UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS num. 123456

Deep Neural Models for Extractive Text Summarization

Ivan Lovrenčić

Zagreb, February 2022.

CONTENTS

1. Introduction

We live in an age of nearly unlimited data, with textual corpora in almost any given language overflowing the internet. Sites like Facebook and Twitter generate unbelievable amounts of textual records, but even structured data, like research papers, articles, or blog posts, is being produced in large quantities.[1] This growth has pushed the Natural Language Processing (NLP) field to new heights, offering researchers vast amounts of structured and easily accessible data to train up-and-coming deep learning models. As a result, text summarization, an NLP sub-domain devoted to creating an informationally-rich and condensed subset of the original text, is simultaneously thriving on large corpora of textual data and potentially solving the text-overflowing problem in itself.

Text summarization is a sub-field of natural language processing. As such, the aim of text summarization is to generate the subset of the original text that contains all the most relevant or significant information from the original content. Two main sub-domains within the text summarization field are extractive and abstractive summarization. Both approaches focus on the similar task, but there are some discernible differences. The most noteworthy distinction is that abstractive summarization is a considerably more challenging task because it aims to generate original summaries that strongly resemble something that humans might write. In contrast, extractive merely focuses on just locating the most salient parts of the original text and merging them.

Since abstractive summarization requires much more powerful tools, initial summarization attempts had to be strictly extractive, as the researchers could use only a tiny portion of now-available NLP mechanisms to identify the most salient sentences in the text. One of those initial attempts was Hans Peter Luhn's algorithm from 1958. (Luhn, 1958). Like many other algorithms from that time, Luhn's algorithm heavily relied on linguistic rules and features like syntactic parsing, grammar checking, and/or huge corpora of often-used phrases and sentences (Jing and McKeown, 2000). Consequently, the algorithms were crude and not scalable, as they required lots of external knowledge that was not easily transferable.

Fortunately, with the rise of machine learning and especially deep learning, the tides began to turn. Currently, in the age of powerful neural architectures that reach human-level

[1] https://www.dsayce.com/social-media/tweets-day/

performance and seemingly understand language, the researchers shifted their focus from early-day rule-based systems to entirely data-driven approaches. This shift allowed models to be more versatile, more scalable and, most importantly, more transferable. Consequently, researchers are constantly developing novel models that prosper from these large architectures and enormous quantities of easily-accessible internet data (Zhong et al., 2019).

However, this growth appears to be slowing down, especially in the context of extractive summarization (Liu, 2019). Initially, the issue seemed to be in the neural architectures. Recurrent neural networks, a use-to-be state-of-the-art in text summarization, was an obvious bottleneck. Because of their inherent sequential nature, they preclude parallelization of training (Vaswani et al., 2017). Yet, even once the Transformers solved the problem of training time, the growth has continued to stagnate.

Currently, researchers believe that the problem of neural extractive summarization is in its intrinsic greedy nature. Specifically, most current neural extractive models follow a similar path: (1) encode sentences individually, (2) encode a document to add context and (3) find a subset of sentences that belong in the summary (Zhong et al., 2019). The central issue here is that the decision which sentences should be in the summary is made individually for each sentence. This approach favours the most salient and most generalized sentences, which will result in a summary that contains similar sentences with redundant information.

The focus of this thesis is a novel approach in extractive summarization that concentrates on fixing the problem of sentence redundancy. The idea and the model for this thesis were largely motivated by work of Zhong et al. (2020). In their paper, Zhong et al. (2020) propose a novel summary-level summarization model, whose objective is to generate summary candidates and then use semantic text matching to determine which summary is most similar to the original document. This way, we change the problem from a sentence-level to a summary-level task. In this thesis, I will demonstrate the inner workings of this novel approach and how the commonly used sentence-level extractive methods fall short compared to this new text-matching proposal.

2. Related Work

In the beginning, research in extractive summarization was primarily focused on methods that strictly relied on human-engineered features and/or large corpora of external knowledge. Those approaches can be now broadly classified into three major groups: (1) greedy approaches (e.g. (Carbonell and Goldstein, 1998)), (2) graph-based approaches (e.g. (Erkan and Radev, 2004)) and (3) constraint optimization-based approaches (e.g. (McDonald, 2007)). However, recently, with remarkable results in the field of deep-learning, researchers shifted their focus from those conventional ways to the contemporary data-driven deep-learning approaches.

One of the first examples of a fully neural end-to-end approach in extractive summarization was Kågebäck et al. (2014). The authors leveraged a recursive auto-encoder to summarize documents, and it's one of the first instances of using neural networks for extractive document summarization. Soon after, Cheng and Lapata (2016), proposed a fully automated recurrent neural network (RNN) model for extractive summarization. As with most neural approaches, they used an encoder-decoder architecture, where the encoder learns the representation of sentences and documents, while the decoder classifies each sentence based on the encoder's representations. Shortly after, Nallapati et al. (2017) proposed a similar approach, but their model employed a single sequence model with no decoder and therefore had comparatively fewer parameters.

Both (Cheng and Lapata, 2016) and (Nallapati et al., 2017) formulated extractive summarization as a sequence classification problem and solved it with an encoder-decoder framework. However, because these models make independent binary decisions for each sentence, the summaries have high redundancy (Zhong et al., 2020). To address these issues, several solutions were proposed such as the introduction of the auto-regressive decoder (Zhou et al., 2018) or trigram blocking (Liu and Lapata, 2019). In both examples, the goal is to improve the quality of the summary, by reducing the redundancy of information among sentences.

In the following years, many other works showcased the strengths of neural extractive architectures such as reinforcement learning (Narayan et al., 2018), neural latent learning (Zhang et al., 2018.) and multi-root tree induction (Liu et al., 2019). However, it seemed like an improvement on automatic metrics like ROUGE (Lin, 2004) has reached a bottleneck

due to the complexity of the task (Liu, 2019). One of the bottlenecks was that the commonly used RNNs and their inherent sequential nature, which precludes the parallelization within training examples. This issue becomes critical at longer sequence lengths, as memory constraints limit batching across examples (Vaswani et al., 2017). As a solution, Vaswani et al. (2017) introduced the Transformer, a novel architecture that eschews recurrence and instead relies entirely on an attention mechanism, which allows for significantly more parallelization.

Since then, both extractive and abstractive research has been monopolized by the Transformers. The major advantage of Transformers is the ability to pretrain these huge neural architectures, which can be used as encoders for different natural language processing tasks (Devlin et al., 2018). This offered an immense upgrade for abstractive and extractive summarization. The RNN-based extractive model from Zhou et al. (2018) was recently dethroned by Liu and Lapata (2019) extractive model based on Bidirectional Encoder Representations from Transformers (BERT; Devlin et al. (2018)). The BERT-based model achieved an astonishing 2 points of improvement on the ROUGE automatic metric over the previous state-of-the-art.

The latest state-of-the-art in the extractive field was brought by Zhong et al. (2020), where they framed summarization as a semantic text matching problem. The idea is to generate N candidate summaries from the original text and then use BERT to encode them. Then, the goal is to find the candidate that is semantically the closest to the original document. By generating the whole summaries, the task has been raised from a sentence-level task to the summary level, thus solving the problem of redundancy of information in the summaries (Zhong et al., 2020). The idea behind that paper is also the main idea that will be discussed in this thesis.

3. Deep Learning for Text Summarization

3.1. Text Summarization

Text summarization, as we mentioned in the previous chapter, is a sub-field natural language processing. The goal of text summarization is to create a summary that includes the most important or relevant information within original content (Torres-Moreno, 2014). A necessary distinction to mention is when we refer to text summarization, especially in this thesis, we actually refer to *automatic* text summarization. This is crucial, as automatic text summarization focuses on *computationally* shortening a set of data.

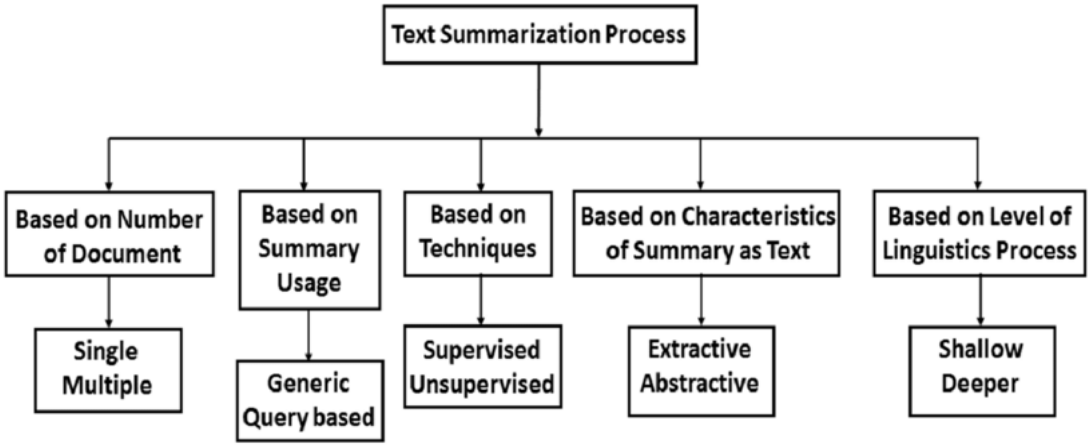

Figure 3.1: Classification of summarization techniques (Gupta and Gupta, 2019)

Automatic text summarization is a vast and intricate field with many sub-fields. Figure 3.1 shows us the variety of categorizations within the domain of text summarization. In the context of this thesis, we will focus strictly on extractive, single-document and generic summarization. We will cover both the supervised and unsupervised text summarization approaches. Also, since we will focus only on deep learning methods, the categorization based on the linguistics processes is not applicable in our case.

