Automatic Abstractive Text Summarization : A deeper look into convolutional sequence-to-sequence networks

Auteur : Vermeylen, Valentin
Promoteur(s) : Ittoo, Ashwin; 12800
Faculté : Faculté des Sciences appliquées
Diplôme : Master en ingénieur civil en informatique, à finalité spécialisée en "intelligent systems"
Année académique : 2020-2021
URI/URL : http://hdl.handle.net/2268.2/13292

University of Liège
School of Engineering and Computer Science
Faculty of Applied Sciences

Automatic Abstractive Text Summarization

A deeper look into convolutional sequence-to-sequence networks

Author
Valentin Vermeylen

Supervisors
Ashwin Ittoo
Samy Doloris

Master's thesis carried out to obtain the degree of Master of Science in Computer Science and Engineering by Valentin Vermeylen

Academic year 2020-2021

Abstract

As the amount of information produced everyday continually increases, the desire for summaries containing only the most salient parts of the texts continues to gain traction. Even though the possibility to extract parts of texts and gluing them together already exists, we usually prefer fluent, human-like summaries.

That is the concern of the Artificial Intelligence subfield of Automatic Abstractive Summarization. Although the task is typically solved using recurrent neural networks, that architecture comes with several challenges, the biggest being the amount of time and computational power required to train the models. Fortunately, another less computationally intensive paradigm exists, based on convolutional networks, even though it has not been as extensively studied.

This thesis is concerned with that convolutional framework, and explores questions and assumptions that have not been answered previously, such as the advantages and drawbacks of using pretrained embeddings, or the tradeoff between performance gains and the added complexity of mechanisms such as reinforcement learning or pointing-generation. Experiments about the abstractiveness of the models, their fine-tuning on a different dataset, and their ability to capture long-distanced dependencies are also performed through the use of both the CNN/DailyMail dataset, and the XSUM dataset.

Those experiments show that using more convolutional blocks in the model makes sense up to a certain point, that the use of pretrained embeddings is advisable, as is the use of the pointer-generator network implemented in this work. The use of reinforcement learning is also advisable at the end of the model training.

Finally, this thesis is concluded with additional experiments that could be implemented in future works, as well as practical advises regarding the use of abstractive summarization in the context of general terms and conditions summarization.

Acknowledgements

First of all, I would like to thank NRB, and especially the team responsible for the submission of this subject, both for allowing me to work on such an interesting topic while broadening my understanding of the Natural Language Processing field, and for giving me access to their computational power, without which this work would not have been possible. I would especially like to thank Samy Doloris, for his guidance, his proofreading of this thesis and the time he dedicated to providing me with an access to the GPU cluster.

I would also equally like to thank Pr. Ashwin Ittoo for his guidance in the definition of the research questions, as well as his availability to answer my questions, read this thesis and refer me to relevant works.

Finally, I would like to thank my family who supported me through this peculiar year.

Contents

Chapter 1

Introduction

As the amount of information people have access to grows exponentially, the desire to automatically summarize large texts in order to extract the key ideas and concepts drew a lot of traction in the recent years. Among the practical examples already deployed, we can cite bots on websites such as Reddit automatically extracting the most important parts of news articles, effectively reducing the size of the text by 80 to 90%. Other fields where this idea could be of high interest are, for example, in the cases of long technical texts, of the ever-increasing amount of scientific publications, or in the case of general terms and conditions of use, which few people ever read. In all of those, having a concise summary of the text would result in important gains of time for many people; and in people actually reading what they are signing in the latter example.

That summarization task, given the name of *Automatic Summarization*, is part of the Artificial Intelligence subfield called Natural Language Processing (abbreviated NLP), and has been tackled by various methods falling under the categorization of either Abstractive Summarization, or Extractive Summarization.

As the name suggests, Extractive Summarization techniques aim at extracting the most salient (important) sentences or parts of sentences to create an shorter, information-dense version of the original text. This task produces summaries that are, qualitatively, far from human-generated ones, since no creative process needs to take place (no reformulation/-paraphrases/real summarization). The models only produce a patchwork of sentences and parts of sentences taken from the original texts.

Abstractive Summarization techniques, on the other hand, aim at summarizing texts by having a neural network generate an original summary, in a fashion that is much more human-like. Indeed, when we think of summaries, we usually picture texts that are not simply bits of the original document glued together, but rather an original, fluent creation.

With the advent of deep neural networks enabled by ever-increasing computational power, Abstractive Summarization overtook Extractive Summarization and its mainly statistical methods in recent years, in terms of research and publications.

Many network architectures have been proposed and tested on the task, but the most

common ones are based on Recurrent Neural Networks (RNN), a kind of networks particularly well adapted to data that presents itself as an ordered sequence, as is the case for texts, where the relative order of words is of great importance.

There are however several shortcomings with that kind of neural networks. First of all, their structure does not permit one to parallelize the training process in the time dimension. What this means is that, per batch of sentences, words from the same summary have to be generated sequentially. This obviously impairs the training time of the networks, oftentimes resulting in trainings that take weeks to produce results and that are computationally-intensive, which makes such architectures hardly handy for the individual. A second problem that occurs with such networks comes from the complex gradient flow arising during the backpropagation process. As weights are shared during the production of the whole sequence, the gradients must also flow through time (*i.e.* through the whole sentence), which can easily lead to exploding or vanishing gradients, impairing the training and performance.

To answer those shortcomings of RNN-based networks, several works have looked into Convolutional Neural Networks, inherently able to parallelize computations and showcasing a gradient flow that is less complex and more linear. However, works on such methods have been few and disparate and the preferred methods to tackle the task of Text Summarization still seem to rely on recurrent neural networks.

This thesis briefly answers the question of whether or not CNN-based Abstractive Summarization methods can achieve similar, or even better, results than RNN-based networks, and at what cost. Is the tradeoff between performance and training time interesting ? Does adding complexity to the CNN via concepts taken from other works in the field of NLP yield better results ? On top of those considerations, we will also conduct some experiments to answer questions that have not been addressed in CNN-based Abstractive Summarization papers, such as the role of pretrained embeddings in the quality of the generated texts, the influence of the number of convolutional blocks on the performance, or the inherent abstractiveness of the models.

Chapter 2

Necessary background

This chapter introduces the background necessary to understand the methods discussed both in the review of the literature and in my own experiments. The notions that are addressed in this chapter should be known to one who is already well-versed in the field of Natural Language Processing and can thus be skipped if the reader is comfortable with them.

1 Recurrent Neural Networks[43]

Recurrent Neural Networks are a special kind of artificial neural networks that are particularly well-tailored for processing data that are intrinsically sequential, such as text or speech, two kinds of data made from individual ordered entities (words). The main difference between Convolutional or Feedforward neural networks and Recurrent ones is that the latter have some kind of memory that allows them to influence the current output via data that they have already processed. This mechanism is critical in applications where the output of the network at a given time depends on what has previously been outputted or inputted, such as is the case for text generation, where subsequent tokens are not independent (The word to be produced at time t depends on what has already been produced in the $t-1$ previous timesteps).

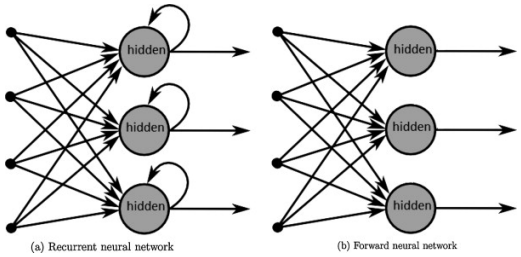

Figure 2.1: Difference between a feedforward neural network and a recurrent neural network.

Source: `https://www.researchgate.net/figure/Recurrent-versus-feedforward-neural-network_fig5_266204519`

As can be seen in Figure 2.1, the difference between feedforward and recurrent neural networks comes from the recurrent loop located in the neurons that allows the recurrent network to maintain an internal state from iteration to iteration, acting in practice as some sort of memory of the previous inputs. This is made clearer when taking a look at Figure 2.2, where the network is unrolled through time and where we can see that the recurrent loop feeds the previous state to the cell.

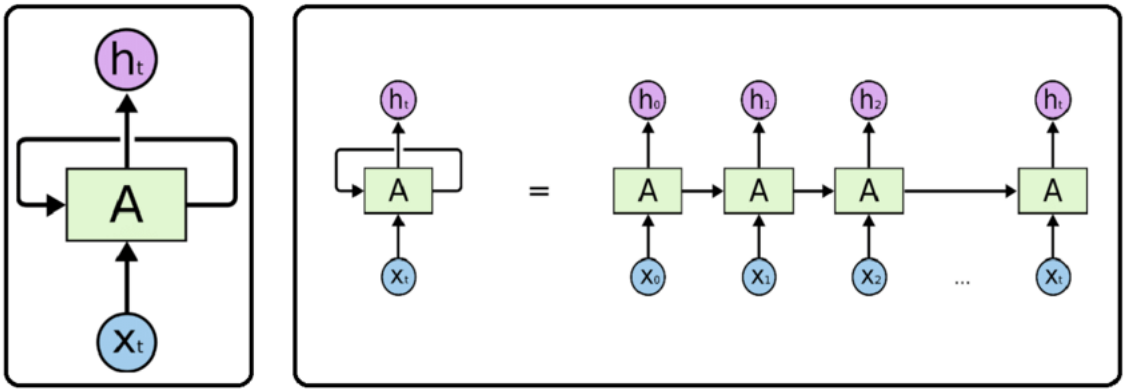

Figure 2.2: Unrolled Recurrent Neural Network

Source: `https://www.researchgate.net/figure/Schematic-representation-of-RNN-a-in-rolled-form-b-in-unrolled-form-from_fig3_343206534`

Another difference is that, in such networks, instead of simply using backpropagation, we have to use backpropagation through time (BPTT) in order to determine the gradients used in the learning process. The main difference between the two algorithms, which poses some problems in the case of RNNs, is that the errors used to compute the gradients must be summed across the input sequence, since the weights associated with a layer do not change across parts of the sequence being passed through them. All tokens produced in the several timesteps it takes to output a sentence therefore share a common weight matrix, which leads to the problems of vanishing and exploding gradients, as weights matrices are multiplied many times over.

1.1 Variant RNN architectures

Bidirectional RNN

Whereas the unidirectional RNN just mentioned only uses previous inputs in order to make predictions, bidirectional networks are also able to use future data. That is done by stacking two unidirectional RNN together, one reading the sequence forward and the other reading it backwards. The resulting hidden states h are simply concatenated together afterwards to produce a bidirectional context.

Long Short-Term Memory (LSTM)

Long Short-Term Memory networks aim at answering the aforementioned problem of vanishing gradients while also addressing the problem of long-term dependencies. Indeed,

RNN models do not capture well those kinds of far-distanced dependencies in the input. For example, the dependency of two tokens that are highly correlated but far from one another in the input sentence will most probably not be picked up by a regular RNN. To solve that issue, LSTM networks use multi-gated cells in the hidden layers of the network, able to control the flow of information and to restrict the magnitude of the gradients, eliminating the problem of vanishing gradients. The issue of exploding gradients, however, are not being taken care of by that kind of network but must be eliminated through weight-clipping.

2 Sequence-to-sequence network

Sequence-to-sequence networks are a special type of RNN networks designed to solve complex linguistic tasks such as machine translation or text summarization. They are encoder-decoder architectures in which, historically, both the encoder and decoder were recurrent networks, most commonly LSTM. They can however be CNN-based, as will be seen further in this work.

The role of the encoder is to read the input sequence and produce a context vector that contains some kind of summary of the sequence. Outputs from that part of the network are typically discarded, as only the hidden states representing the context are of interest. This is illustrated in Figure 2.3, where the hidden states are represented by h_i and c_i.

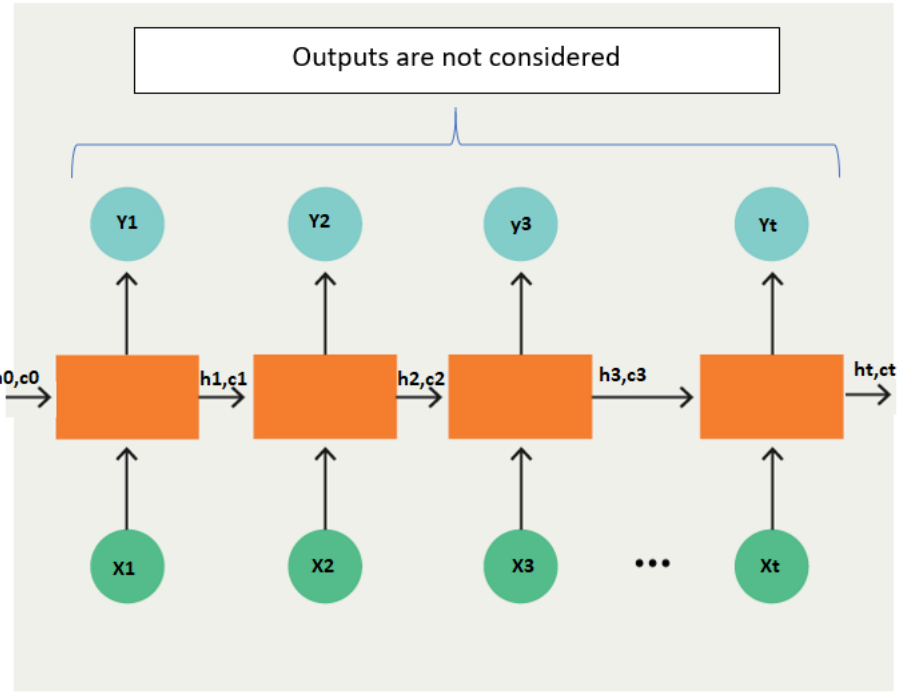

Figure 2.3: Encoder part of a Sequence-to-Sequence network[3]

The decoder part of the network, on the other hand, is in charge of producing the output of the whole architecture. In the case of the summarization, it will be fed the previously decoded tokens as well as the context vector that has been produced by the encoder (it

thus does not get fed the input themselves, but some kind of fixed-size summary of it typically in the form of h_t). An example of such a network is provided in Figure 2.4.

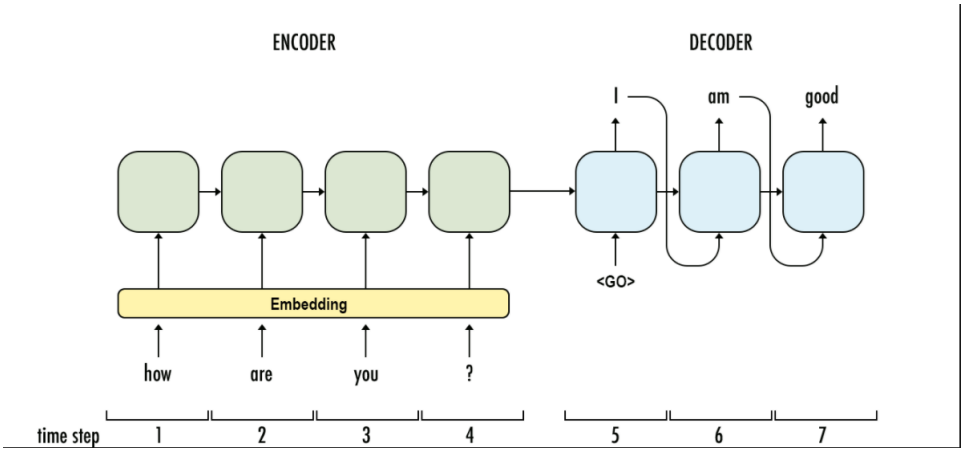

Figure 2.4: Example of a Sequence-to-Sequence network

Source: `https://stackoverflow.com/questions/47400126/optimizing-the-neural-network-after-each-output-in-sequence-to-sequence-learnin`

Although these networks work well for sequential data, they are still subject to some problems :

- The memory of this architecture is very limited. As the whole input sequence is crammed in a fixed-size context vector, the encoding tends to be lossy for longer sequences, and proves unable to capture longer-range dependencies.
- The deeper the network, the harder it is to train, and the more the effect of vanishing or exploding gradients are felt, especially in RNN-based networks. However, deep networks are necessary to obtain quality summaries.

3 Attention

In order to answer the aforementioned problems, a mechanism called **attention** can be implemented in the sequence-to-sequence model. The idea for attention stems from the fact that not all input tokens are of equal importance for predicting each output token. In attentional networks, the context vector that is fed to the decoder at each decoding step is a weighted sum of all the encoder hidden states. The decoder can thus rely on a dynamic context vector that will focus on different input tokens at different time steps. In other words, that mechanism enables the network, during the decoding phase, to know which tokens from the input it should pay attention to to produce a given token, and those important input tokens change at each decoding step.

4 Convolutional Neural Networks[11]

Convolutional Neural Networks (or CNNs) are a type of neural networks that are most often used in the field of Computer Vision, since they are able to easily identify patterns

in an image and extract the relevant features.

CNNs are composed of 3 main types of layers :

- The *Convolutional layer* : This is the main building block of a CNN, where most of the computations occur. It receives the input as a matrix and will then slide a fixed-sized kernel across it, allowing the network to check whether or not the feature that should be captured by that kernel is present in the area under examination. That step is called a convolution, and one convolutional layer will typically have hundreds of kernels, all looking for different features in the input.
- The *Pooling layer* : This layer is used to perform some dimensionality reduction (reducing the number of parameters of the input) by applying an aggregation function to its input.
- The *Fully-connected layer* : This layer acts as a part of a multi-layered perceptron in which all nodes are interconnected. It allows to perform the final computations in the network and to shape the output in the desired form.

