to the final word, thus one-hot encoding is needed. To keep the model small in the final number of parameters (decoder generator is a FFNN of ca decoder_out_embedding=770 x number of vocabulary words), the final vocabulary cannot be very large (we will use a final vocabulary of only ca. 10000 words): this will for sure affect the performance of the model but will keep the model trainable and within our computational resources.
2. defining and training such an "artisanal" encoder-decoder model.

Below we will now describe how each article is pre-processed and later how the encoder-decoder architecture has been defined.

A. PreprocessedDataset creation

Since the pre-processing takes about 1.5 seconds for each article and the preprocessed dataset is needed both for the pre-training task and later for the complete model training, to not slow down the training phases we decided to compute once for all a **Preprocessed dataset**. (When deploying the final model we will still able to preprocess a novel article on the fly). While the input-encoding phase is slow but light in memory (each sentence will be a matrix of numberOfWords=50 x input_emb_dim=770 floats16), the output-encoding using one-hot vector will be fast to compute but heavy in memory (ca numberWordsFinalSentence x finalVocLength: eg 250 x 10000 floats16). The **output encoding will be thus computed on the fly during training.** We have written a library of functions named *libraryForBuildingDatasetOptimized* which contains all the functions to produce the input-encodings for an article and output-encodings for the highlights target sentences (or article itself when pre-training the encoder-decoder weights in an auto-encoder fashion way).

The input encoding for an article works as follow:

1. contractions in the text such as "I'll" are expanded to "I will"
2. the article is splitted in sentences using *nltk tokenizer*
3. for each sentence:
   (a) tokenize at subword level using BERT pre-trained tokenizer (eg. word LONDON is splitted in L, ##ON, ##DON)
   (b) obtain the **embedding for each token summing the last four hidden layers of BERT**
   (c) merge tokens and their embeddings such to go back from token level representations to word level ones
   (d) **identify OOV words**: we consider OOV words proper nouns starting with capital letter (eg. Harry, but also Potter, to distinguish from potter profession) and special digits like $41.1 (which may be difficult to reconstruct by the decoder)
   (e) **for each article we define a vocabulary of OOV words** allowing to substitute each OOV word with a special token '[UNKi]' (where i is the UNK tag number) and substituting the BERT word-level embedding with a new defined embedding that is a-priory associated with that [UNKi]. We keep track of which UNK tag is associated with which word at article level such that if the same OOV word occurs later on in the article is substituted with the same UNKi tag. The decoder is now simplified in just being required to map [UNKi] tag in the input-embedding with the same [UNKi] tag but in the output-embedding. Using the unknownWordsDictionaryForAnArticle we can substitute back the UNKi tag with the OVV word it was place holder for. How the encoding for each [UNKi] is defined? The encoding produced by BERT is a vector of length 768 and the values in it span a range of real numbers between [-10.0 and +10.0]. We decided to expand this vector of 2 dimensions, obtaining a final embedding vector for each word of length 770 where: if the word is not OVV we keep BERT embedding with 00 in the last two positions, otherwise if OOV the embedding is 768 zeros and 2 values in the last two positions (a dictionary keeps track of which embedding is associated with which UNKi tag). We think that this solution will not disrupt BERT embedding as can be seen as moving along the 2 new dimensions for the UNK words, while leaving the 768 dimensional space unchanged.
4. an article is discarted if more than 100 UNK words are present in it, due to our computational resources available (100 UNK words are 100 proper nouns or group of digits, thus the number of articles that we will discard is very small).

A pre-processed article in PreprocessedDataset is a dictionary containing:

- a list of lists (for each sentence), where each list contains the words and UNKi tags that were placed.
- a list of Tensors where each tensor is a 2-D matrix for each sentence of dimension (numberOfWordsInSentence x inp_emb_dim = 770). (each node in the graph will be described by a matrix, or, more probably, will be passed through the encoder and thus each node-sentence will be described by a 770 length vector).
- the unknownWordsDictionaryForAnArticle

The **function to produce the output-embedding for a set of highlights target sentences** is more trivial:

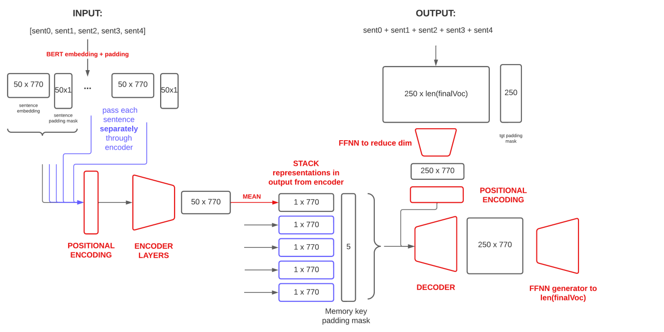

Fig. 2. Encoder-Decoder models pre-training.