3.1.1. Extractive Summarization

Extractive text summarization is a sub-domain of automatic text summarization. Extractive summarization aims to find the most salient sentences within the original text, then extract and merge them in one summary. In Figure 3.2., we illustrate the extractive approach.

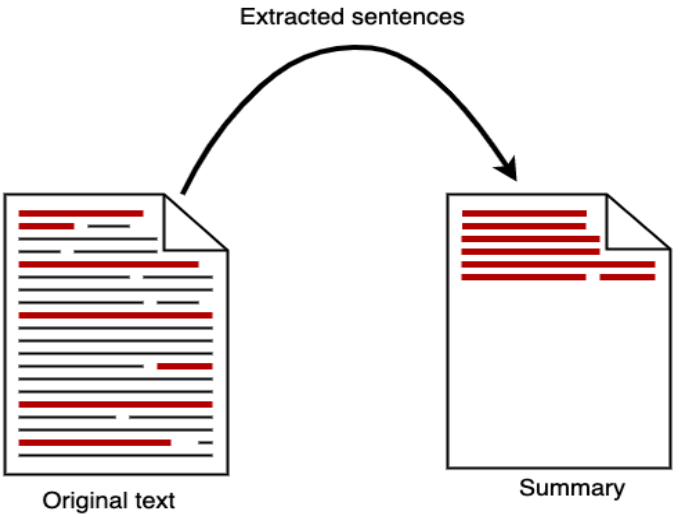

Figure 3.2: Extractive summarization

Formally, in single-document extractive summarization, we refer to a single document consisting of n sentences as a $D = (s_1, ..., s_n)$. Each document has a gold summary $G = (s_1, ..., s_m)$, which consists of m sentences (where $m \leq n$) from the original document. The goal of extractive summarization is to extract and form the candidate summary from the original text that maximizes the evaluation metrics such as ROUGE (Lin, 2004).

As mentioned earlier, nowadays, text summarization research is predominated by deep learning models. This is because deep learning provides end-to-end, scalable and transferable models. Basically, researchers can achieve state-of-the-art results without any external composites of language knowledge or intricate rule systems. Consequently, the only focus of this thesis will be strictly on the deep learning methods for extractive summarization.

Luckily, in general, most of existing neural extractive summarization systems can be abstracted into the following framework, consisting of three major modules: (1) sentence encoder, (2) document encoder and (3) decoder (Zhong et al., 2019). At first, a sentence encoder will be utilized to convert each sentence s_i into a sentential representation $\boldsymbol{d_i}$. Then, these sentence representations will be contextualized by a document encoder into $\boldsymbol{c_i}$ (Zhong et al., 2019). Finally, a decoder will extract a subset of sentences based on these contextualized sentence representations (Zhong et al., 2019).

Sentence encoder

A sentence encoder can be any deep neural network that can map sequences s_i = (x_1,...,$x_{|s_i|}$) (in this case sentences) into a fixed-length vector that then can be used later in the model. A traditional choice for this architecture are convolutional neural networks (CNN; LeCun et al. (1999)), however, researchers Kedzie et al. (2018) have shown that the choice of the architecture for the sentence encoder does not affect the overall performance. In theory, we can use any of the neural architectures that we will go over in the following chapters.

Document encoder

The job of the document encoder is to give context to sentence encodings. Without context, sentence encodings can't provide enough information to the decoder to decide whether the sentence should be included in the final summary. Given a sequence of sentence encodings ($\boldsymbol{d_1}, ..., \boldsymbol{d_n}$), the task of document encoder is to contextualize each sentence therefore obtaining the contextualized representations ($\boldsymbol{c_1}, ..., \boldsymbol{c_n}$) (Zhong et al., 2019). Currently, for document encoders, most extractive approaches rely on two architectures - Long short-term memory (LSTM; Hochreiter and Schmidhuber (1997)) networks and Transformers (Vaswani et al., 2017). However, similarly to the sentence encoder, we can theoretically employ any of the neural architectures we will discuss in the following chapters.

Decoder

The decoder is used to extract a subset of sentences from the original document based on contextualized representations: ($\boldsymbol{c_1}, ..., \boldsymbol{c_n}$). Most existing decoder architectures can be divided into *autoregressive* and *non autoregressive* versions (Zhong et al., 2019). An autoregressive decoder is a type of decoder that takes previous predictions in consideration when predicting a new output. Based on the type of decoder, we can frame extractive summarization in two tasks - **sequence labelling** and **pointer networks**.

Sequence labelling In sequence labelling, models are generally equipped with a non autoregressive decoder (Zhong et al., 2019). Given the document D = ($s_1, ..., s_n$), the summaries are extracted by predicting a sequence of labels ($y_1, ..., y_n$) ($y_i \in \{0, 1\}$) for the document, where $y_i = 1$ means that ith sentence should be included in the summary.

Pointer Networks As a representative of auto-regressive decoder, the pointer network-based decoder has shown superior performance for extractive summarization (Zhong et al., 2019). The Pointer network employs an attention mechanism when selecting sentences. This

way, for each binary decision, the network is aware of the previous predictions, and it can use that knowledge to make a better decision.

3.2. Deep Learning

Deep learning (Bengio et al., 2017) is a subfield of machine learning (ML) and artificial intelligence (AI) that relies on artificial neural networks for modelling problems from a broad list of fields such as computer vision, bioinformatics, natural language processing and speech recognition. Two main learning setups are commonly considered in machine learning - *supervised* and *unsupervised* learning.

In supervised learning, the dataset consists of both the data samples and corresponding labels, $D = \{(x_i, y_i)\}_{i=0}^{N}$, where x_i corresponds to the ith data sample and y_i is the corresponding label. Each data sample $x \in \mathbb{R}^n$ is a vector of features. Supervised learning aims to find an underlying function that maps all data samples X into the corresponding outputs (labels) Y. There are two types of supervised learning - classification and regression. In classification, y are discrete classes, whereas, in regression, it is a continuous value.

On the other hand, unsupervised learning does not require the corresponding data labels. Consequently, unsupervised learning models have become quite prevalent, as they often demand little or no human activity and they much easier to train. Generally, unsupervised learning is associated with clustering algorithms, which aim to construct meaningful clusters from the data. Other common examples of unsupervised learning are density estimation approaches and dimensionality reduction algorithms. In the context of this thesis, we will go over various text summarization approaches that rely both on supervised and unsupervised learning techniques.

As we already established, deep learning has become the predominant tool for natural language processing tasks (Gupta and Gupta, 2019). In the case of text summarization, both extractive and abstractive summarization approaches strongly depend on the encoder-decoder frameworks. In the light of that, in the next few sections, we will go through standard neural architectures currently still used for text summarization tasks.

3.2.1. Feed-forward neural networks

Feed-forward neural networks are the most basic form of artificial neural networks. Their distinguishing characteristic is that connections between neurons do not form a cycle and that information can only flow forward (hence the name feed-forward) from the input to the output layer. There are no cycles or loops in the network. The building block of all artificial neural networks are artificial neurons.

Perceptron

Artificial neurons and artificial neural networks were originally inspired by their biological counterparts. In 1943., Warren McCulloch and Walter Pitts introduced the first artificial neuron - the *Threshold Logic Unit (TLU)*. As we can see in Figure 3.3., the artificial neuron has several parts: inputs (x_1, x_2, x_3), weights (w_1, w_2, w_3), bias (b_1), and the body, where we multiply inputs with weights and sum them together with the bias. Lastly, each neuron has a non-linear transformation (activation) function $f(x)$ that is applied over the weighted sum. In the case of TLU-perceptron, we use the step function, which returns one for positive values and zeroes for negative.

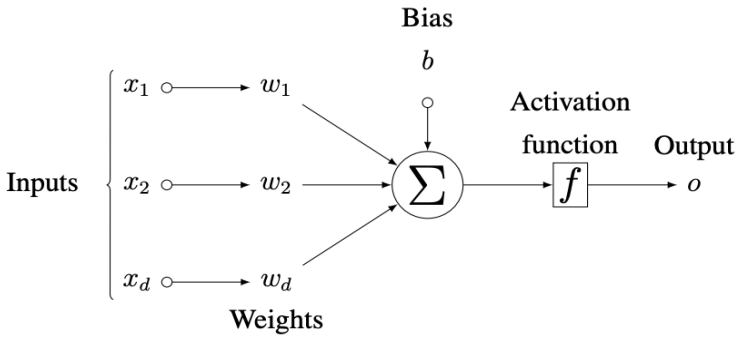

Figure 3.3: McCulloch and Pitts' artificial neuron (1943.)

Initially, this first model was remarkably basic with only binary inputs and outputs and several restrictions on possible weights. More importantly, this first neuron could not actually learn from the data. Effectively, it was a fixed threshold unit with pre-programmed weights for specific use-cases. In 1949., Donald Hebb noticed that biological neurons can actually learn by changing the level of conductivity between two neighbouring neurons. He famously remarked, "Neurons that fire together, wire together". A few years later (1958.), Frank Rosenblatt combined D. Hebb's insight on how biological neurons learn with McCulloch and Pitts' artificial neuron model (1943.). and introduced a simple algorithm that allowed artificial neurons to learn from the data. Algorithm 1. shows us details of his method.

Algorithm 1 Perceptron algorithm (F. Rosenblatt, 1958.)