Although CNN are usually used in the field of Computed Vision, their use in the field of NLP makes sense, as the input matrix can represent embedded words, and the kernel of size n that passes across it effectively enables the network to read multiple (n) words at once. This means that the network is inherently able to concurrently process one word and $\frac{n}{2}$ words on each side, which amounts to being able to process the central word and its bidirectional context. We thus already see an advantage of using CNN-based sequence-to-sequence networks instead of RNN-based ones : the bidirectionality that requires almost twice the parameters in the encoder of RNN-based networks is given for free in their convolutional counterparts.

5 Evaluating Natural Language Processing tasks

In order to evaluate tasks such as machine translation and text summarization, several metrics have been devised., such as WER, BLEU and ROUGE.

The first metric used to assess such tasks was the Word Error Rate (WER), *i.e.* the proportion of common words between the prediction and the ground truth. However, one obvious problem with that metric is that there was no sense of syntactical order taken into account. The sentence

The man ate the apple

had the same WER score as the two following sentences :

The apple ate the man
The the ate man apple

Which made little sense when trying to evaluate the quality of generated texts. Fluency was not taken into account; the metric just relied on some kind of accuracy.

Then came the BLEU score (BiLingual Evaluation Understudy), initially developed for text translations. Its formulation is :

$$\text{BLEU}_\text{n} = \min\left(1, \frac{\text{output-length}}{\text{reference-length}}\right)\left(\prod_{i=1}^{n} \text{precision}_{\,i}\right)^{\frac{1}{n}}$$

It is not perfect but offers 5 advantages[1] :

- It is quick and inexpensive to compute
- It is intuitive
- It is language-independent
- It correlates with the human way of evaluating such tasks
- It is now widely adopted and the preferred method to benchmark results

The idea behind BLEU is to compute the fraction of n-grams[1] in the generated text that appear in the ground truth (unigrams, bigrams, trigrams etc), while penalizing too short texts through the first term in the formula. Although the positions of the n-grams in both texts are not relevant to the score, the relative order of tokens inside the n-grams is, and so the higher-level n-grams that show a match, the closer the generated summary is to the target one and the better the fluency and order of words. One obvious drawback of this method, which will be brought again in section 2 of chapter 4 is that this metric tries to force the summaries to be as close as possible to the target one in order to get a high score, which does not make a lot of sense in an abstract summarization setting where the goal is specifically for the network to produce new original sentences, as opposed to Extractive Summarization where some sentences had to be selected while other not, and no creative sentences were permitted. Using the BLEU score, paraphrases, synonyms and correct but differently formulated sentences will be penalized as if they were flat out wrong, although they may make sense. A perfect score of 1 is therefore usually not obtainable, not even for humans, who typically score in the low 0.3 range when evaluated using BLEU. That problem is even greater for automatic summarization than it is for machine translation (for which the metric was created), as the latter has less leeway to improvise. Translations are often close to 1-to-1, whereas summarization is much further away from that bijection.

However, we do not have much choice, as there is no other metric that could solve such problem, currently and probably in the future. Aside from using human labour to evaluate the quality of the summaries, we are stuck with using BLEU, as everyone else does for the task. Furthermore, the fact that BLEU has become the *de facto* metric to measure performance in many NLP tasks would make it hard to replace, as all performances in the literature and all baseline results are reported using the BLEU score, or its counterpart, ROUGE.

ROUGE score

Whereas the BLEU score measures how much the n-grams in the machine-generated summaries appear in the human reference summary, the original ROUGE (Recall-Oriented

[1]n-grams correspond to contiguous sequences of n tokens in a text.

Understudy for Gisting Evaluation) score measured the reverse, *i.e.* how much the n-grams in the reference text appear in the machine-generated one. In effect, the BLEU score measures some kind of **precision** whereas the ROUGE score measures **recall**[40], as its name implies.

More precisely, the usual implementations of ROUGE measure the precision, the recall and the F1-score of predicted sentences with regards to target ones. ROUGE-N measures those metrics with respect to only the N-grams, and ROUGE-L measures the longest matching sequence of words.[5]

Both kinds of scores are correlated and will be used in the experiments.

6 Embeddings

One section of the experiments will be concerned with the concept of word embeddings, and they are thus introduced here.

Word embeddings are a way to represent words and sentences in a compact and consistent way. Neural networks work with numbers, but would be unable to handle words as such. In order to provide texts to the networks, we have to somehow transform the words or the characters into something that the network would be able to work with, *i.e.* tensors of numbers. Word embeddings allow one to do just so by mapping words to a numerical vector of fixed size that represents the word in a lower-dimensional space, since the dimension of the embedding would typically be far lower than the dimensionality of one's whole vocabulary. Otherwise, we would simply have to represent the words as one-hot vectors. The reason we do not do so is to reduce the input dimensionality, but mainly because one property of word embeddings is that words that are close in meaning will have close embeddings and consistent relationships, as is illustrated in Figure 2.5. Word embeddings should allow the network to predict words that are close to a given word and to capture semantic information automatically.

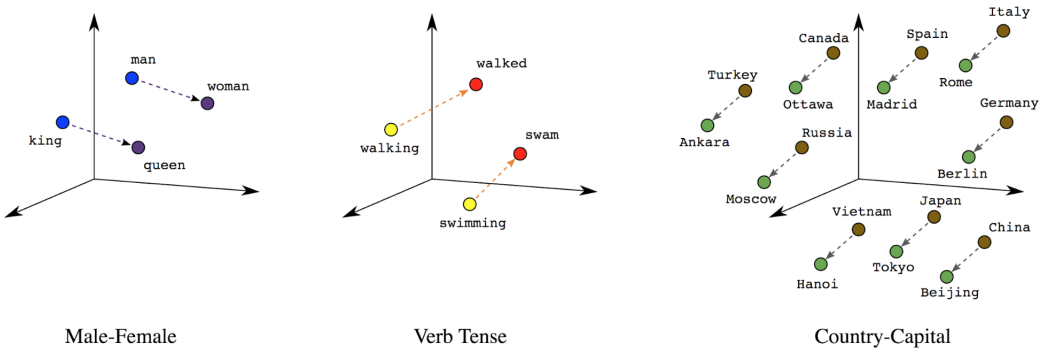

Figure 2.5: Word Embeddings[21]

For example, an embedding of size 100 will represent any given word with a vector of 100 numerical values, effectively reducing the dimensionality of the input. Such embeddings can either be learned on-the-fly by the network (in such cases, the embedding layer

contains weights that are trainable), or they can be pretrained.

6.1 Pretrained embeddings[45]

Pretrained embeddings are embeddings that have been trained on a given corpus and for a given task, and that are transferred and used as is in your network, for datasets and tasks that are not necessarily those the embeddings were created on. In this work, we will mainly use the Word2Vec and GloVe word embeddings, and therefore mention only those here.

Word2Vec

In this scheme, words are initially assigned either a random or a one-hot vector. The training corpus is then enumerated. Two models are possible. In the *Continuous Bag of Words* model, the vector representations of the neighboring words will be made to resemble that of the central word. In the *Skipgram* model, we do just the opposite, and we aim at making the representation of the central word closer to the neighboring word representations. The Skipgram model has been shown to produce more meaningful embeddings[45] and is therefore the one used in the experiment comparing the word embeddings.

The above process is then repeated iteratively until we get the desired word embeddings.

GloVe

GloVe is the result of applying another method to create word embeddings. In that method, we iterate over a corpus and build the co-occurence matrix containing the co-occurencces of each word with all other words in the corpus. This means that words separated by n words will get a co-occurence value of $\frac{1}{n+1}$.

Concerning the word vectors, they are initially assigned randomly, then their relative distances are modified so that words that occur frequently together or have a high co-occurence value but are far apart are brought closer together; and words that are close in distance but rarely seen together are brought apart.

That process is repeated many times until we end up with a set of vectors that approximate the information contained in the co-occurence matrix.

This method is the preferred one in the literature, since it is able to capture both semantic and syntactic information better than Word2Vec. As a result, this is the embedding that will be used by default in the experiments, unless stated otherwise.

Selecting the corpus to train the embeddings on

If the embeddings were trained by a single person on a small corpus, one problem would arise, namely that the embeddings would be highly biased towards the corpus they were trained on. This would be problematic, since embeddings are made to be used on other tasks and on new corpora.

Fortunately, word embeddings of both GloVe and Word2Vec kinds are available online, and were trained on a variety of corpora, the main one being the entirety of Wikipedia in a given language. Training the embeddings on such an extensive text source is the best way to reduce the bias problem aforementioned, but is obviously not available to the lone computer scientist given the huge training cost associated with such a task, and are thus typically offered by technology giant such as Facebook or Google.

7 Decoding process and Beam Search

Whenever a Natural Language Processing model has some text as its output, one is faced with several choices for the final text that will be produced by the model. The networks will typically generate a probability distribution over the output vocabulary (if you set the output vocabulary of the network to 1 million possible words, the model will output 1 million values, with the largest ones corresponding to the preferred words). One simple algorithm to handle that distribution and produce a single word is to use a **greedy search** algorithm, *i.e.* always selecting the word that corresponds to the highest value in the probability distribution. This method is often used because it is quick, simple to implement, and results across different models are still comparable so long as they all use such a search algorithm.

However, the final sentences that are produced are not optimal, since the greedy search is a heuristic method. A better algorithm is the so-called **Beam Search** algorithm, that builds upon the greedy search.

Beam Search improves the naive greedy search in 2 ways :

- First of all, Beam Search will select the best N (a parameter known as the beam width) words at each step (for example, the first word to be decoded will be a list of N possible words in this paradigm), whereas greedy search selects the single best word per step.
- Secondly, Beam Search does not consider each individual word in the final sentence as independent, but rather considers the joint probability of the sentence it is building. In comparison, in greedy search, once a word has been outputted, it is set and does not influence the probabilities of the remaining part of the sentence.

Since, as humans and evaluators of the model, we are interested in the complete sentence, it makes sense that a decoding procedure that considers the whole sentence jointly would produce better results than a naive heuristics simply selecting the most probable word at each time step without regards for what will be outputted afterwards or what has previously been outputted[17][2]. An example of beam search with a beam width of 3 is provided in Figure 2.6. How the leaves are selected to be added to the queue is not how it is done for text summarization, but should illustrate the point regardless.

However, although this procedure will produce better summaries than the naive greedy search, it is also much more expensive and could not be done for all models and exper-

[2]The reader interested in a more detailed explanation of the inner working of the Beam Search algorithm is welcome to visit this intuitive explanation : https://towardsdatascience.com/foundations-of-nlp-explained-visually-beam-search-how-it-works-1586b9849a24

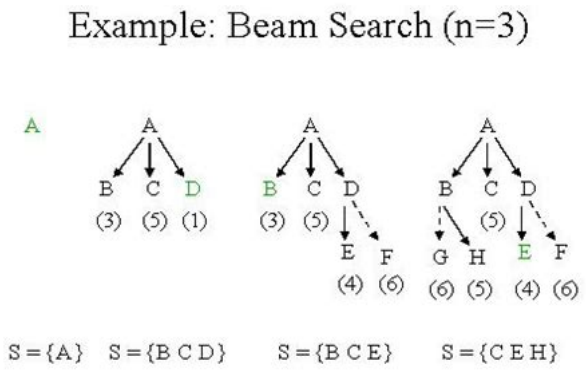

Figure 2.6: An example of beam search with a beam width of 3

Source: `https://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/46927-f97/slides/Lec3/sld023.html`

iments. Since the greedy search produces results that are already satisfactory and that enable comparison between models, that is the decoding scheme that will be used in most of this work. Whenever applicable, I will explicitly mention that the results have been obtained through beam search.

Chapter 3

Literature Review

This chapter aims at introducing and summarizing the literature produced in the field of Automatic Abstractive Summarization, focusing particularly on Convolutional Neural Networks methods, as it is what this thesis is concerned about. The timeline for this chapter is partly inspired by the work of *Shi et al.* [38], as well as personal research about the relevant literature. This chapter introduces the various papers related to the thesis' subject while staying relatively high-level in the explanations of the methods.

1 Early works - Extractive methods[8]

As previously mentioned, the earliest works concerned with automatic text summarization typically utilized *extractive* methods, as those researches were conducted long before the Neural Network boom that was seen in recent years and which was enabled by ever-improving hardware. The use of fully data-driven methods as we see today was simply not achievable at the time automatic text summarization was first explored.

In extractive methods, the summary that is produced is a patchwork of sentences and parts of sentences coming from the input document and simply glued together, hence the name *extractive*.

Without entering into too much details, extractive text summarization was, in the early days, done using rule-based, deterministic algorithms that ranked the various sentences composing the text based on their perceived importance, before extracting the top n ones and gluing them together. Such an algorithm was called an *importance evaluator*, and it used both a hard-coded set of rules for determining the importance of sentences, as well as an "encyclopedia" that contained knowledge related to a specific domain. It was a far-cry from our current methods, where no *a priori* knowledge about the language or the corpus is fed to the networks.

As time went by and more research came out of the field, developments were made that allowed one to better determine the importance of sentences in a document. One such method, for example, measured the importance of a sentence by the order of relatedness it has with the other sentences (the more related one sentence is to the set of other sentences,

the more important it is). However, those systems still required some knowledge about the subject of the texts to be summarized, and were still extractive by nature.

Many subsequent works expanded over that framework, introducing more complex statistical models for the task.

Finally, the advent of neural networks enabled important developments in the field of automatic summarization. The paradigm shifted, as the networks were then trained on a large corpus of texts and their corresponding summaries, and learned to infer such summaries. No rules had to be fed to the network, as it learned the relevant ones itself. Thus was opened the road to the subfield of abstractive text summarization.

2 The First Seq2Seq model applied to Abstractive Text Summarization

The first work about abstractive text summarization entirely driven by data and not making use of *a priori* human knowledge was published in 2015 by *Rush et al.*[33]. They recognized that extractive methods were inherently limited and that designing an abstractive generalization-style summarizer had been proven both challenging and unsatisfying. However, witnessing the success *Bahdanau et al.* who applied the Sequence-to-sequence model to Machine Translation the year prior[7], the authors set out to applying that framework to the task of abstractive summarization.

The idea behind their model was to combine a neural language model as a decoder with a contextual input encoder. That encoder was similar to the attentive encoder introduced by *Bahdanau et al.* and also made use of attention.

As opposed to previous works in the field of automatic summarization, the authors incorporated less linguistic structures but relied instead on learning such language features from the data directly, paving the road for future data-driven abstractive summarization methods while showcasing results that were either better than or on par with current methods at that time.

3 Introducing Recurrent Neural Networks in the Decoder

Chopra et al.[10] later (in 2016) built upon the work of *Rush et al.* and replaced the feed-forward neural language model used as decoder by a conditional recurrent neural network, more fitted to NLP tasks. They also encoded the positions of the tokens appearing in the input text in the encoder part of their network, and the results obtained on the summarization datasets showed an increase in performance when compared with the network the authors extended upon.

4 Handling various shortcomings

Recognizing several critical problems with the models presented before, *Nallapati et al.*[24] proposed some new mechanisms to improve the overall quality of the generated summaries.

The first problem they set out to solve was the one of identifying key concepts and entities in the input document. Their thesis was that to accomplish such a goal, they would have to go beyond the word-embedding-based representation of the document and capture linguistic features such as parts-of-speech, named-entity tags and words statistics in the document (in the form of term frequency or IDF for example). They therefore created additional embedding matrices for the vocabulary of each tag-type, one-hot vectors for continuous measures such as TF, etc. The final embedding vector was simply a concatenation of all those elements. However, although the idea was interesting, it was not used extensively in future works, probably because it reverted the trend to incorporate as little *a priori* human knowledge as possible in the networks.

The second problem they identified was the modeling of rare / out-of-vocabulary (OOV) words. Typically, words that are not part of the training data cannot be modeled at test time, as they are either transformed into unknown tokens in the embedding phase of the encoding or left as is but not part of the output vocabulary anyway since the latter is fixed and determined at training time. One obvious way to solve that problem would be to increase the size of the output vocabulary to encompass all words in a given language, but we can directly see that this idea would not hold practically, due to the ever-increasing size of a language's vocabulary and the inability to have enough training data containing all words anyway.

The way such out-of-vocabulary words were historically handled was thus to replace them with an `unknown` token at test time, resulting in sub-optimal and sometimes non-legible summaries.

One intuitive way to handle such words, which is the one used by the authors, is to simply point the location of the rare word in the input text for the decoder to copy in the decoding process, if that word is perceived as important through the mechanism of attention. That is implemented through a switching decoder/pointer architecture that is depicted in Figure 3.1.

The decoder is equipped with a *switch* that decides between using either the generator or the pointer mechanism in order to produce the next token at every decoding step. When the switch is turned on, the normal behaviour of the encoder-decoder is altered and the word that is outputted corresponds to a rare word at a given position in the input text.