1. expand contractions
2. replace each '.' with [SEP] tag if another sentence follows, otherwise [EOS] (End of Summary)
3. place ['START'] token at the beginning of the highlights
4. substitute each word with its one-hot encoding from FinalVocabulary (which contains also around 100 UNKi tags)
5. if the word does not belong neither to the FinalVocabulary nor is associated to an UNKi tags use '[UNK4BOTHDICT]'

B. Encoder-Decoder models pretraining

What we came up with is to pre-train the encoder and decoder modules in an "autoencoder" fashion way, that is requiring in output the same sentences provided in input, but with a few adjustments, first and foremost a different embeddings of the input and output.

Fig. 2 helps to visualize what is now presented:

INPUT: 5 sentences from an article, pre-processed as described above. With probability p=0.5 we pick the first 5 sentences, with probability 1-p we pick 5 random sentences: **this should help with regularization.** The sentences are passed **through the encoder one by one separately**. This turned out to be quite tricky to implement as we needed to pad/truncate all the sentences to a max_number_of_words length, create a padding mask for each sentence, append them such to have for all training samples always of same tensor size, but then later split them back to provide them separately through the encoder. Would have been also nice to be able to provide a variable number of sentences and not always 5, but this translated in having an input sentence to the encoder with src_padding_mask all to True and therefore in the MultiAttentionMechanism computing a softmax(0/0) that would give Nan.

BETWEEN the encoder and decoder (at the bottleneck): **the hidden representations from the 5 sentences are stacked vertically** and passed to the decoder. In the final model the GNN will work on the encoder output of each node-sentence (a vector of length 770); we will then pick the top nodes and stack them vertically. We pass this matrix (maxNumberOfStackedVectors x 770) to the decoder. When training the final GNN model the encoder will not be trained more, training only the GNN weights and decoder (since the inputs to it will be slightly modified by the GNN). If less than $k <= 5$ nodes we **use a memory_mask to mask unused representations**.

OUTPUT: all the sentences that were provided in input concatenated in 1 single sentence, encoded using one-hot vector of our defined FinalVocabulary. (In the final model these will be the highlights sentences).

Summarizing:

- Encoder task is to condense a single sentence representation (a matrix: nWordsInASentence x 770) in a vector of length 770.
- Decoder task is to start from a stack of condensed representations and produce 1/5 summary sentences (in the autoencoder case the 5 input sentences that were fed separately).

After a month of hard work and dedication, we were finally able to see some encouraging results. We produced a **PreprocessedDataset of almost 100.000 articles**: if we consider an average of 30 sentences per article, the number of possible 5 sentences that could be used as input-output pair was huge and allowed us to train a very complex transformer model of around **50.465.402 parameters**. We progressively increased the model complexity ending up with an encoder and decoder modules of 4 layers each, with 11 heads. Most of the number of parameters though is still on the fully connected NN to the one-hot output embedding. The final pre-training run of the encoder and decoder modules took 8 days, running on 6 RTX3090 GPUs (although we think the time bottleneck was at the pre-processed articles stored individually as pickles). The training and validation losses over training are reported in Fig. 3.