1: Cyclically go through N training data samples, one by one
2: Classify current data sample

1. If the data sample is classified correctly, do not change the weights
   (a) If all N data samples are classified correctly, stop the algorithm
   (b) If not, then continue to the next data sample
2. If the data sample is not classified correctly, update the weights based on the next rule:
   $w_i(k+1) \leftarrow w_i(k) + \mu * (t - o) * x_i$
   w_i is an ith weight
   μ is a learning rate
   t is a target output
   o is an actual network's output
   x_i is an ith input

Even though the perceptron's algorithm presented a great leap forward, it had one substantial shortcoming - it was a linear classificator. This means that perceptron can only be applied on linearly divisible problems (Minsky and Papert, 2017). Naturally, this introduced a significant limitation, as most problems are not linearly divisible. In Figure 3.4, we can notice an illustration of a relatively simple boolean function as an example of a linearly non-divisible problem. To model these non-linear problems, we have to increase the expressive power of the network. In practice, we can increase expressivity by introducing non-linearity and multiple layers into the network.

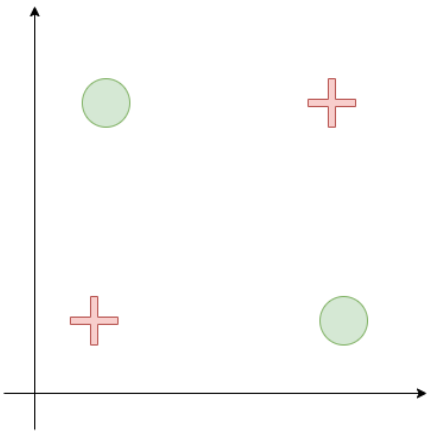

Figure 3.4: Example of linearly non-divisible function - XOR function

Multi-layer perceptron

The artificial neural network with two or more layers is called a multi-layer perceptron (MLP). An example of the multi-layer perceptron is shown in Figure 3.5. As we can notice, in MLP, every neuron is connected with all neurons from the following layer. Every MLP is defined by two hyperparameters - depth (number of layers) and width (number of neurons inside the layer). As an example, the network in Figure 3.5. has three layers - input, hidden and output layer. However, in practice, we would not necessarily count all layers, but rather just the number of hidden layers. Then, in our case, the network has a depth of one. The width of the network is four, as that is the maximum number of neurons in hidden layers.

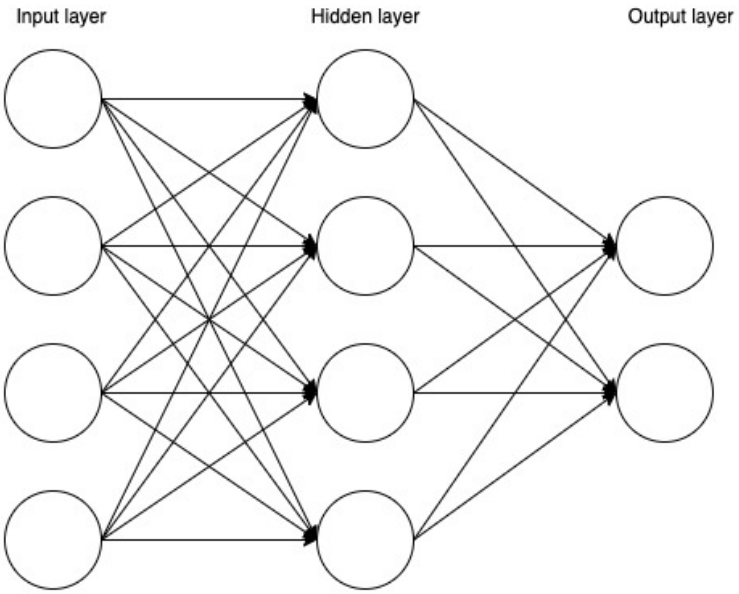

Figure 3.5: Feedforward neural network with several layers

Multi-layer perceptrons are universal approximators. Given a continuous function on a compact set on an n-dimensional space, then there exists a one-hidden-layer feedforward network which approximates the function (Hornik et al., 1989). However, adding more neurons and layers will not necessarily always increase the network's expressivity. A multi-layer perceptron with only linear neurons is identical to just having one linear neuron. Only by introducing non-linear activation functions, the multi-layer perceptron becomes a universal approximator (Hornik et al., 1989). In theory, we can use any monotonically increasing function, but, in practice, we already have several functions that perform remarkably well.

In Table 3.1., we detail several popular activation functions. Each activation function has its advantages and disadvantages. As an example, currently, the ReLU activation function is the recommended function for deep neural networks (Bengio et al., 2017). As opposed to sigmoid and hyperbolic tangent, the ReLU function passes zero whenever the input $x \leq 0$,

Table 3.1: Activation functions

Sigmoid function	$\sigma(x) = (1 + e^{-x})^{-1}$
Hyperbolic Tangent	$tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$
Rectified Linear Unit (ReLU)	$ReLU(x) = max(0, x)$

which translates to sparser representations. Another advantage is that when $x > 0$, ReLU's gradient has a constant value. In contrast, the gradient of the sigmoid function becomes increasingly small as the absolute value of x increases (the vanishing gradient problem). However, the ReLU has a problem with differentiation at zero, and since ReLU doesn't pass values smaller than zero, it can lead to "dead" neurons. In other words, the weights of this "dead" neuron will never be updated again.

Apart from the name, every machine learning algorithm is defined by its model, cost function and optimization procedure. In this case, the model is the feed-forward neural network. The cost (or loss) function is a function that defines how well is the model performing (Zell, 1994). In theory, the cost function could be any (preferably) convex function. However, in practice, there are several functions that are commonly used such as 0-1 loss (Equation 3.1), cross-entropy loss (Equation 3.2) and Mean Squared Error (Equation 3.3).

The training of any neural network is, in a nutshell, an optimization problem. The optimization procedure describes how the network learns. In Figure 3.3., the training of the neuron would consist of finding values for $w1, w2, w3$ and $b1$ that would produce optimal results on the unseen dataset. In order to find those optimal weights, we would have to know how much we have to change the network's parameters. We can do that by using *gradient descent* and the *backpropagation* algorithm that allows the network to propagate the change back through the network (Rumelhart et al., 1985). We can see the full backpropagation algorithm in the Algorithm 2.

– 0-1 loss (classification) -

$$loss_{01}(y, \widehat{y}) = \begin{cases} 1, & \text{if } y == \widehat{y} \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

– Cross-entropy loss (classification) -

$$loss_{CE}(Y, \widehat{Y}) = -\sum logp(\widehat{y_i} = y_i | f(x, \theta)) \tag{3.2}$$

– Mean squared error (regression) -

$$loss_{MSE}(y, \widehat{y}) = \frac{1}{N} \sum_{i=0}^{N} (y - \widehat{y_i})^2 \tag{3.3}$$

Algorithm 2 The backpropagation algorithm

* Initialize network's weights randomly
* Objective: $\min_{w_l^{ij}} E = \frac{1}{N}\sum_{n=1}^{N} E_n$
  - We want to minimize the cost (or energy) function with regards to weights
  - Depending on the cost function E_n, the process changes, but we will use a standard Mean Squared Error (MSE): $E = \frac{1}{2}\sum_{i=1}^{L}(t_i - o_i)^2$

1: Forward pass

1. We pass a data sample $\vec{x} = x_1, x_2, ..., x_n$ through all layers of the network
2. We determine the output of the network $\vec{o} = o_1, o_2, ..., o_l$
3. Based on the loss function* the network uses, we calculate the error E by putting the the output $\vec{o}$ and the true value $\vec{t}$ into the cost function $E = \frac{1}{2}\sum_{i=1}^{L}(t_i - o_i)^2$

2: Backward pass

1. Once we have calculated the error, we want to change the weights of the network, so that the output of the network produces a smaller error. We start with the weights that are connected to the output neurons and then propagate back
2. We can propagate the error and update the weights by using $Generalized\ Delta\ rule$

$$\begin{cases} \Delta w_{ji}^l = \eta * \delta_i^l * x_j^{l-1} = -\eta * \frac{\partial E}{\partial w_{ji}^l} \\ \delta_i^L = (t_i - o_i) * g^{'}(\xi_i^L) \\ \delta_i^l = (\sum_{r=1}^{N_{l+1}} \delta_r^{l+1} w_{ri}^{l+1}) * g^{'}(\xi_i^l), l = 1, ...L-1 \\ x_i^l = g(\xi_i^{l+1}) \\ \xi_i^{l+1} = \sum_{j=1}^{N_l} w_{ji}^{l+1} * xi_j^l \end{cases}$$

* η is a small constant called $learning\ rate$

  t_i is the target output

  o_i is the actual output

  x_i^l is the ith input in the lth layer

  ξ_i^l is the weighted sum of the ith neuron in the lth layer

  $g(x)$ is the neuron's activation function

  $g^{'}$ the derivative of neuron's activation function

  l is the number of the current layer (L is the last layer)

3: Repeat the process until certain conditions are met (e.g. repeat for 10 epochs or until the error is below certain value)

3.2.2. Convolutional neural networks

As mentioned in the previous section, traditional feed-forward neural networks are excellent approximators. However, this expressivity also makes them especially susceptible to overfitting (Hornik et al., 1989). Because each neuron is connected to each neuron from the next layer, the feed-forward network's parameters can grow exponentially. Common ways of preventing overfitting (also referred to as regularization) include penalization of parameters during training (weight decay) and trimming connectivity (dropout layers, skipped connections, etc.).