Finally, the last problem that the authors tackled is that of capturing hierarchical structure in a document. When the input text is very long, it is important to identify the key sentences from which the summary can be derived on top of identifying the important keywords. The way the authors dealt with this idea is through the use of two RNN networks on the input side, one dealing with the word level and the other one dealing with the sentence level. The attention mechanism operates at both levels, and the word-

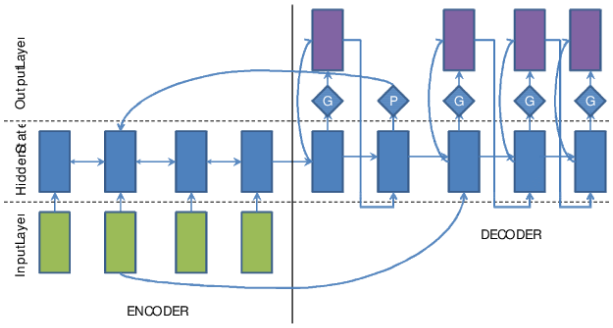

Figure 3.1: Switching generator/pointer model[24].

level attention is then re-weighted by the corresponding sentence-level attention.

Furthermore, the authors created the CNN/DailyMail dataset. Previous works were evaluated on datasets such as DUC or Gigaword, which only consisted of one-word summaries. Although such limited datasets were enough for training simple networks and have a common benchmark, they would not reflect a real-world use for automatic abstractive summarization. The CNN/DailyMail dataset was the first dataset comprising multiple sentences summaries originating from news articles. Most works on the field published after this article then used this dataset as benchmark, either on top of the previously mentioned ones, or by itself entirely.

5 Multi-sentences summarization

Once the CNN/DailyMail dataset was released, several works began investigating the possibility of generating longer, multi-sentences summaries. However, several obstacles quickly arose.

Firstly, the longer the source document, the less reliably the summarizer is able to extract salient, useful information from it. Furthermore, they tend to exhibit severe word- and sentence-level repetitions, as the model cannot keep track of what has already been synthesized.

In order to handle those problems, *See et al.*[35] proposed the pointer-generator network, as well as a coverage mechanism to keep track of what has already been summarized. Their architecture also deals with out-of-vocabulary words and with the longer summaries used in the CNN/DailyMail dataset, outperforming the then current state-of-the-art by at least 2 ROUGE points.

Their pointer-generator network allows them to copy words from the input text via *pointing*, as was already done by *Nallapati et al.*. Where the two networks differ is in the way the pointing mechanism is handled, since the baseline for the two networks is the same and is represented in Figure 3.2.

Where *Nallapati et al.*[24] trained their pointer component to only activate for out-of-

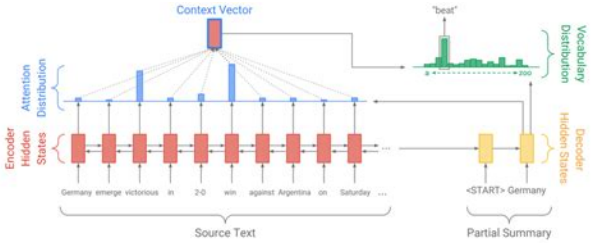

Figure 3.2: Baseline sequence-to-sequence model with attention[35].

vocabulary words and did not mix probability distributions from pointing and generation, *See et al.* allowed the pointing mechanism to activate for any word, and mixed the probability distributions given by both the generator and the copier to output a given word, as is represented in Figure 3.3. From the study they conducted on the abstractiveness of their model, they found out that their pointing mechanism made the abstractive system more reliable, as it copied factual details accurately more often than previously introduced models, in part because it can copy whole salient sentences, but that actually made it partly extractive. However, since the datasets used at that time (GigaWord, CNN/Daily-Mail) exhibited more extractiveness than abstractiveness, the BLEU and ROUGE scores obtained by the authors increased with respect to the previous state-of-the-art.

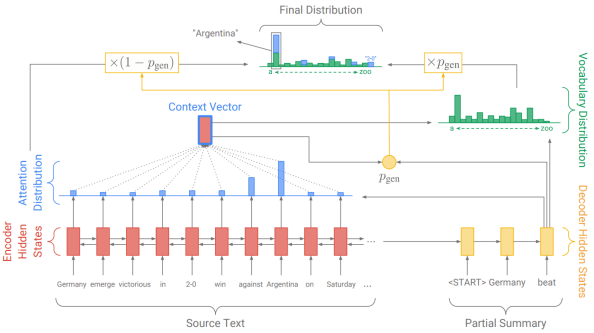

Figure 3.3: Pointer-generator model[35].

To deal with the second problem the authors observed, namely that of repetitions in the outputted summaries, they implemented a coverage mechanism, originating from Neural Machine Translation, stopping the models from outputting the same n-gram more than once.

NB : More mathematical information about the pointer-generator network is provided in section 6 in chapter 4, and compared with the one I used for CNN-based pointer generators as I felt it made more sense to have both formulations near one another.

6 Moving beyond RNN-based sequence-to-sequence models

Most of the early work in the field of abstractive neural summarization had been conducted using RNN-based sequence-to-sequence models, initially with simple recurrent

neural networks, and afterwards with more intricate models such as Long Short-Term Memory networks (LSTM) or Gated Recurrent Units (GRU). Those more complex networks deal with the issue of vanishing gradients, which typically make RNN-based networks harder to train. However, one problem that cannot be solved by these models is that of parallelization. Indeed, the fact that sequences must be processed sequentially incurs very long training times, especially when dealing with multi-sentences summary tasks such as the CNN/DailyMail dataset induces. This inability to parallelize RNN-based networks along the time dimension leads to training times that are challenging, especially for low-resources teams and organizations. It is therefore no surprise that all previously mentioned papers were released by teams that had the backing of big technological companies such as Google Brain, and/or large universities such as Stanford.

For this reason, this work, and several ones that preceded it, is concerned with CNN-based methods.

CNN-based neural networks have the ability to parallelize along the time dimension during training, alleviating the previously raised issues. Furthermore, the computational complexities of such convolutional models grow linearly with respect to the sentences length, and the amount of non-linearities that are applied to the input are fixed and independent from the length of the source, whereas RNN-based networks apply non-linearities linearly with the length of the input. The paths from output to input are also short, linear, and do not pass through gates or recurrent links, which allows the gradients to be propagated more efficiently than for RNN-based networks and bypass the problems of vanishing and exploding gradients. Finally, as mentioned in the Background section of this work, convolutional models inherently offer the benefits of bidirectional networks without any added complexity.

7 Introducing CNN-based abstractive models

The earliest CNN-based model that has been proposed for the task of neural machine translation, a task which suffers from the same aforementioned problem as abstractive summarization, was called *ByteNet* and was developed by *Kalchbrenner et al.*[22]. The model is a one-dimensional CNN composed of an encoder and a decoder, stacked on top of one another. It reached state-of-the-art performance on character-to-character translation of a benchmark dataset, outperforming RNN-based methods running in quadratic time.

The idea behind CNN-based methods is to use the various *kernels* present in each convolutional layer to extract features from the input text. If the first convolutional layer has a filter width of x, it is able to see concurrently x consecutive tokens. Each layer having hundreds of these filters, different kinds of features and information will be extracted by each one as they slide across the input sentence, looking at x tokens at a time to catch close dependencies. The output produced by passing through such a layer can then be fed to subsequent convolutional layers, allowing the network to catch word dependencies that are further apart, as the effective receptive fields increases linearly with the number of convolutional layers[1].

[1]If 6 subsequent convolutional layers each have a kernel width of 5, a single cell in the last layer would

7.1 Convolutional Sequence to Sequence learning[18]

Several further works continued exploring the use of CNN in neural machine translation, but it is *Gehring et al.*[18] that really took the framework and applied it to neural abstractive summarization as a side experiment in their neural machine translation-centered paper, in 2017.

Their approach is based entirely on Convolutional Neural Networks used in the context of a sequence-to-sequence model, with attention added to the decoder. As previously mentioned, the effective context size of such a network can be either increased or decreased by stacking multiple Convolutional layers on top of one another in the encoder. Furthermore, stacking such layers inherently creates hierarchical representations of the input sequence, as words that are closer together interact at lower layers while more distant words interact at higher layers. This enables the network to capture long-distance relationships better than a recurrent neural network could, as long-term dependencies are typically lost in RNN-based sequence-to-sequence networks, and they must tweak the vanilla networks to overcome that problem.

On top of convolutional layers, the authors also use Gated Linear Units (GLU), and residual connections, as well as attention at every layer of the decoder, all of which does not add significant overhead. As this architecture is the basis for this thesis, I will now present it in more details. The following sections are obviously heavily influenced by the original work of *Gehring et al.*[18], and the illustrations come from [13]. I will first introduce the relevant notations and concepts before illustrating the network.

Formulation

Let us first introduce the notations that will be used in this section.

- The input sequence of words $\mathbf{x} = (x_1, x_2, ..., x_m)$, embedded as vector $\mathbf{w} = (w_1, w_2, ..., w_m)$
- The embedding of the absolute positions of words in the input sequence $\mathbf{p} = (p_1, p_2, ..., p_m)$
- The input element representations $\mathbf{e} = \mathbf{w} + \mathbf{p}$
- The encoder state representations $\mathbf{z} = (z_1, z_2, ..., z_m)$ that are produced by the encoder after having been fed $\mathbf{x}$
- The output sequence $\mathbf{y} = (y_1, y_2, ..., y_n)$ generated from the state representations $\mathbf{z}$
- The decoder hidden states $\mathbf{h} = (h_1, h_2, ..., h_n)$
- The embedding of previously generated words outputted by the decoder $\mathbf{g} = (g_1, g_2, ..., g_n)$

With those notations in mind, we can start detailing the network. Both the encoder and the decoder share a convolutional block structure that is repeated and stacked. It computes intermediate states based on a fixed number of input elements. As multiple hidden states will be produced in both parts of the architecture, since these blocks are stacked on top of one another, we will denote :

depend on 25 input tokens.

- $\mathbf{h^l} = (h_1^l, ..., h_n^l)$ is the output of the l-th block in the decoder network.
- $\mathbf{z^l} = (z_1^l, ..., z_n^l)$ is the output of the l-th block in the encoder network.

Each one of these convolutional blocks contains a one-dimensional convolutional layer followed by a non-linearity in the form of a Gated Linear Unit[2] that allows the network to either focus on some parts of the input, or the whole input. The kernel size of each convolutional block will be set to 5, meaning that, in each layer, the network will be able to receive the information about 5 elements from the previous layer, and each layer will have 512 of those kernels.

Each kernel is represented as $W \in \mathbb{R}^{2d \times kd}$, $b_w \in \mathbb{R}^{2d}$ and takes as input $X \in \mathbb{R}^{k \times d}$, which is the concatenation of k input elements embedded in d dimensions, and maps X to a single element $Y \in \mathbb{R}^{2d}$.

Finally, residual connections are added in each convolutional block and take the form :

$$h_i^l = v\left(W^l \left[h_{i-k/2}^{l-1}, \ldots, h_{i+k/2}^{l-1}\right] + b_w^l\right) + h_i^{l-1}$$

One thing to keep in mind is that padding must added to both the encoder in order for the input length to match the convolutional layers; and the decoder, so that future information provided in the training summaries are not available to the decoder at a given time.

At the end of the decoder's convolutional blocks, the next word distribution is computed thanks to a softmax function, turning an $f-$dimensional vector into a vector respecting the constraints associated with a probability distribution.

Multi-step attention

On top of the attention inherent to the network that has previously been discussed (the receptive fields obtained through the kernels allow the network to have access to dependencies between both close and distant words of the input sequence), the authors introduced another explicit attention mechanism for each decoder layer. To do so, they combined the current decoder state of layer l h_i^l with an embedding of the previous target element g_i in the following way :

$$d_i^l = W_d^l h_i^l + b_d^l + g_i$$

The attention a_{ij}^l of state i and source element j at layer l is then derived from d_i in the following way :

$$a_{ij}^l = \frac{\exp\left(d_i^l \cdot z_j^u\right)}{\sum_{t=1}^m \exp\left(d_i^l \cdot z_t^u\right)}$$

[2]Taking $Y = [A\ B] \in \mathbb{R}^{2d}$, we have that the GLU operation is : $v([AB]) = A \otimes \sigma(B)$, with the sigmoid used to control which inputs from A are relevant.

The input c_i^l to the current decoder layer is then a weighted sum of the encoder outputs and the input embeddings e_j :

$$c_i^l = \sum_{j=1}^{m} a_{ij}^l \left(z_j^u + e_j\right)$$

This kind of attention is slightly different from the attention introduced by Bahdanau. Adding the embeddings in the previous formula provides point information about specific input elements. Furthermore, whereas Bahdanau's attention is single step, this one is a "multi-step" attention scheme since it is implemented at each convolutional layer.

Detailed model

Encoder

The CNN-based encoder will be fed the positions and embeddings of the source text, and will produce 2 context tokens per input token, to be fed to the attentional module in the decoder, as can be seen from the dimensionality of Y that has been previously presented. The two vectors of tokens thus produced correspond to the *conved* and *combined* vectors in Figure 3.4. In order to follow this description more easily, that figure shows the architecture under consideration on the task of Neural Machine Translation, but the model for both tasks does not differ at that stage.

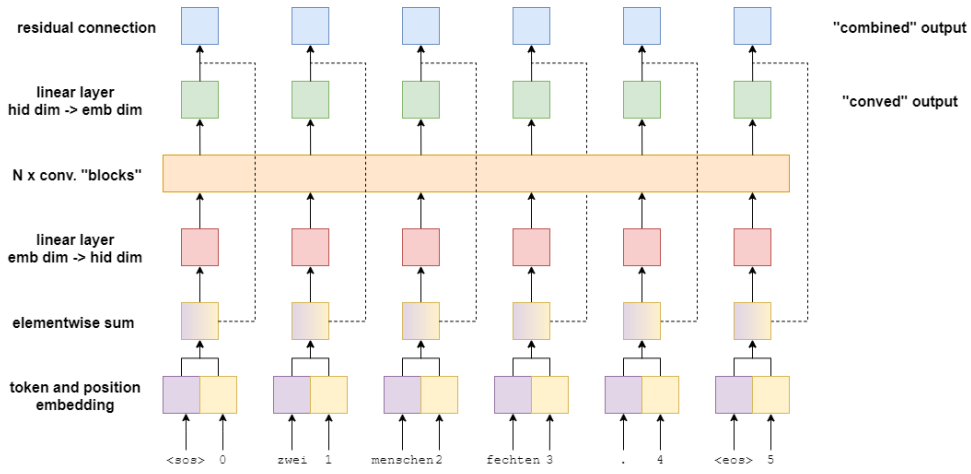

Figure 3.4: Architecture of the Encoder part of the Convolutional Seq2Seq.[13]

As said in the previous section, the input of the encoder part of the model is the summed combination of both the positions and tokens embeddings. Two embedding layers are therefore required at the input of the network. Adding the position is not something that must be done in regular RNN-based models, but is necessary in CNN-based ones since we do not process sentences sequentially, but in parallel. The vector resulting from the summation therefore encodes both the position of each token in the input sentence and their embedding at the same time.

This vector then passes through a linear (dense) layer that outputs a vector of the expected dimensionality to proceed in the convolutional blocks.

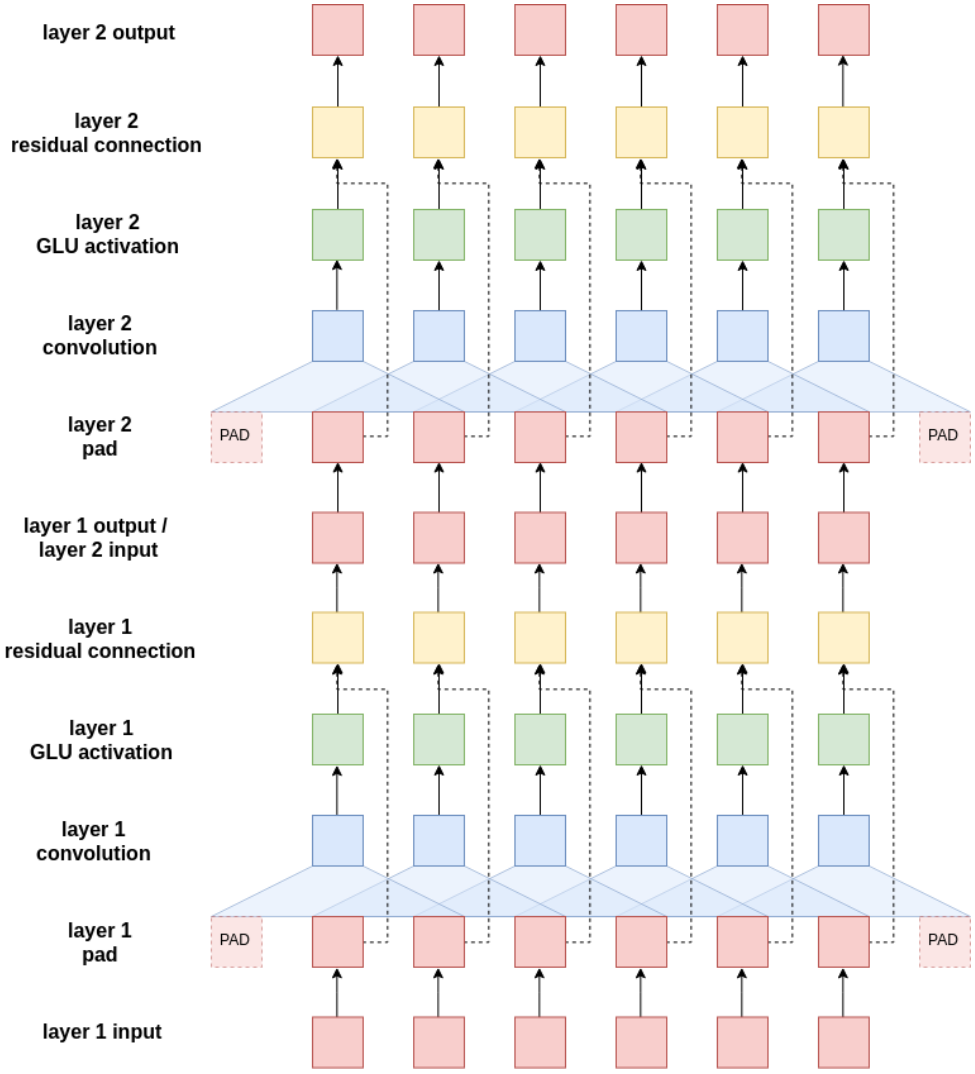

Figure 3.5: Architecture of the Encoder's Convolutional Blocks from the Convolutional Seq2Seq[13].