An example of "encoded, stacked and decoded back sentences" is reported below:

- GROUND-TRUTH sentences:
  1. tv5monde offers round-the-clock entertainment and news programming that reaches 260 million homes worldwide, according to the ministry of culture and Communications
  2. it functions under a partnership among the governments of france, canada and switzerland, as well as the wallonia-brussels Federation
  3. however, by late morning, a number of pages on the network's website had messages saying they were under maintenance
  4. on a mobile site, which was still active, the network said it was "hacked by an islamist group
  5. the outage began around 8:45 p.m paris time (2:45 p.m
- MODEL OUTPUT:
  1. french offers unk programming and news programming that reaches isis million people worldwide , according to the ministry of health and services
  2. it operates under a partnership between the governments of france , france and france , as well as the unk partnership
  3. however , by late morning , a number of pages on the network unk website had messages saying they were under maintenance
  4. on a mobile site , which was still active , the network said it was " unk by an al french group
  5. " the unk began around unk unk time (unk unk

By making the model more complex and, more importantly, by making the UNK tag classes more punishable (higher class weight loss), it is possible to improve the performance of the encoder and decoder models, but we live this to a future work, keeping these pre-trained models as a good starting point for the final GNN model.

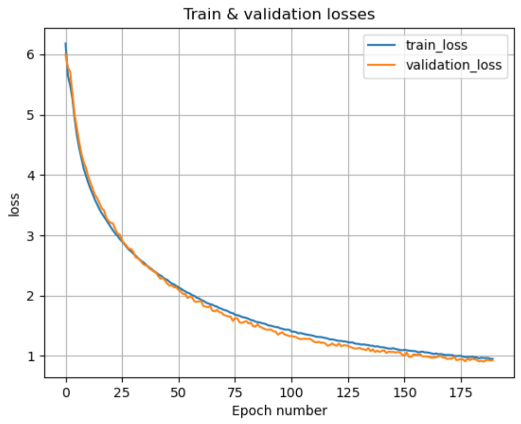

Fig. 3. Encoder-decoder pre-training losses.

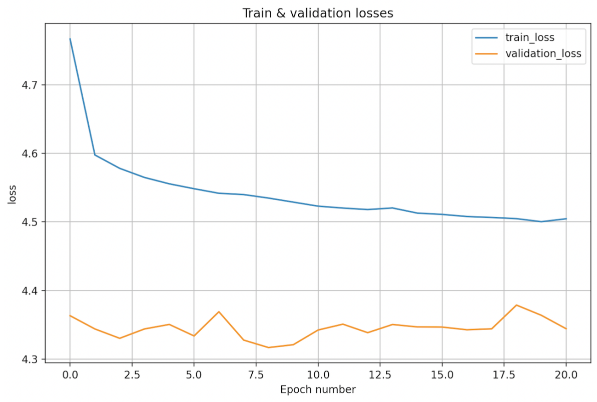

Fig. 4. Training losses in one of the final model's training trials.

V. GNN FINAL MODEL - DEFINITION AND TRAINING

We have in detail presented the overall system architecture, we have described how we implemented the articles pre-processing phase, how we constructed the sentence correlation graph for each article, how we implemented and pre-trained the encoder and decoder modules, the GNN AGGREGATE and UPDATE operators: all that is left to do is put everything together and train the final GNN model. But this is sometimes easier to say than to do. **To recap:**

1. each article is pre-processed through BERT encoder and a dictionary of (unknown word, TAG) pairs is created;
2. a graph of sentences correlation is constructed from each article and each node-sentence embedding is computed by passing the BERT-pre-processed sentence through our pre-trained encoder (this allows the GNN to work on vectors of fixed size 770, but of course will make the encoding phase no more trainable);
3. the GNN works by allowing nodes to exchange information, from bottom nodes to top ones, according to arcs direction;
4. the 5 nodes with higher in-degree are selected;
5. the stacked top 5 nodes representations are forwarded through the decoder, which outputs the summarizing sentences.

The problem we encountered was that the model was perfectly able to learn on a training set of 1000 samples (of course with bad generalization error), but as soon the number of articles was brought to 100.000 the model was neither able to lower its train loss nor its generalization loss. Moreover the DEI cluster was particularly overwhelmed in the last 2 weeks and few tries could be made. In Fig. 4 the training losses in one of the final complete model's training trials.