Convolutional neural networks (CNN; LeCun et al. (1999)) are, fundamentally, a regularized version of multi-layer perceptron (MLP). However, CNN's have a different approach towards regularization. They reduce the number of parameters by taking advantage of hierarchical patterns in data and assembling patterns of increasing complexity using smaller and simpler patterns embossed in their layers. The name "convolutional neural network" indicates that the network employs a mathematical operation called **convolution** (Bengio et al., 2017). CNNs are neural networks that use convolution in place of general matrix multiplication in at least one of their layers (Bengio et al., 2017).

$$s(t) = (x * w)(t) = \int_t x(a)w(t-a)da \tag{3.4}$$

In Equation 3.4. , we can see a mathematical expression for convolution. In its most general form, the convolution is an operation on two functions of a real-valued argument (Bengio et al., 2017). A simple way to explain convolution is that convolution returns the amount of overlap between two different functions. More formally, convolution gives us a weighted average of the function $x(a)$ at the time moment t, where the weights are determined by the weight function $w(-a)$. From the definition, we can see how the weight function w will highlight different parts of function x at different time intervals t. In convolutional network terminology, the first function (in this case $x(a)$) is often referred as the **input**, and the second argument (in this case the function $w(t-a)$) as the **kernel**. The output (or in this case $s(t)$) is sometimes referred as the **feature map** (Bengio et al., 2017).

$$s[t] = (x * w)[t] = \sum_{a=-\infty}^{\infty} x[a]w[t-a] \tag{3.5}$$

The expression in Equation 3.4. is often used in fields such as signal processing, acoustics and electronics. However, if we wish to apply convolution to machine learning problems, we have to use a discrete formula. In Equation 3.5., we can see a discrete expression for convolution. The issue with that equation is that the typical inputs in CNNs are images, which are not a one-dimensional array, but a two-dimensional matrix. To apply convolution to the two-dimensional matrix, we have to employ the Equation 3.6. As we can see from

the equation and in Figure 3.6., two-dimensional convolution is just a matrix dot product between a section of an input matrix and a kernel matrix.

$$s[i,j] = (X * K)[i,j] = \sum_m \sum_n X[i-m, j-n]K[m,n] \tag{3.6}$$

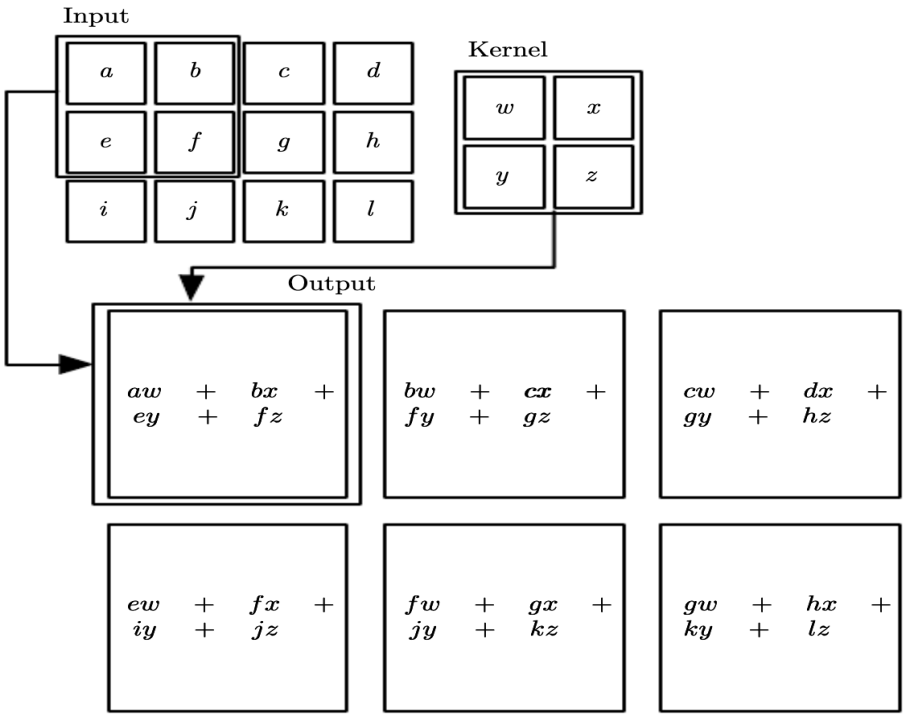

Figure 3.6: An example of 2D convolution (Bengio et al., 2017)

Traditional feed-forward layers use matrix multiplication by a matrix of parameters describing the interaction between the input and output unit. This means that every output interacts with every input unit. However, convolutional neural networks have *sparse interactions*. CNNs achieve that by making the kernel smaller than the input: e.g., an image can have millions or thousands of pixels, but while processing it using the kernel we can detect meaningful information that is of tens or hundreds of pixels. (Bengio et al., 2017). In Figure 3.6. , we can notice, output neurons aren't affected by all the inputs, as they would be in the feed-forward network. Specifically, we can see that the first output (the leftmost box in the output matrix) is only affected by the inputs a, b, e and f. As a result, we store fewer parameters, which not only reduces the memory requirement of the model but also improves the statistical efficiency of the model (Bengio et al., 2017).

Another significant benefit of convolution is *parameter sharing*. Parameter sharing refers to the ability to use the same parameter for more than one function in the network (Bengio et al., 2017). In CNNs, each value of the kernel matrix is used at every position of the

[1]https://towardsdatascience.com/sesn-cec766026179

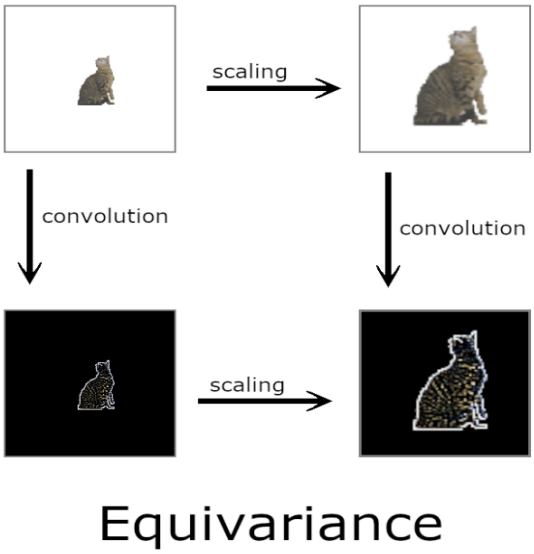

Figure 3.7: The final image will be the same, no matter whether we first do the convolution and then scaling, or first scaling and then convolution[1]

input. As an example, in Figure 3.6, the parameters that we share are the values w, x, y and z. Each output neuron will reuse the same weights w, x, y, z. As we can see, each neuron has a weighted sum that always employs identical w, x, y, z weights, and only the inputs change. CNNs, by utilising this parameter sharing technique, significantly reduce the number of parameters they have to store.

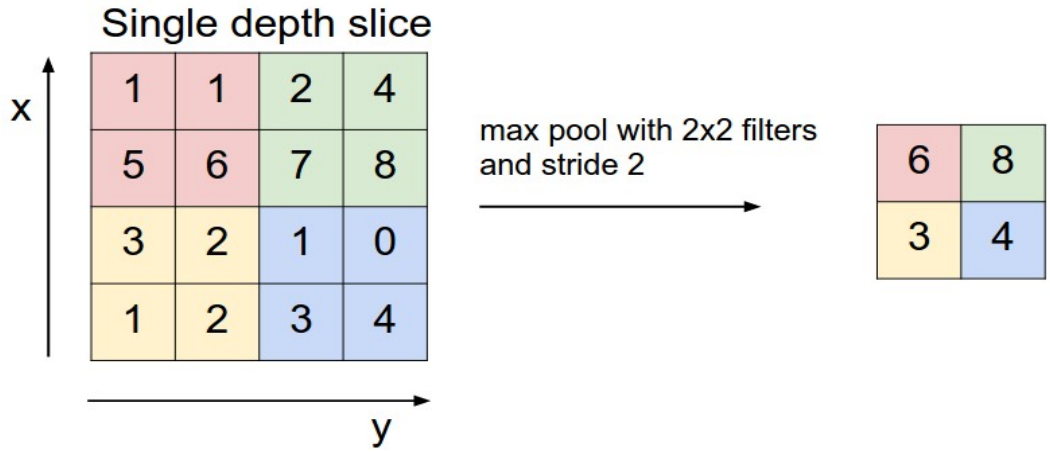

Figure 3.8: 2D Maxpooling example (Bengio et al., 2017)

As an example, in Figure 3.6., if we didn't share parameters, we would have to store six kernel matrices instead of one. Sharing parameters makes sense, as we assume that if computing one feature at some spatial point ($x1$, $y1$) is beneficial, then it should also be valuable to compute that same feature at some other spatial point ($x2$,$y2$). Due to this, CNNs also have a property of *translation equavarience* (Bengio et al., 2017). We can see an

example of equavarience in Figure 3.7.

Apart from the convolutional layers, a significant step for CNNs is the pooling layer. Pooling layers help to reduce the spatial size of the representation, which decreases the required amount of computation and weights (Bengio et al., 2017). Each pooling layer employs a different summary statistic that is then applied over the output. For example, in Figure 3.8, we can observe a two-dimensional max-pooling layer. As we can see, the max-pooling layer reduced the output's size from 4x4 to 2x2 matrix. An additional advantage of the pooling layer is that it helps to make network representations approximately invariant to small translations in the input. Essentially, invariance to translation implies that if we translate the input by a small amount, the values of most of the pooled outputs should not change (Bengio et al., 2017).

3.2.3. Recurrent neural networks

Although both traditional feed-forward neural networks and convolutional neural networks are powerful and expressive networks, they have a major flaw - constrained inputs and outputs. Essentially, the networks have fixed size inputs and outputs, and it is not possible to adjust them while the model is running. This, however, does not present an issue for all deep learning applications, as lots of machine learning tasks have predetermined input and output sizes (such as image classification and so on). But, in the case of text summarization and other similar sequence tasks, where the input sequences are rarely the same length it does present a critical issue. Furthermore, the most substantial disadvantage of traditional FFNNs and CNNs is their incapacity to interpret temporal information. A bit more informally, both FFNNs and CNNs can't pass on the information from previous inputs and then use it with future inputs. Recurrent neural networks (RNN; Rumelhart et al. (1985)) address both of these issues. They are networks with loops in them, allowing information to persist.