These blocks (Figure 3.5 displays a sequence of two such blocks), are first composed of a padding operation to keep a consistent vector length throughout the network. The input then passes through the convolutional layer, which outputs a vector twice as long as the input since it then passes through the GLU layer, which will halve the length of the input it receives. In effect, the size of the vector after that GLU activation layer is the same as the size of the input vector, and it can thus be summed with the input through a residual connection.

This output can then be fed into a similar block, since the dimensions stay the same. In practice, for the task of abstractive summarization, both the encoder and the decoder have 6 to 8 of these blocks in cascade.

The output vector of these stacked blocks then passes through another linear layer that reshapes it to the embedding dimension. That final vector, which is called the "*conved*" output, is finally element-wise summed with the embedding vector through a residual

connection, producing the so-called "*combined*" output.

Decoder

A schematic of the decoder architecture is provided at Figure 3.6 for ease of following the description.

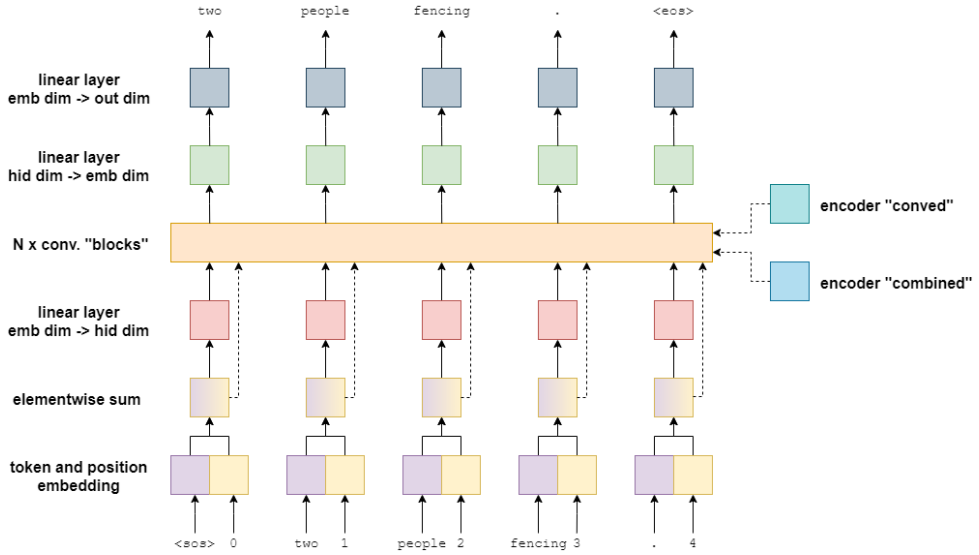

Figure 3.6: Architecture of the Decoder part of the Convolutional Seq2Seq[13].

The main input of the decoder is either the whole sentence to be predicted during training, or the output generated by itself up to a given point at test time (and in practice), as is the case for RNN-based Sequence-to-sequence models too. Where this decoder differs from the aforementioned one, however, is in the parallelized way the output is produced during training, as opposed to the sequential way that RNN-based methods use.

In this model, all output tokens are generated in parallel, this is problematic with regards to the possibility of using reinforcement learning where we may want the network to generate a sentence by itself without feeding it the expected previous tokens at each step. Optimizing the network to generate a complete sentence that makes sense by itself is not possible without losing the benefits of parallelization. However, one possible reinforcement learning scheme that can be applied to this framework is given in the next section, and is explored in the experiments.

The architecture is similar to that of the Encoder, and as such we will not repeat already mentioned information, only the parts where they differ. Among the differences, we have no residual connections after the convolutional blocks between the outputs and the embeddings. Furthermore, the convolutional blocks take as input both outputs of the Encoder, as can be seen in Figure 3.7.

As the whole expected sentence must be fed to the decoder and is processed in parallel, the padding must be done in such a way that the filter does not have access to future ground-truth information at any given point. The padding is therefore only done at the beginning of the sentences, and the last token of the expected sentence, which is an "*end-of-sentence*" token, is omitted.

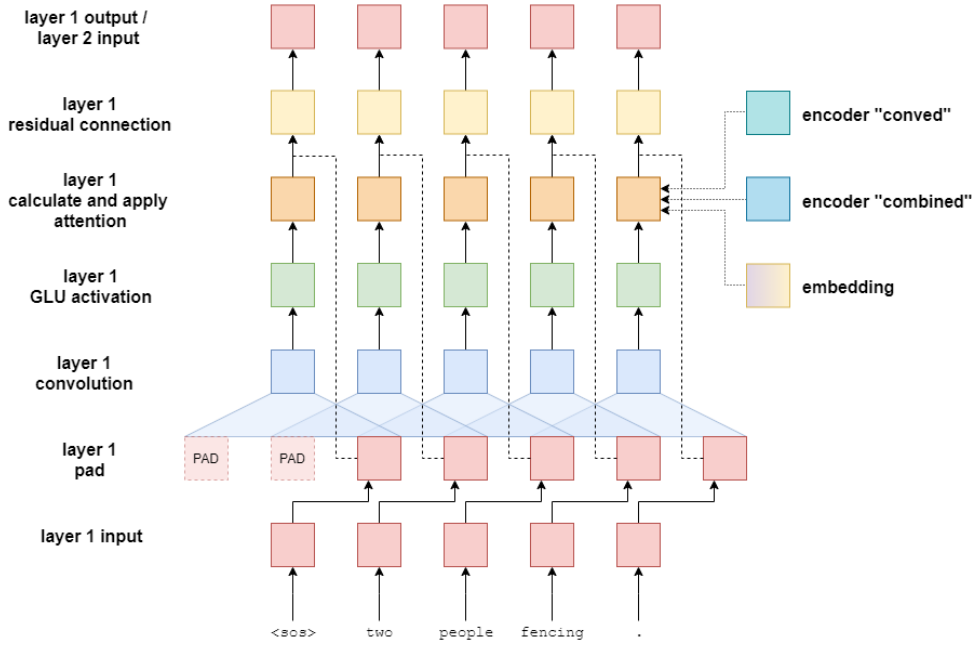

Figure 3.7: Architecture of the Decoder's Convolutional Blocks from the Convolutional Seq2Seq[13].

The second part that differs from the encoder's convolutional blocks is the computation and appliance of the attention mechanism after the GLU activation layer. This is where the encoder's outputted contexts are used in the network, as well as the decoder's input embeddings[3], through the attention formula outlined previously.

7.2 Proposed improvements for Convolutional Sequence-to-Sequence learning

After the release of ConvSeq2Seq, presented just above, few papers tackled the problem of convolutional-based abstractive summarization methods, but some works have nonetheless been published. We are now going to take a look at these articles, which will partly make the focus of this thesis.

A Reinforced Topic-Aware Convolutional Sequence-to-Sequence Model for Abstractive Text Summarization[42]

The first paper we will take a look at was published in 2018 by *Wang et al.*[42] and aims at incorporating context information in the convolutional sequence-to-sequence model. Furthermore, it tackles the problem of the inconsistency of training and testing measurements through reinforcement learning. Indeed, in typical sequence-to-sequence training, the model is trained by minimizing a cross-entropy loss between the ground-truth tokens and the output of the model. However, in order to evaluate the performance of the model, it is the BLEU or ROUGE score that is used, as it makes more sense in the context of

[3]It must be noted that the Figure is not totally accurate for the sake of clarity. Indeed, each token gets access to its embedding, not just the last one.

natural language processing. The reason one of those measures is not used during training is that they are not differentiable, which means they cannot be used as the loss in a neural network. The way this is dealt with in RNN-based abstractive summarization models is through reinforcement learning and curriculum learning. However, such a method could not transposed to CNN-based models since, as previously mentioned, the model cannot use its own prediction as input during the training, as the whole summary is computed in parallel and going to a sequential decoding would go against the very purpose of using CNN-based networks in the first place[4].

What the authors of this paper proposed to alleviate that issue was to use Self-Critical Sequence Training (SCST) for optimization. This new kind of training also helps with the aforementioned problem of *exposure bias*[31], which occurs because text generation models are typically trained to predict the next word in a sequence given the expected previous words but, at test time, the model must generate the entire sequence from scratch and does not have access to ground-truth tokens. This difference between training and testing usually deteriorates the quality of the generated summaries, as errors and inaccuracies tend to accumulate as words are decoded and the mechanism makes use of previously outputted and not necessarily correct words to generate the rest of the sentence.

Self-Critical Sequence Training, introduced by *Rennir et al.* [32] in 2017, differs from other reinforcement learning methods in that the network being trained uses its own output to normalize the rewards. This avoids the need for a second model to estimate the reward signal (as is done in Actor-Critic algorithms), and "harmonizes the model with respect to its test-time inference procedure", enabling the benefits mentioned above.

The way this training procedure is used by the authors of [42] is by generating 2 sentences given any one input sentence $\mathbf{x}$. The first one, denoted $\hat{y}$, is obtained by greedily selecting the most probable words for each token while the second one, denoted y^s, is generated by sampling the vocabulary distribution generated by the model. The ROUGE score for both sentences is then computed to act as the reward signal, and the reinforcement loss that will have to be minimized is the following :

$$L_{\text{rl}} = -\left(r\left(y^s\right) - r(\hat{y})\right) \log p_\theta\left(y^s\right)$$

It is easier to see why the exposure bias is reduced in this training regime. Indeed, the baseline against which the model is compared is its own prediction at test time, in the form of $r(\hat{y})$. The model is thus exposed to its own distribution and the algorithm forces it to produce an output with a large ROUGE score, which reduces the exposure bias.

In practice, this loss is slowly combined with the cross-entropy loss as training progresses and as convergence nears.

Don't Give Me the Details, Just the Summary! Topic-Aware Convolutional Neural Networks for Extreme Summarization[25]

In this work, *Narayan et al.* introduced the task of *extreme summarization*, which aims at summarizing a document into a single sentence in an abstractive manner. The ques-

[4] *i.e.* having a faster training time through parallelization.

tion that should be answered by that sentence is *"What is this article about ?"* Their approach is a purely convolutional-based one, and their model was trained on a dataset of BBC articles they created and called the *XSum* dataset. As was done for the previously mentioned paper, the model is conditioned on the article's topics.

This paper is interesting because the goal it pursues is entirely abstractive summarization. Previous models, trained on Gigaword, DUC or CNN/DailyMail, were mainly extractive methods which showcased a bit of abstractiveness. That is because these datasets favored such results inherently. *Extreme summarization*, on the other hand, is not amenable to extractive summarization and must be approached in an abstractive way, since the summaries from XSUM are widely different from the titles of an article for example, as can be seen in Figure 3.8. The goal is to generate a human-like summary containing paraphrases, reformulations, synthesis,...

SUMMARY: *A man and a child have been killed after a light aircraft made an emergency landing on a beach in Portugal.*

DOCUMENT: Authorities said the incident took place on Sao Joao beach in Caparica, south-west of Lisbon.
The National Maritime Authority said a middle-aged man and a young girl died after they were unable to avoid the plane.
[*6 sentences with 139 words are abbreviated from here.*]
Other reports said the victims had been sunbathing when the plane made its emergency landing.
[*Another 4 sentences with 67 words are abbreviated from here.*]
Video footage from the scene carried by local broadcasters showed a small recreational plane parked on the sand, apparently intact and surrounded by beachgoers and emergency workers.
[*Last 2 sentences with 19 words are abbreviated.*]

Figure 3.8: An abridged example from our extreme summarization dataset showing the document and its oneline summary. Document content present in the summary is color-coded.[25]

Since the objective of the authors was to summarize "long" documents, they argued that Convolutional Neural Network-based models are more suited when compared to RNN-based ones, since they can capture longer-term dependencies more effectively, as we have already mentioned. Document-level summarization is therefore possible, as opposed to paragraphs-level summarization usually enabled by RNN-based networks.

The way their encoder incorporates topical information is by associating each input token with a topic vector that determines whether it is representative of the document's content. The decoder, on the other hand, conditions each predicted token on the document topic vector. Aside from that addition, the architecture is the same as for *Gehring et al.*'s convolutional sequence-to-sequence model.

Word's and document's topical information can be obtained by any topic model, but the authors used Latent Dirichlet Allocation for the task.

Controllable Abstractive Summarization[16]

In this article, *Fan et al.* tackle the problem of including user preference such as summary length or style into the summarization process. Those are not inputs we are really interested in for this work, but the authors briefly mention some ideas and concepts that are interesting. Among those, one can name the use of Byte-Pair Encoding to deal with rare and out-of-vocabulary words, as opposed to the traditional pointer-generator model; or the use of intra-attention on top of a Bahdanau's like attention traditionally used in convolutional sequence-to-sequence models. Those are elements we will explore in the experiments, and we will therefore introduce them in their respective sections.

7.3 Other improvements

On top of the improvements that are highlighted in the previous section, some that have been tried on RNN-based models have not been applied to CNN-based models, to the best of my knowledge.

Among those, we can cite mechanisms such as coverage and pointing, that were presented earlier, and which we will implement in this work.

Something else that will not be explored in this work due to time/computational power requirements is the use of a generative adversarial network (GAN) framework in which the generator would be a CNN-based sequence-to-sequence model. Such a framework has only been tested on LSTM-based networks[23] in 2018, where the authors showed that, qualitatively, their model generated more abstractive, readable and diverse summaries than their current counterparts.

Chapter 4

Architecture and experiments

The basic architecture used in this thesis is that of *Gehring et al.*[18], since it is the one that most works have built upon over the years, and is therefore the baseline I will use to assess the effectiveness of what will be implemented in the following sections. It has been implemented from the original paper, with at times the help of codes from this repository[14] and from this repository[37] implementing various sequence-to-sequence models with pytorch. That ended up being the framework I used, and the basic model I use, in its most vanilla form, is heavily inspired by that latter repository. The Byte-Pair encoding script I used in section 6 to produce the abridged vocabulary and to preprocess accordingly the datasets comes from this repository[9]. The pointer-generator architecture used in section 6 is entirely my own and adapted from the original RNN-based paper[19] and one of its implementations[30]. All the results are solely mine, as are codes written with regards to the convolutional pointer-generator, reinforcement learning, embeddings and intra-attention.

Concerning the dataset used to train and assess the performance of the models, the XSUM dataset will be the main one used, supplemented with the CNN/DailyMail dataset. The reason behind this choice is that the XSUM dataset has been specifically tailored for abstractive summarization and has been released more recently than its counterpart, making it less explored in the literature. Models trained using the CNN/DailyMail dataset are also typically much more extractive by design, as will be shown in sections 4 and 5.

As a reminder of the goals of these experiments, we will first start by assessing the performance of one of our convolutional model against RNN-based networks, both in terms of training time and ROUGE scores. Afterwards, we will take a look at the importance of the number of convolutional blocks in the model on both a more extractive dataset (CNN/DailyMail), and a more abstractive one (XSUM). This will be followed by an extensive study of word embeddings and the usefulness of pretrained ones, before introducing a pointer-generator model and comparing it to a scheme serving a similar purpose (BPE). Finally, we will conclude with an experiment about reinforcement learning and its use in the framework of convolutional sequence-to-sequence training, before mentioning aborted experiments and possible future works.

Before that, however, I will mention my early work that did not made its way in this thesis, some problems encountered and the training methodology I used.

1 Early work and problems encountered

Before devising the research questions this work would be based on, and before realizing that training a regular RNN-based sequence-to-sequence model for the task of abstractive summarization on a dataset large and complex enough to get interesting and applicable results would require very large amount of computational time and power, I familiarized myself with the task by implementing simple RNN-based models on the Amazon Review dataset[39], a small dataset that summarizes Amazon products reviews into the title of the review. Although such an approach could not yield good results, it provided me with some experience with regards to the task, its challenges and the overall framework of sequence-to-sequence and pointer-generator models. Due to the lack of relevance of these experiments in the final version of this work, I will not mention it further, continuing this section by explaining the various challenges that appeared when working with the convolutional models directly.

The problems I encountered during the development of the code were mainly the incompleteness of some papers, making it quite difficult to pinpoint the exact architecture and training procedure the authors used to reach their results. As training a simple model on a reasonably sized dataset takes between 3 and 4 days, the process of fine-tuning the parameters, training process and architectures took some time and could not really be parallelized.