We are currently working to solve this problem and **we identified 3 main new solutions**:

- first, we believe that **the decoder complexity is too high** for the number of samples: it is true that has been trained on the same number of articles but the number of sentences that it could draw from was an average of 30 per article, while here works always with the top 5 node representations. We already tried to freeze the decoder layers, but this probably does not allow for the gradients to properly back-propagate. We will try using an **optimizer with different optimization strategies** for the GNN and decoder models respectively.
- **the top nodes selection**, working by picking the 5 nodes with higher in-degree, **is not the best strategy**. By manual inspection we can see that many times the system selects the important summarizing concepts and, if some concept is missing, the GNN message propagation should allow for the correlated sentences to add information, but other times it misses the correct summarizing concepts (it must be said that the golden summaries are human answers to questions, and thus, sometime, also not that really summarizing concepts are picked as highlights). Despite of this, the worst weakness of this method is that the number of summarizing sentences is fixed to 5, which does not reflect in the dataset. The considerations about the nodes with higher in-degree are still valid. A possible future solution could be the one of selecting the top 10 nodes with higher in-degree and **learn an automatic way to pick the summarizing nodes among these**: e.g. perform a node-classification task by comparing the node embeddings with the highlights embeddings obtained through the same pre-processing phases, i.e. learn binary classification in the same embedding space.
- **new AGGREGATE and UPDATE operators:** Recently, we developed new operators by drawing ideas from LSTM cells. In a LSTM cell 4 NNs act in a synchronous way to decide what to remove and what to add on a "cell state", depending on the current input and the hidden representation at previous time step. We think this is exactly what the message passing layer in the GNN model should do: **look at the child and parent concepts, decide what information to remove from the parent node and what to add from the child node.** The idea thus we are trying to copy and implement as MESSAGE operator in the GNN model is the one of having 2 NNs looking at the node and child concatenated embeddings: a NN will perform some non-linear transformations to this new vector (we call this NN *Add*), while another NN will act as faucet to decide what to add of this new representation to the current node embedding (we call this NN *Faucet*. A sigmoid function applied to *Faucet* makes the NN act a faucet deciding what amount from *Add* to add at each position in the embedding. We don't implement the forget gate as we think that the GNN working on the pair parent-child independently may make more children nodes to discard the same information. In any case, by allowing the NN *Add* transformation to have also negative embeddings should be sufficient to discard some information, if needed. Another feature to implement could be the one of making the message operator look at all children embeddings simultaneously.

LSTM based AGGREGATE operator:

$$\mathbf{m}_{N(u)}^{(k)} = \sum_{v \in N(u)} \frac{\sigma((W_{faucet} \times (u||v))) \times (W_{add} \times (u||v))}{\sqrt{|N(u)|}}$$

and the UPDATE operator as:

$$\mathbf{h}_{u}^{(k+1)} = RELU((W_{self} \times (h_u^{(k)}||m_{N(u)}^{(k)}) + b)$$

VI. CONCLUSIONS

Of course, after four months of work, it is unpleasant to observe that the effort does not yield fruitful results. We can only conclude positively that: there is no need for a boring metrics section if no system is functioning (yet). Joking aside we are not giving up in this project, we already identified and highlighted the points of weakness of our system and which could be possible solutions to be implemented. We are confident that the unfinished system won't be penalized, but appreciating more the ideas and efforts we put in this project, hoping through this report to have aroused the reader's curiosity about this new method of approaching the task of document summarization.

REFERENCES

[1] Lingfei Wu, Yu Chen, Kai Shen, Xiaojie Guo, Hanning Gao, Shucheng Li, Jian Pei and Bo Long (2021). *Graph Neural Networks for Natural Language Processing: a Survey.*

[2] Abigail See, Peter J. Liu and Christopher D. Manning (2017). *Get To The Point: Summarization with Pointer-Generator Networks*, CoRR journal.

[3] Ruipeng Jia, Yanan Cao, Hengzhu Tang, Fang Fang, Cong Cao and Shi Wang (2020). *Neural Extractive Summarization with Hierarchical Attentive Heterogeneous Graph Network.*

VII. CONTRIBUTIONS

Most of the ideas came up working and discussing together. The main different contribution to the implementation of the project was Andrew working on the encoder-decoder model pre-training and libraryForBuildingDataset, while Alessandro focused on constructing the graphs based on sentences correlation. The final GNN model was written by Alessandro using Andrew's pretrained encoder and decoder models. The GNN layer was written at four hands. The project report was written by Andrew. Too much time has been devoted to this project to count the weeks of work, but we believe it was worth it.