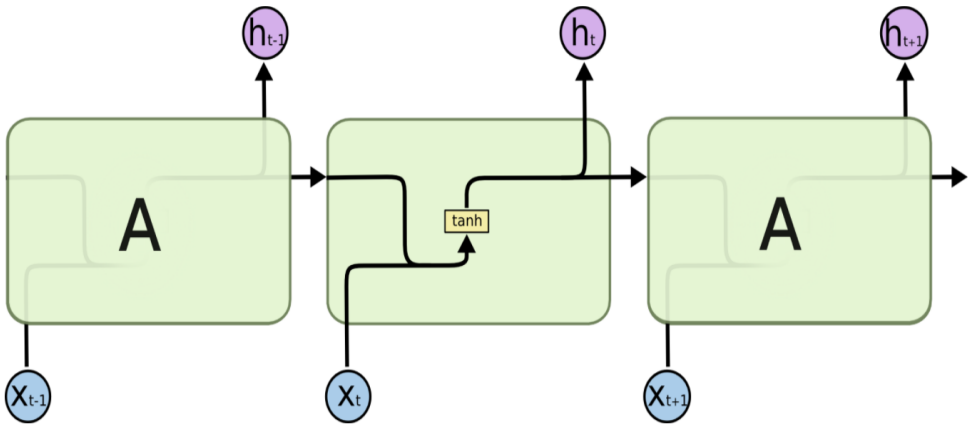

Figure 3.9: An unrolled recurrent neural network[2]

Architecture

Recurrent neural networks are a special type of artificial neural network specialized for processing sequential data (Bengio et al., 2017). Because of their distinct architecture, the information from previous inputs can stream through the network. Additionally, the networks can process variable-length sequences by "unfolding" the network and processing a part of the input sequence at each time step. However, it's important to note that the input to the network is fixed ($\vec{x_t}$ is a fixed-length vector), but the number of those individual vectors $\vec{x_1}, \vec{x_2}, ..., \vec{x_n}$ can be arbitrary per input sequence.

In Figure 3.9, we can notice several components of a standard vanilla RNN. First, the green box with the letter A represents the body of the network (also commonly referred to as a cell). In the cell, we can notice that the information h_{t-1} (also commonly referred to as a "hidden state") from the previous step and the current input x_t are "merging", and then they are passed through the activation function. This newly created hidden state is then passed on, and the whole cycle repeats until the end of the input sequence. In theory, this hidden state should preserve information from all previous inputs, which, in turn, should provide better context for future inputs.

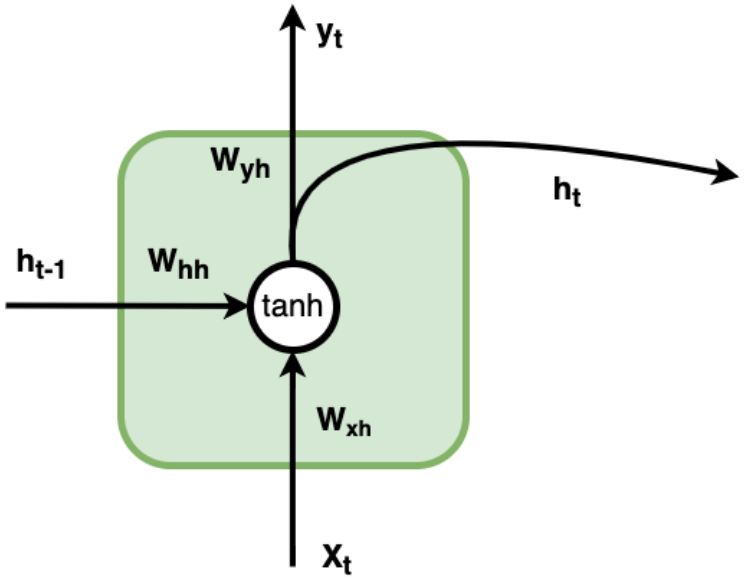

Figure 3.10: RNN cell

Surprisingly, recurrent neural networks have a pretty straightforward architecture. In general, a standard RNN cell has an input layer (labeled as x_t), a hidden layer (labeled as h_t) and an output layer (labeled as y_t). Every layer has its belonging weights and biases. The hidden layer is the crucial part, as it distinguishes RNNs from the rest of the networks. In the hidden layer, the RNN streams the memory from previous inputs. For example, in Figure

[2]http://colah.github.io/posts/2015-08-Understanding-LSTMs/

3.10. in the middle cell, the network is making a decision based on the input x_t, but, also, on the previous state h_{t-1}, which "stores" the information from all the previous inputs.

Similarly to CNNs, recurrent neural networks also share parameters. As we can notice in Figure 3.10, a standard vanilla RNN cell has three groups of weights - $\boldsymbol{W_{hh}}$ (weights of the hidden layer), $\boldsymbol{W_{yh}}$ (weights of the output layer) and $\boldsymbol{W_{xh}}$ (weights of the input layer). These weights (parameters) are then shared throughout the input sequence. For example, for a sequence s_i = $\{\vec{x_1}, \vec{x_2}, \vec{x_3}\}$ the network will have to "unfold" three times. Each fold, the network will process a $\vec{x_i}$, but it will use the same weights. This is parameter sharing. In Equation 3.7., we can see expressions that define the network's internal state and an output.

$$\begin{aligned} \boldsymbol{h_t} &= tanh(\boldsymbol{W_{hh}} * \boldsymbol{h_{t-1}} + \boldsymbol{W_{xh}} * \boldsymbol{x_t}) \\ \boldsymbol{y_t} &= \boldsymbol{W_{yh}} * \boldsymbol{h_t} \end{aligned} \tag{3.7}$$

We can notice, the equation for $\boldsymbol{y_t}$ boils down to a few dot products. To calculate the new hidden state, we simply sum the previous state's and the current input's dot product together and pass it through the $tanh$ function. Here, we form a new "memory" by modifying the hidden state. The network's output is a dot product of the new hidden state and the weights. In practice, researchers often pass the output through an activation function such as softmax. That way, the output is the probability distribution, which is human-interpretable. A meaningful parameter in the network is the size of the hidden state vector $\vec{\boldsymbol{h_t}}$. In general, the larger the vector, the more memory it can store. However, if we increase the vector's size too much, the network's learning can substantially slow down.

Optimization of RNNs

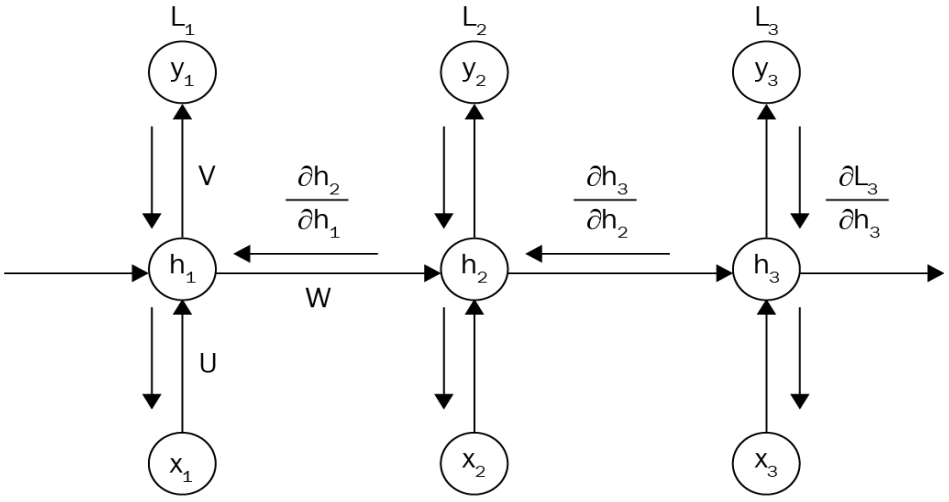

Figure 3.11: Backpropagation through time in RNN

RNNs, similarly to previous neural architectures, rely on gradient descent and backpropagation to train and learn from the data (Sutskever et al., 2014). However, since we are now dealing with the sequential data, RNNs use a unique version of backpropagation called $backpropagation\ through\ time$ (BPTT; Sutskever et al. (2014)). In Figure 3.11., we can notice an example of how the backpropagation functions in RNNs. As we can see, because of RNN's unfolding, we have to propagate the error throughout all the folds back. This unfolding, as we mentioned, comes from the fact that RNNs process input sequentially.

In Figure 3.11., we can notice how the backpropagation begins at the last output and moves back in time. If we look back to the Equation 3.7., and apply the chain rule, we will see that for weights $\boldsymbol{W_{yh}}$ we only need values at the current time t, whereas for weights $\boldsymbol{W_{hh}}$ and $\boldsymbol{W_{xh}}$ we need to go back in the past. Essentially, the hidden state at the timestep t depends on all previous hidden states.

$$\frac{\partial L_3}{\partial \boldsymbol{W_{yh}}} = \frac{\partial L_3}{\partial y_3}\frac{\partial y_3}{\partial \boldsymbol{W_{yh}}} \tag{3.8}$$

As we can notice in Equation 3.9., the aforementioned hidden state dependency takes its place. Namely, $\boldsymbol{h_t}$ depends on $\boldsymbol{h_{t-1}}$ which depends on $\boldsymbol{h_{t-2}}$ and so on.

$$\frac{\partial L_3}{\partial \boldsymbol{W_{hh}}} = \frac{\partial L_t}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial \boldsymbol{W_{hh}}} \tag{3.9}$$

Because of that dependency, to take gradients for $\boldsymbol{W_{hh}}$ and $\boldsymbol{W_{xh}}$, we have to sum over all gradients over all time-steps K. The Equation 3.10., shows us all the details.

$$\frac{\partial L_3}{\partial \boldsymbol{W_{hh}}} = \sum_{k}^{K}\frac{\partial L_t}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial h_{3-k}}\frac{\partial h_{3-k}}{\partial \boldsymbol{W_{hh}}} \tag{3.10}$$

Intuitively, based on the previous equations and Figure 3.11. , we can reason that there is an issue with performing backpropagation in this form. RNNs are recognised for their capability to pass on the memory from previous inputs, and it is precisely this capability that enables RNNs to process long sequences. However, long sequences imply long chain rule dependencies for gradient computation. Consequently, the network will require a lot of time to train.