The hardest parameter to tune was beyond a doubt the learning rate. Selecting it too high (0.25) quickly led some models to output exploding prediction values, leading to *Not-a-Number* losses that could not be optimized. Selecting it too small, on the other hand, led to very slow improvements in the training and evaluation of the model. The game was therefore to find both the right starting point and decreasing method so that the results would improve over training and not stall/violently decrease in performance.

On top of being quite sensitive to the learning rate, the models were also very sensitive to the optimization technique. Although Adam or Adagrad were used in several papers, I found that it led to very poor results and a very hard fine-tuning of the parameters. The selection of an appropriate learning rate was even more critical in that case.

Finally, I used the training procedure that was used by *Narayan et al.*[25] for all datasets and models, which gave the best results and allowed to start training with a large learning rate. That enabled me to speed up training, as the number of epochs required to reach convergence decreased. The training procedure in question consists in starting with a learning rate value of 0.1 and optimize using the Stochastic Gradient Descent algorithm with Nesterov Momentum (with a momentum of 0.99). Using such a momentum scheme allows to accelerate training progress, avoids the model getting stuck in a flat loss area and slows down the search when approaching a local minimum. The learning rate is also divided by half when the validation loss does not decrease from epoch to epoch, until reaching a lower bound of $1e-4$. Those parameters were tested in my own experiments and I found out that selecting a bigger initial learning rate (0.25) did not speed up the

convergence, and lowering the lower bound had the effect of slowing the training without inducing any benefit.

Taking a look at a typical training and plotting the losses, the models initially showcased losses that were similar in form to the one plotted in Appendix A in Figure A.1. Although it is not clear from the magnitudes displayed in the graph, the training loss was of 57547 with a validation loss of 1330 at the end of the first epoch before shooting to losses in the order of $1e6$ with an initial learning rate of 0.001. At that point, the learning rate was cut by a factor 10 and the model started producing decreasing losses, but still ended the training with a validation loss of about 75 and not much decrease from one epoch to the next. That loss was too high for the model to produce anything remotely correct, and the learning rate was then too small to see any significant progress in a respectable training time. When we compare those losses to the ones that were obtained using SGD with Nesterov Momentum (all else being equal except for the initial learning rate being even bigger for SGD), however, we see directly that the latter is poised to produce better results. Those losses are presented in Figure A.2. As we can see, even though the learning rate is much greater than for the Adam optimizer, the losses immediately start decreasing and continue to do so until about 35 epochs, where they begin to converge. In light of such graphs and corresponding results, this is the training method that will be used in all experiments.

Another problem I faced with the Covid situation was that I did not receive the full GPU access before mid-July 2021, making it hard to test and train the models, and reducing the amount and breadth of experiments I would be able to conduct.

Finally, one problem I encountered while sending jobs to IBM Cloud is that some of them were marked as running, and were actually executed for some time, before stalling. They were still considered as being executed, but seemingly stopped making any progress. When I realized that issue, I relaunched some of those scripts, but that problem seemed more prevalent when a lot of jobs were running simultaneously, and therefore some of those relaunched jobs were halted again. That problem with batches of experiments also limited the number of scripts I could run at any one time. The only way to detect the halting problem, however, was to wait for the expected number of days to check if the execution was over and, if not, sometimes the log indicated that a problem with the server had occurred. For example, one job I monitored ran for 2 days before entering that state of non-execution. Most of my script ran, and I have results for all experiments, especially for the XSUM dataset, which is the main one under consideration, but I still lacked results from some scripts that should have complemented the results or answered some of my hypotheses. Whenever applicable, I will reference such scripts, my hypotheses regarding them and the results I expected.

2 Training and Evaluation methodology

For all experiments, the dataset in use was divided in 3 parts : A training set consisting of two-thirds of the examples, a validation set consisting of a fifth of the examples, and a test set containing the remaining data samples. The models were trained on the training set, evaluated at each epoch on the validation set to decide whether or not the learning

rate should be decreased and the model saved as a checkpoint, and finally assessed on the test set. For the two first kinds of sets, the models were trained and evaluated using the cross-entropy loss and, for the test set, they were evaluated using either the BLEU or ROUGE scores. As a reminder, the BLEU score assesses the accuracy of the predictions outputted by the model by comparing various n-grams to the target summaries. One drawback of that method is that if no n-gram of a given order are found in the predicted summary, the whole score will be 0, even if the summary makes sense. Furthermore, this score only assesses the accuracy of the summary when compared with a predefined ground truth. If the model is truly abstractive and can produce novel sentences, it is very well possible that it will obtain a low BLEU score, as paraphrasing a summary is considered as erroneous, even if the whole sense of the sentence is there. Those problems also occur for the ROUGE score, which is just an offshoot of the BLEU score but works on the same principles. To summarize the point, evaluating an abstractive model automatically and in a meaningful way when compared with our human intuition is much more difficult than to create an abstractive model. As a matter of fact, it may even be impossible.

One possibility that had been envisioned by a research group in May 2021 was to use Generative Adversarial Networks[46] for the task. However, this does not alleviate the whole problem, as the summaries are still evaluated using the ROUGE score. One idea could be to use the accuracy of the discriminator as the scoring system, but it comes with its own problems and biases. *In fine*, there is currently no automatic method that can determine the validity of a summary with the same acumen as a human being could, and therefore it can be of interest to personally review some summaries in some cases, as will be done at some point in this thesis.

3 Comparison with RNN-based sequence-to-sequence models

The first experiment we will mention is obviously the one that sparked the desire to tackle the task of abstractive summarization with a convolutional sequence-to-sequence model instead of a RNN-based one.

I will compare the various models through two considerations : training time and performance. Obviously, as I could not train the RNN-based models to convergence to obtain similar results, I will, when comparing ROUGE scores, take results from the literature for those models.

3.1 Time comparison

Let us start this time comparison section by stating that training a simple RNN-based network for one epoch on the XSUM dataset took 11 hours, and this does not take into account the time needed to assess the evaluation loss of the model. When we combine those two steps, we reach a training time of 16 hours per epoch. In other words, if we wanted to train this network for as many epochs as the CNN-based networks, it would require the model to run for close to 27 days, before even being able to assess the performance of the model using $BLEU$ or $ROUGE$! And this training took part on the XSUM dataset,

which is smaller in practice than the CNN/DailyMail dataset.

The reasons why this training is so slow is because of the inability for recurrent networks to parallelize through the time dimension, as previously mentioned. Furthermore, since the computations of the gradients in such networks are more complicated and involve tensors having to be kept in memory until the whole sentence has been outputted, the memory requirements per batch processing are more important than they are for CNN-based models, inducing the requirement for smaller batch sizes, which further slows the training.

The training and inference times for the CNN-based model, on the other hand, can be seen in Table 4.2. As expected, they are much more tractable for a Master's Thesis, and for smaller organizations.

3.2 Performance comparison

Now, let us see how the results we obtain with our convolutional sequence-to-sequence model compare to those obtained with regular RNN-based sequence-to-sequence networks. The results for the RNN-based networks come from the paper *Don't Give Me the Details, Just the Summary! Topic-Aware Convolutional Neural Networks for Extreme Summarization*[25], which introduced the XSUM dataset, and are therefore related to that dataset. Results are given in Table 4.1, and are measured using the $F1$ ROUGE score. The convolutional model that is provided is one I trained myself and that has 6 convolutional blocks in both the encoder and the decoder.

Method	**R1**	**R2**	**RL**
RNN Seq2Seq	28.42	8.77	22.48
RNN-Pt-Gen	29.70	9.21	23.24
RNN-Pt-Gen + coverage	28.10	8.02	21.72
ConvS2S with 6 layers	24.89	6.43	20.34

Table 4.1: Performances of various models evaluated on the XSUM dataset

Although the convolutional model presented does not reach the performances of the other networks, it must be noted that they are close, that these results were obtained using a greedy search instead of the beam search results of the RNN methods reported, and that it was not trained on the whole dataset, but rather on $\frac{4}{5}^{th}$ of it and not until full convergence. In light of these considerations, using a convolutional sequence-to-sequence model instead of a RNN-based one makes sense, especially when considering the time required to train and fine-tune RNN-based networks.

As is also applicable to this whole thesis, the results could be made much better by searching better hyperparameters, using a bigger (full) dataset, more complex models (for example fully training a 8-layered one) and more training epochs, but this would require more time and resources than I had available. My results are still valid since they compare models trained in a similar fashion, but I believe that they could all be improved significantly by doing what I just outlined. For example, the vanilla convolutional

sequence-to-sequence model trained by *Narayan et al.*[25] outperformed all the RNN-based baselines in Table 4.1 by several ROUGE points whereas mine underperformed those baselines.

4 Influence of the number of convolutional blocks over the quality of the summaries

As has been hypothesized in various aforementioned works, an increasing number of convolutional layers, in the encoder at least, should enable the model to capture more long-distanced relationships between input tokens, and therefore to output more meaningful and salient summaries. The first question we are interested in answering is therefore that of the tradeoff between the expected increase in performance enabled by more convolutional blocks in the model and the increase in training time and spatial complexity of the model, as it would require linearly more parameters. My hypothesis is that the performance gains are appreciable for larger datasets, where long-distanced dependencies play a major role and are not captured by a few convolutional layers, but are not worth the increased training time (induced by more parameters to train at each epochs, more operations to be conducted, and more epochs to reach convergence) in the case of smaller datasets.

To test this hypothesis, I will run the baseline model on the *CNN/DailyMail* and *XSum* datasets. We will use 4, 6, 8 and 10 convolutional layers in both the encoder and the decoder and measure the performance of the models using the BLEU metric on a testing dataset unseen during training. The whole training methodology follows that provided in *Gehring et al.*'s paper[18] and already mentioned in the previous sections. That comparison concerning the number of convolutional blocks is also extended in the next section, which tackles the concept of pretrained embeddings.

4.1 Time comparison

First of all, let us compare the change in training time when the amount of convolutional blocks is changed, all other things being equal. All the reported times were obtained by training the models on IBM Cloud with 1 virtual GPU and averaged over all training epochs in the case of the training time. The inference time, on the other hand, corresponds to the total time required to assess the performance of the model on the whole test part of the dataset. The times reported here correspond to a greedy generation, as opposed to a beam search. That choice was made because performances were comparable whatever the evaluation method, but the greedy decoding took 10 times less time to obtain the results than the beam search method. For all methods, the results were much better using beam search, and such results will be showcased later in this work, but given the amount of models trained and the time needed to obtain beam search results, performances are compared using the greedy decoding in the rest of this work too, unless explicitly stated otherwise.

As we can see in the table, and as is expected, the time complexity grows linearly with the number of layers.

Dataset	Number of layers	Training time per epoch	Inference time
XSUM	6	$137'$	$1h09'$
	8	$173'$	$1h14'$
	10	$216'$	$1h23'$
CNN/DailyMail	6	$141'$	$3h57'$
	8	$179'$	$4h$
	10	$220'$	$4h12'$

Table 4.2: Times comparison as a function of the number of convolutional blocks in the encoder and decoder.

The training time is also similar for the two datasets, all other things being equal. That is to be expected, since the models are the same and the batches are of the same dimensions. The input text is cut to 400 tokens since previous papers[19][25] wrote that doing so did not deteriorate the performance and even increased it while slightly reducing the training time and model complexity. Output tokens are also bounded to a length of 80 tokens, as is typically done in the literature. Doing so does not impair the performance of our models, since they typically never output above 80 tokens when the length is not limited, at least in our experiments. Therefore, the dimension of the batches for the two datasets are the same, and thus the models take the same time per epoch to train, as we set as equal the sizes of the training sets in both cases.

Another point that needs to be taken into account, however, is that the larger the model, the more training epochs it would need to converge and stabilize, which adds to the training time. This must be taken into account, as we will see in the next subsection.

Concerning the inference times, the CNN/DailyMail dataset requires more time than its XSUM counterpart because it is larger and therefore has more samples remaining after having set the training and evaluation sizes of both dataset as equal. However, since the BLEU scores are normalized, having more samples to test on does not induce much change in the scores when we work with tens of thousands of samples already.

4.2 Evaluation of the models

Concerning the performances, the networks have been trained for 40 epochs using the training procedure described in section 2. Let us first evaluate them on the XSUM dataset, which requires a higher level of abstractivity than the CNN/DailyMail one.

XSum dataset results

In order to evaluate the models, I first used the BLEU score as provided by `torchtext`[1]. I will evaluate the predictions of the models using n-grams going from 1 to 4, all equally weighted in the BLEU score.

[1]See `https://pytorch.org/text/stable/data_metrics.html`

With the output length limited to 80 tokens, the scores are given in Table 4.3.

Number of layers	BLEU-1	BLEU-2	BLEU-3	BLEU-4
4	21.66	10.77	5.88	3.53
6	22.89	11.66	6.55	4.01
8	23.35	12.07	6.83	4.29
10	22.59	11.44	6.30	3.75

Table 4.3: BLEU scores for models containing various numbers of convolutional blocks models, trained and tested on the XSUM dataset with an output size limited to 80 tokens.

We can see here that the results for the lower BLEU scores do not differ widely based on the number of convolutional blocks. However, as we increase the order of the BLEU score under consideration, the relative differences become greater. Whereas going from 4 to 6 layers only increased the BLEU-1 score by about 5.68%, the increase in the BLEU-4 score was of 13.6%! What this means is that the models with bigger layers are able to better capture long-distanced dependencies in the original text, particularly in this dataset, where sentences in the summaries (and therefore the n-grams of higher orders) contain words from all over the text and are not mainly extractive (recall Figure 3.8), as would be the case for the CNN/DailyMail dataset.

Another point that must be noted when looking at the results is that the relative increase in performance slows as we add more layers, and even becomes a decrease at some point. The reason for that is that all models were trained for the same number of epochs, and it is a well-known fact that models having more capacity (more parameters, in this case coming from more convolutional blocks) are harder to train and take more epochs to do so. I therefore believe that the smaller models, after the 40 training, were close to their local minimum in terms of convergence, while the larger models were not. Looking at the losses, however, it was clear that the improvement had slowed significantly for those larger models after about 35 epochs. Increasing the number of training epochs for those larger models too much would nonetheless improve the results displayed here, but one should be careful to avoid overfitting.

In light of these results, either 6 or 8 layers will be used in subsequent experiments.

CNN/DailyMail dataset results

With 80 tokens and pretrained embedding (GloVe 100 dimensions), the results for the CNN/DailyMail dataset are given in Table 4.4[2] :

As we can see, there does not seem to be much difference in this case when selecting either 6 or 8 layers, but the bigger the capacity of the network, the slightly better the results nonetheless. This may be explained by the fact that the CNN/DailyMail dataset encourages the models to be mainly extractive by design, and so the possibility to capture

[2]The 4 and 10 layers models were part of the models that ran forever on the cluster, and therefore I do not have their scores.

Number of layers	BLEU-1	BLEU-2	BLEU-3	BLEU-4
6	21.99	11.44	6.93	4.79
8	22.54	11.71	7.16	4.97

Table 4.4: BLEU scores for models containing various numbers of convolutional blocks models, trained and tested on the CNN/DailyMail dataset with an output size limited to 80 tokens.

longer-term dependencies that is enabled by more convolutional blocks in the network does not shine here.

Conclusion of the experiment

Is the added training cost required to train a bigger network worth these small improvements in the performances ? I would say that it is up to a certain point. Selecting 8 layers over 4 or 6 in this context makes sense, but choosing 10 layers does not, as it would require longer and more numerous epochs to reach convergence, for an increase in performance that may not justify it.

On another note, is it possible that the 8-layered model would require more training epochs to reach even better performance ? It should not be excluded and is certainly true, but the relative shapes of the losses for both the 6-layered and 8-layered networks are similar, suggesting a similar convergence rate at the end of training. Nonetheless, if one had as much time as he wanted to train the models, he should train them until the validation loss begins increasing from epoch to epoch, at which point he will know that it is starting to overfit.

5 Influence of the use of pretrained embeddings versus trained ones

The second experiment I have conducted is one measuring the influence of using a pretrained embedding layer when compared with the training of the layer concurrently with the rest of the network.

One downside of using a pretrained embedding is the size of the files that need to be stored in memory. As such files contain embeddings for hundred of thousands of tokens, they usually take several gigabytes and contain a lot of rare and unneeded embeddings. Apart from that small downside, however, there is no real reason not to use them. Nonetheless, several papers[18][25] exploring the use of convolutional sequence-to-sequence networks said that they trained the embeddings with the rest of the network, without necessarily providing a reason as to why they did so. Given the fact that there are really only upsides to using pretrained embeddings, that it speeds up the training as the network does not have to re-learn the weights associated with the embeddings and offers semantic and syntactic informations about the words, I was led to wonder why the use of pretrained embeddings was not the default option. Are there benefits to training the embeddings

yourself (aside from setting the desired dimension for the embeddings, as not all possibilities are offered by pretrained embeddings, although that reason does not make sense to me given the increase in performance we will see in the experiments) ? If anything, this should simply lead to less robust networks that are not able to transfer their skill (summarization) to a different domain (for example, from one dataset to another).