As an alternative, the researchers can utilise $truncated$ backpropagation through time (Sutskever et al., 2014). In that algorithm, the network only allows k_1 long sequences, and then, the network only allows k_2 steps of backpropagation through time, where $k_2 < k_1$.

However, even with the truncated version of backpropagation, RNNs display another significant flaw. Specifically, standard vanilla RNNs are infamous for vanishing or exploding gradient issues. RNNs, especially on longer data sequences, experience gradients that either completely vanishes (no change gets propagated backwards) or explodes (the norm of the gradient update becomes too large) (Pascanu et al., 2013). When this occurs, the network's

performance seriously deteriorates, as the network's weights - either don't change at all or they change too much.

Algorithm 3 Gradient clipping algorithm

if $||g|| > level$ **then**
  $g = \frac{level}{||g||} * g$
end if

Luckily, in the case of exploding gradient, we can utilise a simple clipping algorithm that clips the gradient value when it goes over a certain level. We can see the full algorithm in Algorithm 3. The vanishing gradient problem , unfortunately, does not have a straightforward fix. The only true way to repair all RNN's shortcomings is a different cell structure.

3.2.4. Long short-term memory networks

Long short-term memory (LSTM; Hochreiter and Schmidhuber (1997)) network is a special type of recurrent neural network. The LSTM's cell structure was introduced to counter issues of the standard vanilla RNNs. The main shortcoming of RNNs is that they actually "forget" the information over long data sequences. In practice, RNNs can only store information for 10-20 network timesteps. In other words, for sentences longer than ten to twenty words, the RNNs will not be able to remember context. In contrast, LSTMs are specifically designed to handle these long sequences.

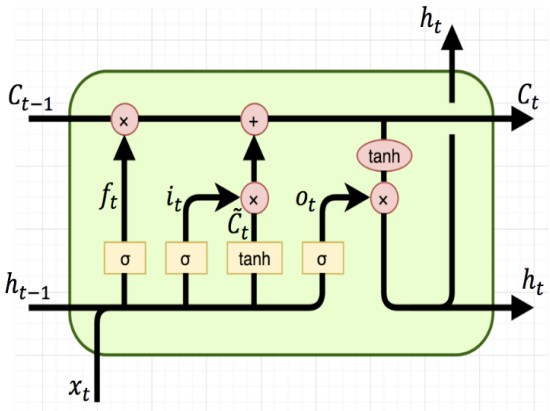

Figure 3.12: LSTM cell[3]

Figure 3.12. shows an illustration of a standard LSTM cell. We can notice, the cell's structure is much more complex. This is because the LSTM introduces certain mechanisms to prevent the loss of temporal data as the information flows through the network. The most

[3]http://colah.github.io/posts/2015-08-Understanding-LSTMs/

significant upgrade is the introduction of so-called "cell state" $\boldsymbol{C_t}$. The goal of the cell state is to store the relevant information that the network might use. The cell state runs straight down the entire chain with only minor linear interactions. This helps to prevent the loss of information due to vanishing or exploding gradients. The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.[4] Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation (Hochreiter and Schmidhuber, 1997).

The first gate (also referred as *forget* gate) $\boldsymbol{f_t}$ determines how much of the current memory from the $\boldsymbol{C_t}$ are we going to remove. The decision is based on the input $\boldsymbol{x_t}$ and the previous hidden state $\boldsymbol{h_{t-1}}$. We can see a full formula for the forget gate in Equation 3.11.

$$\boldsymbol{f_t} = \sigma(\boldsymbol{W_h} * [\boldsymbol{h_{t-1}}, \boldsymbol{x_t}] + \boldsymbol{b_f}) \tag{3.11}$$

The next step is to decide what new information we will store in the cell state. This process has two parts. First, a sigmoid layer called the *input gate layer* decides which values we'll update. Next, a $tanh$ layer creates a vector of new candidate values, $\tilde{\boldsymbol{C}}_{\boldsymbol{t}}$, that will be added to the state. In the next step, we'll combine these two and create an update to the state.

$$\begin{aligned} \boldsymbol{i_t} &= \sigma(\boldsymbol{W_i} * [\boldsymbol{h_{t-1}}, \boldsymbol{x_t}] + \boldsymbol{b_i}) \\ \tilde{\boldsymbol{C}}_{\boldsymbol{t}} &= tanh(\boldsymbol{W_C} * [\boldsymbol{h_{t-1}}, \boldsymbol{x_t}] + \boldsymbol{b_C}) \\ \boldsymbol{C_t} &= \boldsymbol{f_t} \circ \boldsymbol{C_{t-1}} + \boldsymbol{i_t} \circ \tilde{\boldsymbol{C}}_{\boldsymbol{t}} \end{aligned} \tag{3.12}$$

Finally, we have to generate an output. First, we have a sigmoid layer which decides which parts of the current input x_t and the previous hidden state h_{t-1} we're going to output. Then, we pass on the cell state through $tanh$ function to push the values to be between -1 and 1 and multiply it by the output of the sigmoid gate. We can see the details in Equation 3.13.

$$\begin{aligned} \boldsymbol{o_t} &= \sigma(\boldsymbol{W_o} * [\boldsymbol{h_{t-1}}, \boldsymbol{x_t}] + \boldsymbol{b_o}) \\ \boldsymbol{h_t} &= \boldsymbol{o_t} * tanh(\boldsymbol{C_t}) \end{aligned} \tag{3.13}$$

[4] See footnote 3

Applications of RNNs and LSTMs

RNNs were applied to a variety of different tasks. In theory, we have a defined list of distinct task formulations for which researchers can utilise RNNs. In Figure 3.13., we can see an illustration. First, we have the one-to-one formulation. Generally, one-to-one problems are solved with simpler neural architectures like FFNNs or CNNs. An example of a one-to-one problem is image classification. In image classification, both the input and the output are of fixed size.

Next, are the one-to-many tasks. The one-to-many approach is characterized by a fixed vector for input and a sequence for output. A typical example of a one-to-many task is image captioning. In image captioning, the input is a fixed vector, but the output is a variable-length text sequence (image caption).

Further, we have the many-to-one approach, a reverse version of the one-to-many task. The many-to-one approach is commonly applied to sequence classification problems such as sentiment analysis and extractive text summarization.

Lastly, we have two variants of many-to-many approaches. In both variants, both the input and the output are sequences. Commonly, this approach is referred to as the sequence-to-sequence (seq2seq) approach, which can be applied to problems such as machine translation, speech recognition and abstractive text summarization. When the length of the input sequence is equal to the size of the output sequence, we refer to this task as a sequence labelling task. A prime example of a sequence labelling task is the named-entity recognition task.

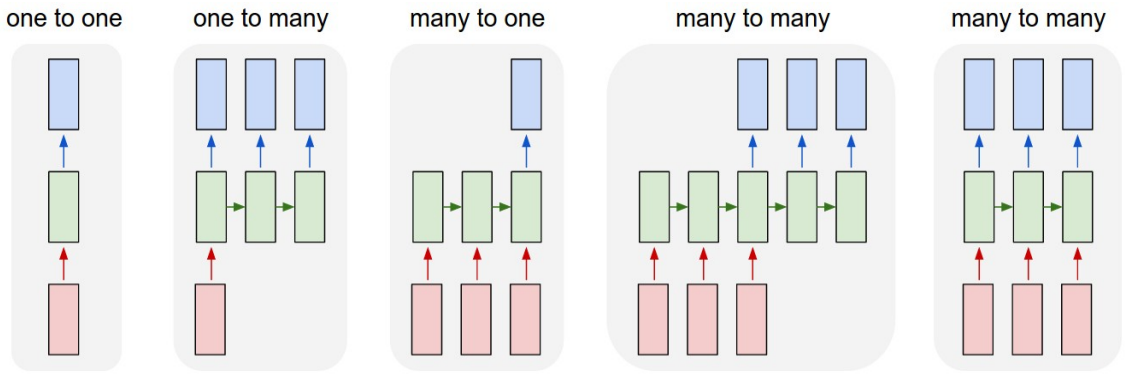

Figure 3.13: Five distinct RNN applications[5]

Encoder-decoder framework

Another significant application of RNNs (and LSTMs) is the encoder-decoder framework. The encoder-decoder framework is an architecture style that is often applied to various NLP

[5]https://towardsdatascience.com/introduction-to-recurrent-neural-network

problems. The encoder and the decoder are two individual deep neural networks and work together as one combined neural network. The encoder's task is to understand the input sequence, while the decoder's job is to determine the appropriate output based on the input sequence (Gupta and Gupta, 2019).

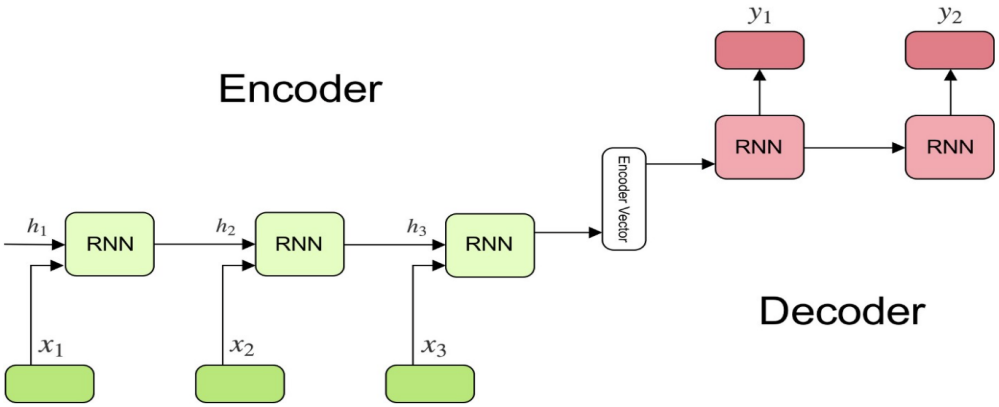

Figure 3.14: Encoder-decoder architecture[6]

A bit more formally, in encoder-decoder framework, an encoder transforms the input sequence, usually a sequence of vectors $\boldsymbol{x} = (x_1, x_2, ..., x_{T_n})$, into a *fixed*-length vector c (Sutskever et al., 2014). The most common approach is to use an RNN

$$\begin{aligned} \boldsymbol{h_t} &= f(\boldsymbol{x_t}, \boldsymbol{h_{t-1}}) \\ \boldsymbol{c} &= q(h_1, h_2, ..., h_{T_n}) \end{aligned} \tag{3.14}$$

where $\boldsymbol{h_t} \in \mathbb{R}^n$ is a hidden state at time t, and c is a vector generated from the sequence of the hidden states. f and q are some non-linear functions. The decoder is often trained to predict the next word $y_{t'}$ given the context vector c and all the previously predicted words y_1, y_2, ..., $y_{t'-1}$ (Bahdanau et al., 2014). We refer to these types of decoders as autoregressive decoders. We can also have a decoder that predicts the output given only the context vector c without the previously predicted outputs. However, in the case of autoregressive decoder, the decoder defines a probability over the output words $\boldsymbol{y}$ by decomposing the joint probability into the ordered conditionals:

$$p(\boldsymbol{y}) = \prod_{t=1}^{T} p(y_t | y_1, ..., y_{t-1}, c) \tag{3.15}$$

where $\boldsymbol{y} = (y_1, ..., y_{T_y})$. In Figure 3.14., we can see an example of encoder-decoder architecture. As we can notice, we use two separate recurrent neural networks. However, it is important to note that we could have used any artificial neural network such as feed-forward neural network, convolutional neural network and so on.

As mentioned earlier, the goal of the encoder is to create a fixed-length context vector c_i that should (ideally) capture all of the information from the input sequence. On the other hand, the job of the decoder is completely determined by the problem we want to solve.

In practice, we can apply a decoder to solve any of the tasks we have mentioned earlier. Essentially, the encoder helps to digest the input into a fixed-length vector, whereas the decoder then can do whatever it wants with the encoder's input. Generally, in the case of extractive text summarization, we use the encoder to encode text sequences (sentences) into fixed-length vectors, and then, we use the decoder to decide whether the sentence should be included in the final summary.

The Attention Mechanism

As mentioned previously, recurrent neural networks have been successfully applied in many tasks involving sequential data. However, even the more cutting-edge models like LSTMs have their limitations, and researchers still have a hard time developing high-quality models when working with long data sequences. This problem is particularly emphasised in the case of encoder-decoder architectures, where the entire input sequence is transformed into a small fixed-length vector. A root issue with the encoder-decoder framework is that the encoder has to compress all relevant information from a variable-length input sequence into a fixed-length vector. This makes it difficult for the decoder to cope with long sequences, especially those longer than the sequences in the training corpus. [7]

To address this problem, Bahdanau et al. (2014) proposed a novel architecture that employs a concept that is today known as the $attention$ mechanism. Essentially, they extend the standard encoder-decoder framework by introducing an attention layer between encoder and decoder that enables the decoder to select which part of the input it should focus on. This approach frees the model from having to encode a whole source sequence into a fixed-length vector and also lets the model focus only on information relevant to the decoder (Bahdanau et al., 2014).

In Figure 3.15., we can observe the architecture that Bahdanau et al. (2014) proposed. In their original model, the researchers based their encoder on a bidirectional RNN. However, the attention mechanism can be used with any artificial neural network.

The authors decided to use bidirectional RNN, as they wanted to capture more context of the input sequence. A unidirectional (standard vanilla) RNN can only capture left-side context. This means that each processed word with a standard RNN will not be encoded with the context from the right side of the sequence. For example, in the input sequence *"I love running a company"*, the word *"running"* has an entirely different meaning when the

[6] https://towardsdatascience.com/understanding-encoder-decoder-sequence-to-sequence-model

[7] https://medium.datadriveninvestor.com/attention-in-rnns

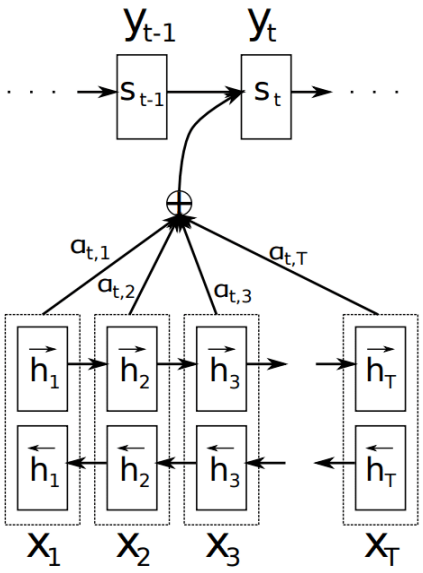

Figure 3.15: Encoder-decoder framework with attention mechanism (Bahdanau et al., 2014)

right side of the sentence is taken into consideration. The benefit of bidirectional networks is obtaining a context from both sides of the sequence (Schuster and Paliwal, 1997). Specifically, a bidirectional RNN consists of two identical RNN, where one network is processing the input from the left side and the other from the right side of the sequence. We can see the output from the bidirectional network in Equation 3.18. Effectively, the output is a concat of outputs from two independent RNN networks.

$$\boldsymbol{h_j} = [\overrightarrow{h_j}, \overleftarrow{h_j}] \tag{3.16}$$

In the attention layer, we are calculating the context vector $\vec{c_i}$. The context vector depends on the sequence of $annotations$ (h_1, h_2, ..., h_T) to which the encoder mapped the input sequence (x_1, x_2,... , x_T) (Bahdanau et al., 2014). Each annotation h_i contains information about the whole input sequence with a strong focus on the parts surrounding the i-th word of the input sequence (Bahdanau et al., 2014). The context vector c_i is computed as a weighted sum of these annotations h_i. The weights α_{ij} are calculated as a softmax output on top of the feed-forward neural network e_{ij}.

$$\boldsymbol{c_i} = \sum_{j=1}^{T} \alpha_{ij} h_j \tag{3.17}$$

$$\boldsymbol{\alpha_{ij}} = \frac{exp(e_{ij})}{\sum_{k=1}^{T_k} exp(e_{ik})} \tag{3.18}$$

$\boldsymbol{e_{ij}}$ is an $allignment$ model which scores how well the inputs around position j and the output at position i match (Bahdanau et al., 2014). The alignment model is a feed-forward neural network which is jointly trained with all the other components of the proposed

system. The probability α_{ij}, or its associated alignment output e_{ij}, reflect the importance of the annotation h_j with the respect to the previous hidden state s_{i-1} in deciding the next state s_i and generating y_i (Bahdanau et al., 2014). Essentially, this represents the attention mechanism in the decoder. The decoder decides which parts of the source sentence to pay attention to. By augmenting the decoder with an attention mechanism, we relieve the encoder from the burden of having to encode all information in the source sentence into a fixed length vector (Bahdanau et al., 2014).

3.2.5. Transformers

Recurrent neural networks, even with improvements such as attention mechanism and LSTM cell, still have one major flaw - they are exceptionally slow to train. This inherent slowness stems from the fact that RNNs (and LSTMs) are trained sequentially. At each time step t, the networks can process only one part of the current input x_t. This inherently sequential nature precludes parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching across examples (Vaswani et al., 2017). As a solution, Vaswani et al. (2017) proposed a novel architecture - the Transformer.

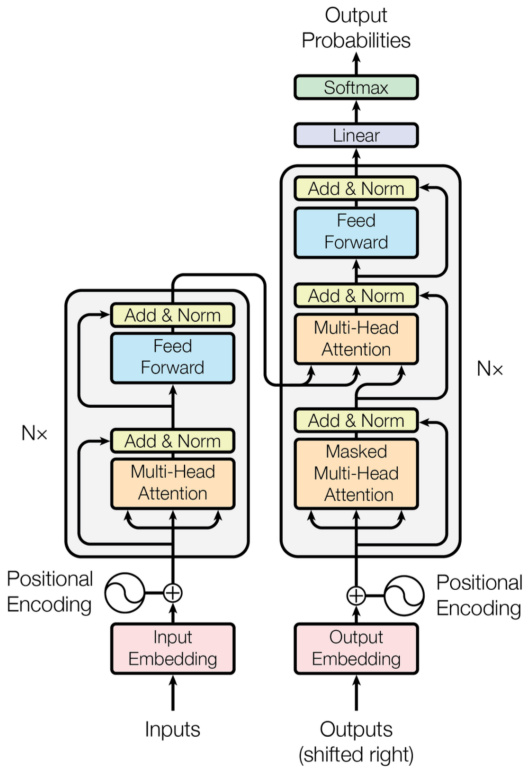

Figure 3.16: Transformer architecture (Vaswani et al., 2017)

The Transformer is a neural architecture that relies on the attention mechanism to capture

dependencies in sequential data. Similarly to RNNs, the transformer was designed to handle sequential input data. However, unlike RNNs, transformers do not process the data chronologically. Instead, transformers rely on attention mechanisms to encode the significance and context into the input sequence. Consequently, transformers do not have to process the sequence step by step, but rather they can do it in parallel. This ability enables transformers to be trained in parallel, which, in turn, allows for better utilization of GPU hardware and faster training time. The model is based on the encoder-decoder architecture that we can see in Figure 3.16.