That assumption will be evaluated in this experiment by applying the summarization model trained on the XSUM dataset on the CNN/DailyMail dataset and vice-versa. One consideration to keep in mind with that approach, however, is that the XSUM dataset was designed for the models to produce more abstract summaries (the summaries have been made by humans with the purpose of creating a small summary in mind, contrarily to CNN/DailyMail where the summaries are typically more extractive). In a second time, I will also evaluate models trained on the CNN/DailyMail and XSUM datasets on the GigaWord dataset, which is related to the CNN/DailyMail due to its more extractive nature, but is also related to the XSUM dataset given the size of the summaries that are typically much shorter than those of CNN/DailyMail. That is not to say that the summaries produced by the model trained on XSUM will necessarily be out-of-touch with those expected by CNN/DailyMail, my expectation is that they will fare quite well, but a model trained to abstract (*i.e.* trained on XSUM) may not do well when evaluated using the BLEU or ROUGE score, as paraphrases, synonyms and words reordering are simply considered as wrong in the eyes of the metrics, even if they make perfect syntactical and semantic sense.

In a second time, I wondered if one embedding, between GloVe and Word2Vec, both trained on Wikipedia, was particularly better than the others. This question is easier to answer and will therefore be presented first.

5.1 Embeddings Comparison

First of all, let us evaluate trained and pretrained embeddings on the XSUM dataset alone. For this, the model we will use is the simple convolutional sequence-to-sequence network containing 8 convolutional blocks for both the encoder and the decoder, and trained for 40 epochs using the procedure described in section 2. The embeddings we will use are GloVe with 100 dimensions, GloVe with 300 dimensions and Word2Vec with 100 dimensions. The dimensionality of 100 will also be used for the trainable embeddings to keep consistency and obtain comparable results. Furthermore, the output will be limited to 80 tokens, as is usually done in the literature[3].

Comparing the various embeddings, we can see that the choice of pretrained embedding does not influence much the performance of the model. Word2Vec, however, shows slightly worse results than its GloVe counterpart, which is a result that is typically observed, as has been mentioned in section 6 of the background.

The greatest difference in performance, as expected, comes from the use of trainable embeddings, which is also confirmed by looking at Table 4.6. One thing to observe is that

[3]The results for CNN/DailyMail are not shown here since the Word2Vec version got halted on the GPU cluster, and simply comparing the use of pretrained versus trained embedding will be done just after.

Word embedding	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Pretrained (GloVe 100D)	23.35	12.07	6.83	4.29
Pretrained (GloVe 300D)	23.85	12.80	6.95	4.35
Pretrained (Word2Vec 100D)	21.56	10.79	5.98	3.63
Trained	18.16	8.01	3.96	2.20

Table 4.5: BLEU scores for various word embeddings trained and evaluated on the XSUM dataset

the trained embeddings fare slightly better on the CNN/DailyMail dataset than they do on the XSUM dataset. My guess is that the CNN/DailyMail is easier to train on, since it is a dataset that favors extractivity, while the XSUM dataset favors abstractivity and therefore requires more training as well as the ability for the network to predict related words (more easily enabled by pretrained embedddings) to reach similar performances.

The results obtained from the trainable embeddings models are not horrendous, and would certainly benefit greatly from more training, although the losses showed the same signs of convergence as the other models after the 40 epochs, but that is the point : why would you increase your training time to reach performances that will either be on par with what one would obtain by using pretrained embeddings, or better but more biased towards the training dataset. I do not have an answer to that question, but my guess is that, to get the best results and reach state-of-the-art scores on a given dataset, having word embeddings that are biased towards the dataset is preferable. However, as we are now going to see, this can come at the cost of the generalizability capacities of the models.

Word embedding	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Pretrained (GloVe 100D)	23.53	11.91	7.36	5.17
Trained	18.13	8.49	4.79	3.25

Table 4.6: BLEU scores for 8-layered models trained and evaluated on the CNN/DailyMail dataset

5.2 Assessing the performance of the models trained on CNN/-DailyMail on XSUM

In order to assess the performances of the models on a different dataset, we will consider as testing set the entirety of the second dataset. The testing time is therefore extended, but we obtain more representative results. This consideration is valid for both this subsection and the next. However, the ROUGE scores have been obtained by evaluating the models on 30000 examples instead of the whole testing set. The results are provided in Tables 4.7 and 4.8.

In this case, the scores of the 6-layered model with pretrained embeddings far outweighs those of its trained counterpart, as expected. What is unexpected, however, is that

Word embedding	Number of convolutional blocks	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Pretrained (GloVe)	**6**	10.40	3.67	1.51	0.74
Pretrained (GloVe)	**8**	11.50	4.24	1.76	0.92
Trained	**6**	6.32	1.89	0.53	0.14
Trained	**8**	10.09	3.80	1.45	0.59

Table 4.7: BLEU scores of models trained on CNN/DailyMail and evaluated on XSUM as a function of their embeddings.

Word embedding	Number of convolutional blocks	ROUGE-1	ROUGE-2	ROUGE-L
Pretrained	**6**	17.67	2.32	13.92
Pretrained	**8**	19.09	2.61	15.14
Trained	**6**	13.41	1.48	12.15
Trained	**8**	18.13	2.67	15.44

Table 4.8: ROUGE scores of models trained on CNN/DailyMail and evaluated on XSUM as a function of their embeddings.

the trained 8-layered model has BLEU scores that are only slightly below those of its pretrained counterpart, and ROUGE scores that are even slightly better of ROUGE-2. That result may be explained by the fact that the model with trainable embeddings outputted shorter summaries that were more on par with the ones expected by XSUM, while the model with the pretrained embeddings outputted slightly longer summaries, possibly resulting in that discrepancy in ROUGE-2 results. However, that is just an hypothesis.

5.3 Assessing the performance of the models trained on XSUM on CNN/DailyMail

Word embedding	Number of convolutional blocks	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Pretrained	**6**	8.65	3.09	1.13	0.46
Pretrained	**8**	11.10	3.76	1.61	0.59
Trained	**6**	6.67	1.98	0.52	0.16
Trained	**8**	7.64	2.35	0.70	0.24

Table 4.9: Performance of models trained on XSUM and evaluated on CNN/DailyMail as a function of their embeddings.

In this setting, as expected, the use of pretrained embeddings yields better results for both kinds of networks, as does the use of more convolutional blocks. That is the expected result. The overall performances are also slightly worse than for the previous experiment, which reinforces my idea that training a model on XSUM is harder to do than it is on CNN/DailyMail and results in abstractive models that are less well-suited for the more extractive CNN/DailyMail dataset.

Word embedding	Number of convolutional blocks	ROUGE-1	ROUGE-2	ROUGE-L
Pretrained	**6**	16.58	2.12	12.59
Pretrained	**8**	18.03	2.34	13.37
Trained	**6**	12.41	0.15	9.96
Trained	**8**	13.43	1.35	10.59

Table 4.10: ROUGE scores of models trained on XSUM and evaluated on CNN/DailyMail as a function of their embeddings.

5.4 Assessing the performance of the models on XSUM and CNN/DailyMail on Gigaword

For this experiment, the models have been tested as is on the GigaWord dataset. The BLEU and ROUGE scores are reported in Table 4.11 and 4.12.

Initial Dataset	Word embedding	Number of convolutional blocks	BLEU-1	BLEU-2	BLEU-3	BLEU-4
CNN/DailyMail	**Pretrained**	**6**	11.31	5.95	3.11	1.63
CNN/DailyMail	**Pretrained**	**8**	12.00	6.21	3.31	1.71
CNN/DailyMail	**Trained**	**6**	1.92	0.35	0.07	0.02
CNN/DailyMail	**Trained**	**8**	5.35	2.21	0.90	0.36
XSUM	**Pretrained**	**6**	8.41	3.15	1.12	0.36
XSUM	**Pretrained**	**8**	9.34	3.45	1.23	0.39
XSUM	**Trained**	**6**	3.56	0.64	0.12	0.00
XSUM	**Trained**	**8**	4.28	1.04	0.23	0.04

Table 4.11: BLEU scores of models trained on XSUM and CNN/DailyMail and evaluated on the Gigaword dataset.

Initial Dataset	Word embedding	Number of convolutional blocks	ROUGE-1	ROUGE-2	ROUGE-L
CNN/DailyMail	**Pretrained**	**6**	17.53	6.77	24.37
CNN/DailyMail	**Pretrained**	**8**	25.01	7.24	24.71
CNN/DailyMail	**Trained**	**6**	5.92	0.19	5.71
CNN/DailyMail	**Trained**	**8**	14.00	2.07	13.18
XSUM	**Pretrained**	**6**	13.94	1.83	12.71
XSUM	**Pretrained**	**8**	16.82	2.87	15.12
XSUM	**Trained**	**6**	5.24	0.16	4.91
XSUM	**Trained**	**8**	7.02	0.40	6.64

Table 4.12: ROUGE F1 scores of models trained on XSUM and CNN/DailyMail and evaluated on the Gigaword dataset.

What we can see is that models trained on CNN/DailyMail consistently fare better than the ones trained on XSUM. That is to be expected, since both GigaWord and CNN/DailyMail share some extractivity and therefore favor the same kind of extractive-oriented networks. Their target distribution are also similar, both consisting of article summaries obtained in a similar way. The use of pretrained embeddings also consistently produces better results. This is actually what we expected from an experiment measuring the

ability to generalize. Let us now have a look at the fine-tuning of these models on the GigaWord dataset.

Fine-tuning on GigaWord

In the final experiment for this section, I explored the results obtained by finetuning the convolutional models trained on XSUM and CNN/DailyMail on the GigaWord dataset. The convolutional models contain 6 convolutional blocks in both the encoder and the decoder, and either pretrained or trainable embeddings. The finetuning process was done by taking the models trained on either the XSUM or CNN/DailyMail dataset and training them for two additional epochs on a subset of GigaWord (30000 samples for the training phase and 10000 for the testing phase). The results are given in Table 4.13. The first thing we can say is that fine-tuning on GigaWord resulted in much better results than those previously observed. As expected for XSUM, which has an output distribution that is far from the GigaWord corpus, using a pretrained embedding led to better and more consistent results. For CNN/DailyMail, on the other hand, the results were slightly better when using a trained embedding. My guess as to why it happens is that both CNN/DailyMail and GigaWord share a common target distribution, as they are both mainly extractively built articles datasets. The bias that is learned on CNN/DailyMail therefore plays in favor of the trainable embedding here, and the more generalizable pretrained embedding yields slightly worse results, which corroborates the hypotheses emitted previously.

Concerning the ROUGE scores for XSUM, they follow the same trend. On top of having models that are more in line with the dictribution of GigaWord, that is also due to the fact that fine-tuning on the dataset had the effect of reducing the summary lengths that were generated, especially for CNN/DailyMail which typically outputs longer summaries, which improves the F1 scores of the sentences.

As we can see in Table 4.14, the summaries that are generated by the XSUM-based model are not necessarily worse than the ones generated by the CNN/DailyMail-based ones (Table 4.15), but were trained to learn to paraphrase and not so much extract, which we see here. This further shows that simply using the BLEU or ROUGE scores as is does not necessarily tell the whole story, particularly in the case of inter-dataset evaluations. However, there is no other alternative aside from reviewing some summaries.

Initial Dataset	Word embedding	Number of convolutional blocks	BLEU-1	BLEU-2	BLEU-3	BLEU-4
CNN/DailyMail	**Pretrained**	**6**	29.17	17.81	11.24	7.36
CNN/DailyMail	**Trained**	**6**	29.71	18.10	11.39	7.42
XSUM	**Pretrained**	**6**	21.51	12.07	7.05	4.25
XSUM	**Trained**	**6**	7.35	2.90	1.47	0.87

Table 4.13: BLEU scores of models finetuned and evaluated on the Gigaword dataset.

5.5 Conclusion of the experiments

As we can see from the results, not using a pretrained embedding seems to not make sense. The performances may not be disastrous when evaluated on the same dataset as the one

Target summary	Predicted summary
russia slams <unk> ukraine	russia condemns ukraine against <unk>
oil prices up in asia on hurricane fears	oil prices rise in asian trade
spanish official inflation slows sharply on oil price fall	spanish inflation fall in march

Table 4.14: Outputs generated with the XSUM-based 6-layered network with pretrained embeddings and fine-tuned on GigaWord

Target summary	Predicted summary
pakistan army commanders meet after us raids row	pakistan army commanders meet on <unk>
<unk> shares close percent lower	seoul shares close percent lower
mozambique ruling party readies to pick presidential candidate	mozambique ruling party to hold talks to choose presidential elections

Table 4.15: Outputs generated with the CNN/DailyMail-based 6-layered network with pretrained embeddings and fine-tuned on GigaWord

the models were trained on, but when evaluating in an inter-dataset way, the results back up our intuition that the models which do not have a pretrained embedding are less able to generalize their performance on other datasets than models equipped with pretrained embeddings. That is to be expected since the embeddings that are trained are especially tailored for the dataset at hand, and therefore I do not believe that a longer training for such models would result in vastly different performance, or different conclusions. Furthermore, as we can see in Table 4.5, the difference in performance for models having a pretrained embedding when compared to those which have not are much less than the ones we can observe in table 4.9. It thus reinforces my belief that both kinds of models reached the same level of convergence, due to the similar scores observed in Table 4.5, but that training our embeddings ourselves leads to poor inter-dataset performance, and training further the model not equipped with pretrained embeddings would only further the bias towards its own training dataset.

As it stands, I thus do not see any good reason not to use pretrained embeddings. On the contrary, training a model not equipped with a pretrained embedding for more epochs than I did would probably produce better results on the dataset on which it was trained on than using pretrained embeddings, but it is my guess that such models would fare even worse than the ones I showed here when evaluated on other datasets, since the weights of the embeddings would overfit the training dataset instead of being general and relying on the other layers to account for that generality. It is also my believe that this kind of inter-dataset evaluation should be performed more often, as the results presented in most scientific papers come from models trained on one dataset and evaluated on the same one. For the task of text summarization, it does not make sense, in my opinion, to not study the inter-dataset performance of your models and parameters, as real-life applications of

your methods will most probably never involve the exact data domain as the one the model was trained on.

However, if you know that the domain your models will be used on is very close to the one it will be trained on, as is the case for GigaWord and CNN/DailyMail, it may make sense to use trainable embeddings and fine-tuning the training. Otherwise, I would advise against using trainable embeddings and rather using pretrained ones.

6 Convolutional Pointer-Generator and Byte-Pair Encoding

In this experiment, let us measure the influence of the use of a pointer-generator mechanism in the Convolutional Sequence-to-Sequence model. To the best of my knowledge, this is something that has been mentioned before, but not implemented. *Fan et al.*[16] mentioned said that they used a combination of sub-word tokenization and weight sharing to deal with the problem that is solved by pointer-generator networks, namely the handling of out-of-vocabulary and rare words. The authors claimed that their solution answered the same problem, while at the same time being less complex and having fewer parameters.

I decided to test that hypothesis in this work, since they did not conduct it, and therefore to check whether or not the use of sub-word tokenization and weight sharing produced results that were on par with the use of a pointer-generator network, all else being equal. My best guess is that pointer-generators should produce better results, because they not only have the ability to copy rare words, but also to copy any word of the input directly, provided that it is important enough. A switching mechanism would provide the same benefits as the sub-word tokenization scheme, but a convolutional pointer-generator network should produce better results, and that is what we will test here.

For the sub-word tokenization, I used the Byte-Pair Encoding (BPE) scheme. That scheme has been adapted from the field of data compression to deal with the problem of rare words not being taken into account in regular word embeddings. To perform subword tokenization, BPE starts from the initial vocabulary at the character-level and merges together the most frequently occurring pairs into subwords. The process is repeated iteratively until the desired vocabulary size is obtained, making the scheme strike a balance between character- and word-level representations that enables it to tackle large corpora with a predetermined vocabulary size, and to handle out-of-vocabulary words effectively without using `unknown` tokens[9]. This scheme is therefore very cost-effective, since it only involves replacing both the input and output vocabularies by the ones generated by the procedure given above. Nothing more needs to be added to the network. Concerning the weight sharing, using pre-trained embeddings already takes care of that consideration. Using Byte-Pair Encoding is therefore very easy and cost-effective. Is it thus of any use to implement the more complex pointer-generator network ? I believe so, but let us find out after a presentation of the formulation I modified to implement the convolutional pointer-generator.

6.1 Architecture of the pointer-generator network

As pointer-generators have not been adapted to Convolutional Sequence-to-Sequence models, or at least the formulation of the architecture, dimensions and equations have not been written in any paper, I had to work from the original paper[35] and adapt the ideas and equations to the convolutional paradigm.

The basic idea is however the same for both architectures :

The vanilla sequence-to-sequence model produces as output the vocabulary distribution $P_{\mathbf{vocab}}$ over the static vocabulary that has been determined prior to the training. Therefore, that static vocabulary cannot take into account rare, or unseen words in a manner other than by producing an `unknown` token. The pointer part of the network, on the other hand, will enable it to copy words from the input to the output. How the network chooses between generating or pointing is through the generation probability p_{gen}.

In the RNN-based pointer-generator, p_{gen} is computed at each time step t from the context vector h_t^*, the decoder state s_t and the decoder input x_t and takes the form :

$$p_{\text{gen}} = \sigma\left(w_{h^*}^T h_t^* + w_s^T s_t + w_x^T x_t + b_{\text{ptr}}\right)$$

Where b_{ptr} corresponds to a bias parameter.

Finally leading to a vocabulary distribution :

$$P(w) = p_{\text{gen}}\, P_{\text{vocab}}\,(w) + (1 - p_{\text{gen}}\,) \sum_{i:w_i=w} a_i^t$$

Where a_i^t is the attention distribution at time step t. The use of attention allows the network to know which input tokens are important, and to copy them if they are relevant enough, whether they are `unknown` tokens or regular ones.

For the convolutional-based pointer-generator, we will keep the same form for p_{gen}, but several changes will have to be made, both to the convolutional sequence-to-sequence model we have used until now, and to the pointer-generator formulation. First of all, the computations for all timesteps will be done in parallel, which introduces an additional dimension to the problem[4] and a reshaping of the relevant tensors.

As all timesteps are computed in parallel in this setting, we do not need the t indices in the formulation of the probability p_{gen}. The sigmoid is still present, but the parameters h^*, s and x will have to be modified to fit our framework. From the decoder, we will define the s parameter as the internal state of the network directly following the convolutional blocks, acting as a decoder state since it contains the information about the previously decoded words. The parameter h, which corresponds to the encoding context in the RNN-based formulation, is here computed in a similar manner, as a sum of the input vector weighted by the respective attention of each word in the vector. Finally, the x parameter also corresponds to the input of the decoder, albeit of a different dimension suitable for this convolutional framework.

[4] A literal dimension, going from a 2-dimensional formulation to a 3-dimensional one.

To take into account `unknown` tokens, an extended source embedding is passed to the network, in which `unknown` tokens are replaced by numerical values outside the range permitted by the vocabulary length. The output predictions are then padded with 0 at those positions, and both that output distribution and the attentional part of the combined probability distribution are added together to produce the end distribution.

It can be noted that I also implemented the coverage mechanism in the first iteration of the model. However, the training was rendered exceedingly slow (two weeks and still no results), and I could not determine whether the problem came from my code or from the Cloud. For that reason, this mechanism, that was introduced in the same paper as the pointer-generator idea, was not implemented in the models which the results shown below are taken from.

6.2 Pointer-generator model training and performance

My implementation of the convolutional pointer-generator had to undergo several iterations before reaching satisfying results, as the only guidance I could rely on was the original paper and code, both designed for RNN-based sequence-to-sequence models. Although both kinds of resources were useful to grasp the ideas behind the framework, I still navigated in somewhat dark waters, not knowing whether or not the model would output results that were any good, since no previous work on which parameters would work best had been done. I also would have neither the time nor the computational power to conduct an extensive grid search to determine the optimal hyper-parameters. My training procedure was therefore chosen to be the same one as for the vanilla CNN-based sequence-to-sequence network I had used, the reasoning behind that decision being that both models were very similar, and "simply" computing additional probabilities and adding some layers should not induce the requirement for a vastly different training methodology. It is therefore entirely possible that this training methodology is not optimal for the network, but the results are still on par with what I expected.

The model used for the experiments was therefore trained in the same way as was presented previously, and contains 6 convolutional layers in both the encoder and the decoder, as well as the pretrained GloVe embedding with 100 dimensions.

After tinkering with my code to obtain a model that seemed promising, I first made it run for 20 epochs to assess its performance before letting it run for the 40 epochs I used for all models. That 40-epochs model got halted, but I have the result of the 30-epochs one, provided below. The results for all experiments are given in Table 4.16. It should be noted that the pointer-generator trained on CNN/DailyMail was one of the models that got typically stopped by the GPU cluster and, therefore, I do not have its results to show here. My hypothesis regarding the results that should have been produced is that the increase in relative performance when compared to its baseline should be greater than for XSUM, since the CNN/DailyMail dataset is more extractive, and so the ability for the network to copy any word from the input should have resulted in a better score.

What we can say from those results is that the use of a pointer-generator seems to make sense, as it outperforms both the baseline and the BPE-based networks. More discussion on these results, however, will be done in the remainder of this section.

Model	Dataset	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Pointer-Generator after 20 epochs	XSUM	21.39	10.41	5.53	3.23
Pointer-Generator after 30 epochs	XSUM	22.24	11.12	6.07	3.61
6-layered BPE after 40 epochs	XSUM	21.00	10.44	5.77	3.48
8-layered BPE after 40 epochs	XSUM	21.59	10.98	6.11	3.71
6-layered baseline after 30 epochs	XSUM	21.92	10.64	5.85	3.53

Table 4.16: BLEU scores of models implementing either a pointer-generator mechanism, or Byte-Pair Encoding

6.3 A deeper look into the generated summaries of the pointer-generator network

Something that can be done to assess whether or not the pointer-generator worked as expected is to look at how many `unknown` tokens it generated after processing the test set. As is expected, no `unknown` token was generated, despite being present in the input they had to process and in the outputs of the vanilla network. This fact comforts us in the idea that the network indeed copies relevant words from the input.

Let us therefore take a look at some of those samples in Table 4.17. Although they are far from perfect, we can see that they show the trend of being more legible than what we can obtain with the vanilla network, while also showing the copying capabilities of the network, for example for the word *dumfries* in the third summary, a rare word that appears only in the source article the summary is based on. Keep also in mind that the network was only trained for 30 epochs, since the 40 epochs network got halted multiple times in its training. Those results therefore would improve given a higher training time for the network, as we saw that the other networks started reaching some convergence at about 35 epochs.

Target summary	Predicted summary
reigning champions toulon scored late penalty to earn vital win against bath in the european champions cup	bath moved to the top of the pro table as they drew up to win over toulon in the pro champions cup
match report to follow	macclesfield extended their unbeaten run to three matches with victory over sutton united at moss park
worker has died in an accident on south of scotland wind farm site	man has died after falling into the wind farm in dumfries

Table 4.17: Summaries generated with the pointer-generator network on XSUM dataset

6.4 BPE results

Byte-Pair Encoding was tested on both the XSUM and CNN/DailyMail datasets, and the results were as expected, not necessarily impressive when looking at the BLEU scores,

which do not change wildly from the vanilla networks as can be seen in Table 4.16, but rather when looking at summaries taken at random. Although the possibility to output the `unknown` token was still available to the model, it only did so once out of 30000 summaries it generated, whereas it was commonplace for the original network (the `unknown` tokens appeared in 1700 generated summaries and appeared more than 3000 times, one example is given in Table 4.18. This means that it was easier for the vanilla model to get a larger BLEU score than for the BPE model, as outputting `unknown` tokens was a good guess). Furthermore, it also generated sub-tokens 1177 times out of the 11016 times they were present in the target summaries. Since `unknown` tokens are now very rare in the target summaries, the model has to output predictions that must be closer to the true sense of the targets in order to keep a BLEU score that is comparable to the vanilla network. Some summaries taken at random are given in Table 4.19, where we can see one example of the effectiveness of BPE. Note that this network was trained for the full 40 epochs.

Target summary	**Predicted summary**
celtic have confirmed striker colin kazim richards has left the club to join brazilian side <unk>	celtic have signed striker <unk> <unk> from turkish side <unk> on loan until the end of the season

Table 4.18: One typical example generated with the vanilla 6-layered network on XSUM dataset

Target summary	**Predicted summary**
reigning champions toulon scored late penalty to earn vital win against bath in the european champions cup	bath moved to the top of the pro table with victory over toulon in the first leg of their european champions cup quarter finals
man has been arrested by anti terror officers investigating bomb threat received by mosque	man has been arrested on suspicion of terrorism offences after an email messages was published by the islamic state
leyton orient have signed former millwall defender shane lowry on two year contract	leyton orient have signed former wolves midfielder james lowry on two year deal
an independent inquiry is being launched after baby girl died hours after being born at shropshire maternity unit	the family of year old girl who died after being discharged from hospital have said they are devastated by the care watchdog
exiled russian tycoon boris bere@@ zo@@ vsky has been found dead at his home outside london	the family of russian businessman alexander bere@@ zo@@ vsky ve has been found dead in the city of st petersburg
scheme to create more than jobs on business park next to london southend airport has been approved by government inspector	plans to build the first runway of the sea in southend have been approved by the council

Table 4.19: Summaries generated with BPE encoding on XSUM dataset

6.5 Conclusion of the experiment

As a conclusion, we can say that the use of a pointer-generator architecture makes more sense than the use of the Byte-Pair Encoding scheme in the context of abstractive summarization, since it not only allows to copy rare and out-of-vocabulary words from the input, but also to copy any word that is considered as important enough by the network, effectively enabling it to be more extractive, a feature that is desirable for datasets such as CNN/DailyMail and GigaWord, but also seems to produce better results despite having been trained for 10 epochs less than its counterpart in the case of the more abstractive XSUM dataset, as we saw in Table 4.16.

For simply dealing with out-of-vocabulary words, the use of BPE over the pointer-generator architecture makes sense, but given the small increase in training time induced by the use of a pointer-generator (about 30%) when compared with the increase in performance, I would advise selecting that architecture regardless, especially if my hypothesis that the gains in performance are even greater for the CNN/DailyMail dataset is true.

7 Use of Reinforcement Learning training

Finally, I conducted an experiment in which the use of reinforcement learning was studied and compared to the use of regular training using cross-entropy loss. As has been presented in the previous chapter, the use of reinforcement learning makes it possible to train the models by using the same non-differentiable metrics that is used to evaluate them, *i.e.* the BLEU or ROUGE score. Indeed, using the cross-entropy loss between the one-hot target distribution and the outputted distribution is not optimal, as the performance of the model is not tested against that measure.

The Reinforcement Learning algorithm I used for this is called Self-Critical Sequence Training (SCST), as has been presented in the previous chapter.

The first thing that is notable when training my models with that kind of training is that it takes longer than the regular training. The models are first trained with the cross-entropy loss, and only after several epochs is reinforcement learning brought in. That increased training time comes from the way the reinforced learning loss is computed. Indeed, the loss has this form :

$$L_{\text{rl}} = -\left(r\left(y^s\right) - r(\hat{y})\right) \log p_\theta\left(y^s\right)$$ [5]

Where r is the rouge score between its argument and the target summary, y^s is obtained by sampling the distribution outputted by the model, and $\hat{y}$ is the greedily selected output of the network. As we can see, not only do we have to compute twice the rouge score for each training example and at each epoch, we also have to create and sample a distribution,

[5] The absolute value that is needed to get a positive loss compatible with the cross-entropy loss is not included in this formulation, as it was not in the work of the authors, but is nonetheless included in my code.

as well as obtaining its log-probabilities. Those are all expensive operations that together slightly impair the speed of the training but that can be parallelized regardless.

Are the results worth of this increase in training time and complexity ?

7.1 Experiment

For this experiment, I took a model containing 6 convolutional blocks in both the encoder and the decoder and trained on the XSUM[6] dataset for 40 epochs using the regular cross-entropy loss. Afterwards, I continued the training through the use of the reinforcement scheme outlines earlier, making sure to decrease the learning rate to 0.0001, as is done in the literature, and setting the number of additional epochs to 5. I also first tried to add the reinforcement loss earlier in the training (after 20 regular training epochs), but the results were highly deteriorated when doing so, meaning that the model should be closer to convergence before introducing the reinforcement learning loss in the training.

The results can be seen in Table 4.20.

Use of RL	**Dataset**	**Comments**	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**
Yes	XSUM	lr set to 0.001	23.42	11.86	6.66	4.09
Yes	XSUM	lr set to 0.0001	22.97	11.71	6.59	4.04
No	XSUM	/	22.89	11.66	6.55	4.01

Table 4.20: BLEU scores of models fine-tuned using reinforcement learning.

We can see a small increase in performance when fine-tuning with the reinforcement training. Another thing to note is that the learning rate that seems to provide the best results is ten times greater than the one proposed in the XSUM paper. This could be explained by the fact that their model might have been trained using the cross-entropy loss until full convergence, and therefore required a smaller learning rate during the reinforcement training part. However, if that is the case, the best idea would be to stop training using the cross-entropy loss before reaching full convergence, and to slowly start integrating the reinforcement learning loss through curriculum learning, as we showed here that training using reinforcement learning while the model had not reached full convergence led to scores that were higher than if we had continued training with the cross-entropy loss, all other things being equal. It is possible that waiting for the model to converge before changing the training strategy may only result in slight improvements, whereas changing the strategy when the model has not yet converged may lead the model to converge to a state unobtainable otherwise that exhibits better performance and fluency.

When looking at the ROUGE scores provided in Table 4.21, which is the metric that was used in the training, we also observe a greater increase in performance for the bigger learning rate, although they are more nuanced.

[6]The model that was training on CNN/DailyMail got halted, I therefore only report XSUM results here.

Use of RL	Dataset	Comments	ROUGE-1	ROUGE-2	ROUGE-L
Yes	XSUM	lr set to 0.001	25.18	6.64	20.78
Yes	XSUM	lr set to 0.0001	25.04	6.61	20.62
No	XSUM	Baseline	24.89	6.43	20.34

Table 4.21: ROUGE scores of models fine-tuned using reinforcement learning.

Conclusion of the experiment

As a conclusion, we can say that the increase in performance observed through the use of reinforcement learning may be worth it, provided that we do not wait until the full convergence of the model with respect to the cross-entropy loss before switching the training strategy, or else the gains in performance might be small and may not make the added computations worth it.

8 Unrealized experiments

In this section, I list the experiments that I would have liked to conduct but could not due to time constraints, as well as some ideas for future works.

8.1 Hierarchical Attention

First of all, I would have liked to include **hierarchical attention** in the models. This is a concept that has been used by *Yang et al.* in 2016 in their paper *"Hierarchical Attention Networks for Document Classification"*[47].

Although the tasks are different and the authors use an RNN-based network, I found the ideas interesting and worthy of a try, had I the time to do so.

The authors included two levels of attention in their models, as they observed that the importance of words and sentences depend highly on their contexts. The use of both word-level and sentence-level attention mechanisms allow the model to pay more or less attention to words and sentences when encoding the document. Their network, called the Hierarchical Attention Network (HAN) therefore consists of a word sequence encoder, a sentence sequence encoder, a word-level attention layer and a sentence-level attention.

Both word-level and sentence-level attentions simply follow Bahdanau's work on attention. The word attentions are weighted and summed to compute the sentence vector that is then encoded and attended to. It can be noted that *Nallapati et al.*[24] used that mechanism for their RNN-based abstractive summarization model.

The reason why I did not list this experiment as a high-priority one is because the lengths of our texts are cut after 400 tokens, whereas the authors work with whole documents, and the use of a convolutional architecture containing stacked convolutional blocks already creates some sort of hierarchical structure. My guess and hypothesis were that such an

improvement would mainly be amenable to RNN-based sequence-to-sequence models, and less so to CNN-based ones.

8.2 Introduction of topic information in the models

In the paper introducing the *XSUM* dataset[25], the authors added topical information in the model, through their so-called *Topic Sensitive Embeddings*. In that framework, the topic distributions are obtained via Latent Dirichlet Allocation, and are passed as additional parameters in the embeddings. During the decoding phase, every word prediction is then conditioned on the document topic, forcing the summary to have the same theme as the document.

Although the idea is interesting, the gains in performance they reached, when compared with the traditional convolutional sequence-to-sequence model, were small and, looking at the generated summaries by myself, I found that the models already grasped the topic of the document quite well without that costly addition. Had I had the time to do so, however, I would have liked to explore and implement that concept.

8.3 Use of Intra-attention on top of regular attention

In their paper *Controllable Abstractive Summarization*[16], *Fan et al.* mention that they used, on top of the Bahdanau-like attention mechanism that is frequent in sequence-to-sequence networks, a mechanism called intra-attention that enables the model to refer back to words it previously generated, effectively reducing the repetition of information. However, aside from saying that they alternate both kinds of attention at each layer, they make no mention of the specifics of the implementation, nor are they mentioned anywhere else. From those limited details, I tried to implement my own intra-attention mechanism, but reached bad results that showed that what I implemented did not allow the network to refer back to previously generated words, since the predictions typically contained the same word repeated at least twice. I therefore did not explore this concept further, but would have liked to make it work.

8.4 Ideas for future works

Among the experiments that should have run but got halted in their execution multiple times, the use of the pointer-generator architecture on the CNN/DailyMail network is the one that interested me the most, for the reasons outlined in the relevant section, namely that the increase in performance obtained on that dataset should be even better than the one obtained on the XSUM dataset due to its more extractive nature. This is an experiment that should be interesting to run in the future.

Another idea, already outlined in the review of the literature, would be the implementation of a convolutional GAN architecture for the task of abstractive summarization. Although it has been done for RNN-based networks, and shows better and more legible summaries, it has not been conducted on CNN-based networks, and should similarly increase the quality of the summaries. Even though the computational power required to train such a network would be greater than the one required to train a vanilla CNN-based network,

it should be interesting to know whether that increase in power and hyparameter tuning time is worth it when compared with the increase in performance.

Chapter 5

Conclusion

1 General conclusions about the use of convolutional sequence-to-sequence networks for the task of Abstractive Summarization

This section summarizes the questions and concerns that sparked the subject of this thesis, and provides my personal answers with regards to the various experiments conducted.

The idea of convolutional sequence-to-sequence networks applied to the task of abstractive summarization came from two observations regarding the usual RNN-based sequence-to-sequence networks that were typically used for the task : first of all, their training was tricky, partly due to the problems of exploding and vanishing gradients; secondly, their training was computationally costly.

The convolutional sequence-to-sequence network came as an answer to both of these issues, due to its simplified path from output to input and its ability to parallelize training along sequences. Furthermore, this kind of network exhibited an inherent ability to model and factor in dependencies in the input document, thanks to its stacked convolutional structure allowing for the neighboring words to be taken into account when the input was fed to the model.

However, despite those advantages, works on convolutional sequence-to-sequence models have not been as numerous as those of their RNN-based counterparts, and the preferred methods recommended to address the issue of text summarization still seemed to rely on a RNN-based structure.

This thesis therefore had the ambition to explore more deeply a convolutional architecture that was faster to train and more adapted to the limited resources at our dispositions, while answering some questions that had been overlooked by the literature, to the best of my knowledge.

We started by comparing the training times and performances of both a RNN-based network and a CNN-based network. The results corroborated our intuitions, as the RNN-

based network required more than 7 times the training time the convolutional network did, while not reaching performance gains that made it worth that time increase.

Afterwards, we explored several architecture and training choices, firstly by showing that increasing the number of convolutional blocks in the decoder and the encoder of the networks made sense up to a certain point, as increasing the complexity of the model in such a way ended up bringing the need for longer training to obtain slight improvements in performance. In the context of low-resource training, we found that using more than 8 convolutional layers in the networks did not make sense.

We then explored the use of pretrained embeddings in the models, which is not something that is the default for the task. We showed that the use of pretrained embeddings made sense in the practical context of task transfer (evaluating the model on a dataset different than the one it was trained on), and hypothesized that, due to the generalization capabilities they brought, they were less likely to show state-of-the-art results on the dataset the models were trained on than their trainable counterparts. The use of trainable embeddings and the dataset bias they bring, however, could make sense if we know that the data the model will be evaluated on is close to the one it was trained on, as the bias then plays in the favor of the evaluator.

Afterwards, we saw that the use of a convolutional pointer-generator architecture made sense when compared both to the vanilla network and the Byte-Pair Encoding scheme, as the increase in training time was slight and the summaries generated were of a better quality. The use of that architecture also eliminated `unknown` tokens from the generated summaries. My guess is that this architecture should fare even better on the CNN/DailyMail dataset, but that experiment could unfortunately not be conducted.

Then, we continued with an experiment that explored whether or not the use of reinforcement learning in convolutional networks made sense, the conclusion being that it does for fine-tuning the model for several epochs near the end of its training, but it should not be brought in too early in the training process at the risk of deteriorating performances.

Finally, we concluded this thesis with aborted experiments and ideas for future works.

2 My recommendations for training a model for insurance policy summarization

As NRB, which proposed the topic of this thesis, would have use of text summarization in the context of summarizing insurance policy texts written in various languages, I will provide my recommendations for going about that task.

As the texts would be in French, Dutch and German, the first thing to do would be to collect / create either a single summarization dataset for insurance texts in one of the three languages, or to create one per language. Without a minimum of data of this kind, I do not believe the summaries would be any good, or at least not useful, since a human supervisor would still have to do a lot of post-processing work on the generated summaries, as the domain of such texts is not close to the ones of the usual summarization datasets (those are more news article-based).

Furthermore, it is my opinion that collecting only one dataset in a given language would be preferable, as you would obtain a dataset thrice as big with the same amount of resources as you would trying to collect three datasets. Two models would thus need to be used for the summarization task : one in charge of learning the summarization in a given language, and another one in charge of translating the summaries in the other two languages. Such a neural machine translation model would not even need to be retrained by the team, since they could make use of the already available and highly accurate DeepL.

With regards to the summarization method, training a model from scratch would require a lot of data to be collected, and would therefore be very expensive. My recommendation would be to use a pre-existing model and fine-tuning it with the limited dataset mentioned above. In such a setting, the team takes a pretrained model and lets it run for some epochs on the fine-tuning data to make the network fit for the new domain. Would the network have to be convolutional ? Not necessarily. If the team does not have to train the whole network but simply has to fine-tune it for a few epochs, it might be interesting to turn to a state-of-the-art model such as Pegasus, developed by Google and only requiring a small number of fine-tuning data (mentioned by the authors here[20]) to reach near state-of-the-art performances on different datasets and that has the fine-tuning procedure readily available on Github[29].

Appendices

Appendix A

Problems encountered Appendix

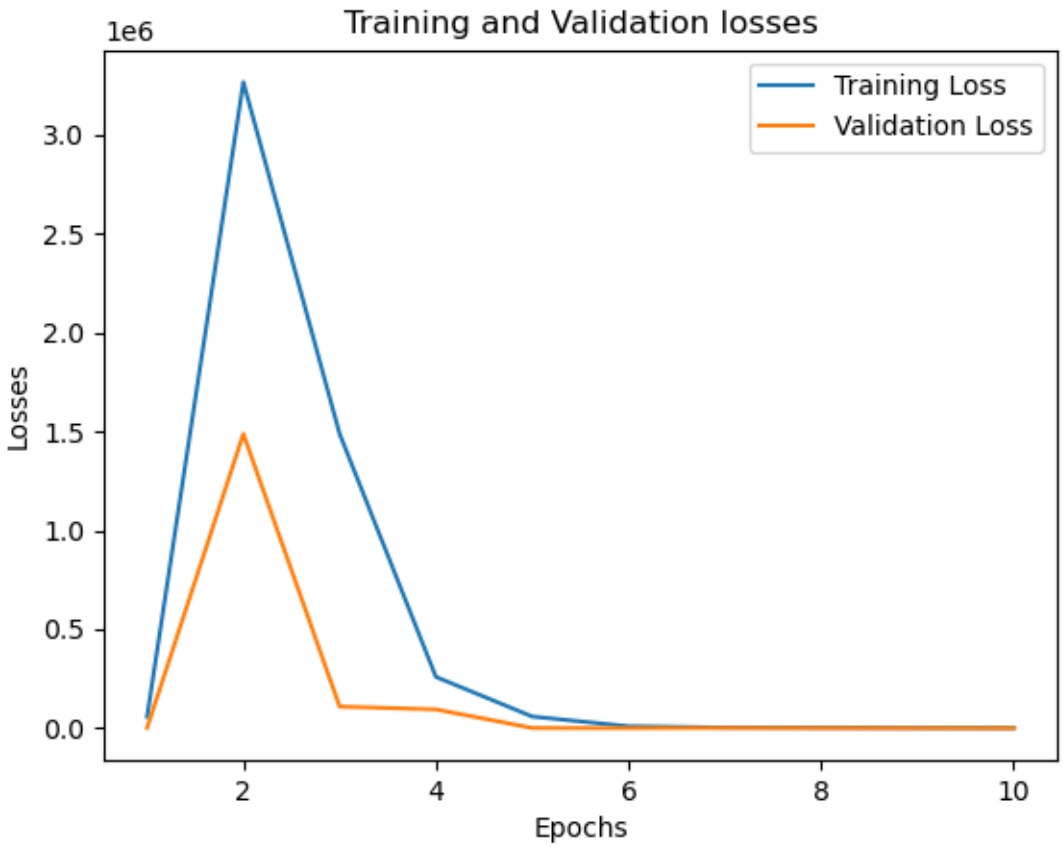

Figure A.1: Training and Validation losses of a model (8 convolutional blocks in both encoder and decoder, no pretrained embedding) trained with Adam optimizer on XSUM dataset.

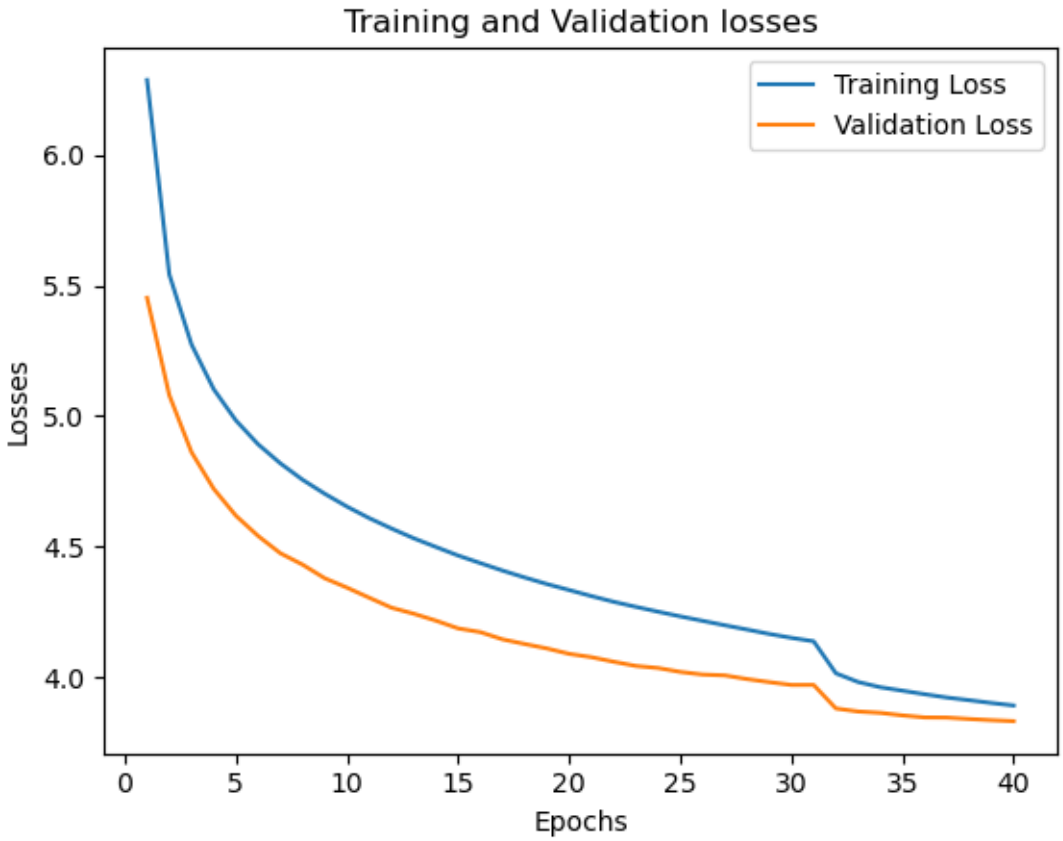

Figure A.2: Training and Validation losses of a model (8 convolutional blocks in both encoder and decoder, pretrained embedding) trained with SGD optimizer with Nesterov momentum on XSUM dataset.

Bibliography

[1] *A Gentle Introduction to Calculating the BLEU Score for Text in Python*. https://machinelearningmastery.com/calculate-bleu-score-for-text-python/#:~:text=The%20Bilingual%20Evaluation%20Understudy%20Score,in%20a%20score%20of%200.0..

[2] *A Neural Attention Model for Abstractive Sentence Summarization*. https://arxiv.org/pdf/1509.00685.pdf.

[3] *A Simple Introduction to Sequence to Sequence Models*. https://www.analyticsvidhya.com/blog/2020/08/a-simple-introduction-to-sequence-to-sequence-models/.

[4] *Abstractive Text Summarization using Sequence-to-sequence RNNs and Beyond*. https://arxiv.org/pdf/1602.06023.pdf.

[5] *An intro to ROUGE, and how to use it to evaluate summaries*. https://www.freecodecamp.org/news/what-is-rouge-and-how-it-works-for-evaluation-of-summaries-e059fb8ac840/.

[6] *Attention Is All You Need*. https://papers.nips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL].

[8] *Brief history of Text Summarization*. https://medium.com/@prasasthy.sanal/brief-history-of-text-summarization-9d1b3787a707.

[9] *Byte Pair Encoding — The Dark Horse of Modern NLP*. https://towardsdatascience.com/byte-pair-encoding-the-dark-horse-of-modern-nlp-eb36c7df4f10.

[10] Sumit Chopra, Michael Auli, and Alexander M. Rush. "Abstractive Sentence Summarization with Attentive Recurrent Neural Networks". In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. San Diego, California: Association for Computational Linguistics, June 2016, pp. 93–98. DOI: 10.18653/v1/N16-1012. URL: https://www.aclweb.org/anthology/N16-1012.

[11] *Convolutional Neural Networks IBM Page*. https://www.ibm.com/cloud/learn/convolutional-neural-networks.

[12] *Convolutional Sequence to Sequence Learning*. https://arxiv.org/pdf/1705.03122v3.pdf.

[13] *Convolutional Sequence-to-Sequence learning Github page*. https://github.com/bentrevett/pytorch-seq2seq/blob/master/5%20-%20Convolutional%20Sequence%20to%20Sequence%20Learning.ipynb.

[14] *Convolutional Sequence-to-Sequence learning with Tensorflow Github page*. https://github.com/tobyyouup/conv_seq2seq.
[15] *Cutting-off Redundant Repeating Generations for Neural Abstractive Summarization*. https://www.aclweb.org/anthology/E17-2047.pdf.
[16] Angela Fan, David Grangier, and Michael Auli. "Controllable Abstractive Summarization". In: Jan. 2018, pp. 45–54. DOI: 10.18653/v1/W18-2706.
[17] *Foundations of NLP Explained Visually: Beam Search, How It Works*. https://towardsdatascience.com/foundations-of-nlp-explained-visually-beam-search-how-it-works-1586b9849a24.
[18] Jonas Gehring et al. *Convolutional Sequence to Sequence Learning*. 2017. arXiv: 1705.03122 [cs.CL].
[19] *Get To The Point: Summarization with Pointer-Generator Networks*. https://www.aclweb.org/anthology/P17-1099.pdf.
[20] *Google AI Blog - PEGASUS: A State-of-the-Art Model for Abstractive Text Summarization*. https://ai.googleblog.com/2020/06/pegasus-state-of-art-model-for.html.
[21] *Google Developers - Text Classification Webpage*. https://developers.google.com/machine-learning/guides/text-classification/step-3#figure-7.
[22] Nal Kalchbrenner et al. *Neural Machine Translation in Linear Time*. 2016. arXiv: 1610.10099 [cs.CL].
[23] Linqing Liu et al. "Generative Adversarial Network for Abstractive Text Summarization". In: 2018. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16238/16492.
[24] Ramesh Nallapati et al. *Abstractive Text Summarization Using Sequence-to-Sequence RNNs and Beyond*. 2016. arXiv: 1602.06023 [cs.CL].
[25] Shashi Narayan, Shay B. Cohen, and Mirella Lapata. *Don't Give Me the Details, Just the Summary! Topic-Aware Convolutional Neural Networks for Extreme Summarization*. 2018. arXiv: 1808.08745 [cs.CL].
[26] *Neural Abstractive Text Summarization with Sequence-to-Sequence Models*. https://arxiv.org/pdf/1812.02303.pdf.
[27] *Neural Headline Generation with Sentence-wise Optimization*. https://arxiv.org/abs/1604.01904.
[28] *Neural Machine Translation in Linear Time*. https://arxiv.org/pdf/1610.10099.pdf.
[29] *PEGASUS: A State-of-the-Art Model for Abstractive Text Summarization - Google AI Blog*. https://ai.googleblog.com/2020/06/pegasus-state-of-art-model-for.html.
[30] *Pointer-Generator Github page*. https://github.com/jiminsun/pointer-generator/.
[31] Marc'Aurelio Ranzato et al. *Sequence Level Training with Recurrent Neural Networks*. 2016. arXiv: 1511.06732 [cs.LG].
[32] Steven J. Rennie et al. *Self-critical Sequence Training for Image Captioning*. 2017. arXiv: 1612.00563 [cs.LG].
[33] Alexander M. Rush, Sumit Chopra, and Jason Weston. *A Neural Attention Model for Abstractive Sentence Summarization*. 2015. arXiv: 1509.00685 [cs.CL].
[34] *Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks*. https://arxiv.org/pdf/1506.03099.pdf.

[35] Abigail See, Peter J. Liu, and Christopher D. Manning. *Get To The Point: Summarization with Pointer-Generator Networks*. 2017. arXiv: 1704.04368 [cs.CL].

[36] *sequence-to-sequence illustration*. https://zhuanlan.zhihu.com/p/27608348.

[37] *Sequence-to-Sequence models with Pytorch Github page*. https://github.com/bentrevett/pytorch-seq2seq.

[38] Tian Shi et al. *Neural Abstractive Text Summarization with Sequence-to-Sequence Models*. 2020. arXiv: 1812.02303 [cs.CL].

[39] *Summarizing Text with Amazon Reviews*. https://www.kaggle.com/currie32/summarizing-text-with-amazon-reviews.

[40] *Text Summarization Evaluation - BLEU vs ROUGE*. https://stackoverflow.com/questions/38045290/text-summarization-evaluation-bleu-vs-rouge.

[41] *The Most Common Evaluation Metrics In NLP*. https://towardsdatascience.com/the-most-common-evaluation-metrics-in-nlp-ced6a763ac8b.

[42] Li Wang et al. *A Reinforced Topic-Aware Convolutional Sequence-to-Sequence Model for Abstractive Text Summarization*. 2020. arXiv: 1805.03616 [cs.CL].

[43] *What are recurrent neural networks?* https://www.ibm.com/cloud/learn/recurrent-neural-networks.

[44] Wikipedia. *Automatic summarization*. URL: https://en.wikipedia.org/wiki/Automatic_summarization. (accessed : 10.10.2020).

[45] *Word Embeddings in NLP*. https://www.geeksforgeeks.org/word-embeddings-in-nlp/.

[46] Tianyang Xu and Chunyun Zhang. *Reinforced Generative Adversarial Network for Abstractive Text Summarization*. 2021. arXiv: 2105.15176 [cs.CL].

[47] Zichao Yang et al. "Hierarchical Attention Networks for Document Classification". In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. San Diego, California: Association for Computational Linguistics, June 2016, pp. 1480–1489. DOI: 10.18653/v1/N16-1174. URL: https://aclanthology.org/N16-1174.