Architecture

A standard Transformer consists of two main parts - the encoder and decoder. In Figure 3.16., we can notice that the model is split in two parts. The left part is the encoder and the right side is the decoder. The encoder maps an input sequence of symbol representations (x_1, ..., x_n) to a sequence of continuous representations $\boldsymbol{z}$ = (z_1, ..., z_n). Given $\boldsymbol{z}$, the decoder then generates an output sequence (y_1, ..., y_m) of symbols one element at a time. At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next (Vaswani et al., 2017).

Encoder

The encoder consists of $N = 6$ stacks of identical encoder layers, and each encoder layer consists of several layers. For starters, before we pass the data inside the encoder, we have some input preprocessing layers. First, we have the *Embedding* layer. Here, the transformer maps the input (e.g. text sentence) into fixed-length vectors. In the standard transformer's encoder, the input will be a list of n 512-dimensional vectors. In the first encoder block, those vectors will be the word embeddings (or any input embeddings), but in other encoder layers, they will be the output of the previous encoder. The size of this input sequence is a hyperparameter. Usually, it would be the length of the longest sentence in our training dataset. In Figure 3.17. , we can see a close-up example of embedding and positional layer.

Further, we have a *Positional Encoding* layer. As we mentioned, transformers do not process data sequentially, so it is crucial to have a way to encode the position of a word within the sentence. In RNNs, this happens inherently since the data is processed sequentially. However, in transformers, we achieve that effect by utilizing any function that encodes positions as vectors. In the original paper, the authors used *sin* and *cosin* functions. We can see the full equations for positional encoding in Equation 3.19.

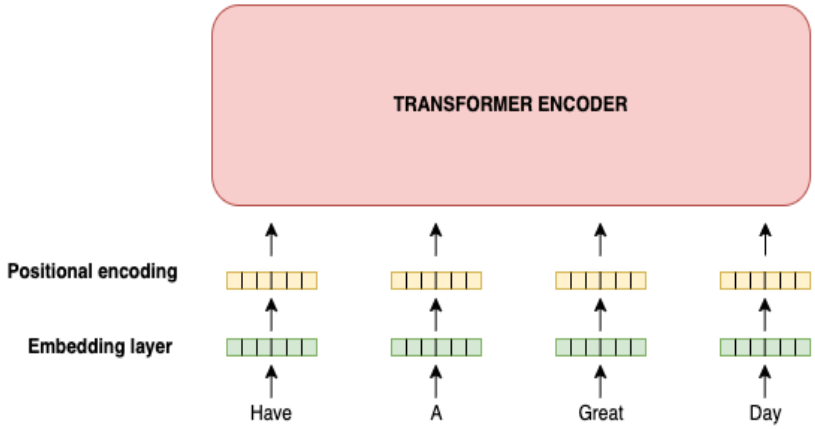

Figure 3.17: Example of embedding and positional encoding for embedding size of 6

$$\begin{aligned} \boldsymbol{PE_{(pos,2i)}} &= sin(pos/10000^{\frac{2i}{d_{model}}}) \\ \boldsymbol{PE_{(pos,2i+1)}} &= cos(pos/10000^{\frac{2i}{d_{model}}}) \end{aligned} \tag{3.19}$$

Self-Attention

After the positional and the embedding layer, vectors $\boldsymbol{c_i}$ are infused with distributional and positional information. Now, we enter the first encoder block. Here, the first important layer we encounter is the *Multi-head Attention layer*. The goal of the multi-head attention layer is to add contextual information to word vectors. For example, in the text sequence *" The policeman didn't catch the offender because he was too slow"*, what does the "he" refer to? When the model is processing the word "he", self-attention allows it to associate "he" with "policeman".

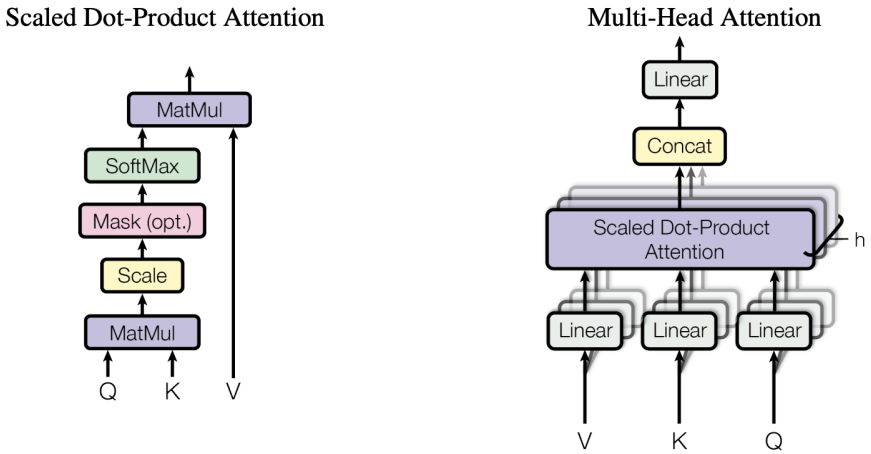

Figure 3.18: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention (Vaswani et al., 2017)

In Figure 3.18. , we illustrate the architecture of multi-head attention layer. Multi-head attention consists of **h** *scaled dot-product attention* modules. The scaled dot-product attention module determines the attention distribution. Each module contains three weights $\boldsymbol{W_q}$, $\boldsymbol{W_k}$ and $\boldsymbol{W_v}$. These weights, together with the input matrix $\boldsymbol{X}$, determine the values of $\boldsymbol{Q}$, $\boldsymbol{K}$ and $\boldsymbol{V}$.

$$\boldsymbol{Q} = \begin{pmatrix} \vec{x_1} \\ \vec{x_2} \\ \vdots \\ \vec{x_n} \end{pmatrix} \cdot \begin{pmatrix} w_{11} & \dots & \dots & w_{1x} \\ w_{21} & w_{22} & \dots & \dots \\ \vdots & \vdots & \vdots & \vdots \\ w_{n1} & \dots & \dots & w_{nx} \end{pmatrix} = \boldsymbol{X} \cdot \boldsymbol{W_q} \tag{3.20}$$

$$\boldsymbol{K} = \begin{pmatrix} \vec{x_1} \\ \vec{x_2} \\ \vdots \\ \vec{x_n} \end{pmatrix} \cdot \begin{pmatrix} w_{11} & \dots & \dots & w_{1x} \\ w_{21} & w_{22} & \dots & \dots \\ \vdots & \vdots & \vdots & \vdots \\ w_{n1} & \dots & \dots & w_{nx} \end{pmatrix} = \boldsymbol{X} \cdot \boldsymbol{W_k} \tag{3.21}$$

$$\boldsymbol{V} = \begin{pmatrix} \vec{x_1} \\ \vec{x_2} \\ \vdots \\ \vec{x_n} \end{pmatrix} \cdot \begin{pmatrix} w_{11} & \dots & \dots & w_{1x} \\ w_{21} & w_{22} & \dots & \dots \\ \vdots & \vdots & \vdots & \vdots \\ w_{n1} & \dots & \dots & w_{nx} \end{pmatrix} = \boldsymbol{X} \cdot \boldsymbol{W_v} \tag{3.22}$$

Naturally, the question arises regarding the values $\boldsymbol{Q}$, $\boldsymbol{V}$, and $\boldsymbol{K}$. What are they, and what are they used for? In short, the authors added them as an abstraction useful for calculating and thinking about attention. More formally, the *query* matrix *Q*, the *key* matrix *K* and the *value* matrix *V* are used to model the interaction between the inputs *X*. In Equation 3.23, we can see the equation that produces h attention vectors for each of the inputs in the matrix *X*. Since all of those matrices are derived from input matrix *X*, this equation is trying to model how much each input vector $\vec{x_i}$ relates to the rest of the input matrix *X*.

$$Attention(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = softmax(\frac{\boldsymbol{Q}\boldsymbol{K^T}}{\sqrt{d_k}})\boldsymbol{V} \tag{3.23}$$

Multi-head attention creates h of these attention matrices, one for each scaled dot-product attention layer. After applying the Multi-head Attention layer, all of these matrices get concatenated and passed to linear layer.

Feedforward layer

After the attention block, a standard encoder block has a fully-connected feedforward layer. This layer is applied to each input vector separately and identically. This consists of two

linear transformations with a ReLU activation in between. The first layer upscales the input, and the second layer downscales to the original dimension.

$$FFN(\vec{x}) = max(0, \vec{x}W_1 + b_1)W_2 + b_2 \tag{3.24}$$

Decoder

Similarly to the encoder, the standard Transformer decoder is comprised of *N = 6* identical blocks. Each decoder layer has several layers, most of which are the same as the ones we already mentioned for the encoder. However, there are some notable differences.

Firstly, the first attention module in the decoder is the Masked Multi-Head attention layer. This module is identical to the Multi-head attention block from the encoder, but we have to mask the unpredicted values of the output since the decoder mustn't know the whole output sequence. In other words, we have to hide the parts of the output that the decoder hasn't predicted already.

Next, we have another *Multi-head Attention* block. This attention block is used to combine attention input vectors from the encoder, with the attention output vectors from the decoder. Essentially, the transformer is connecting input to output words. For example, in the case of machine translation from English to French, this block would act as a mapping module that would try to map individual English words to the corresponding French words.

Lastly, all attention vectors are passed into another feed-forward layer. After that, the transformer passes the vectors into a final linear layer, which functions as a dimensionality correction layer. As an example, in the case of machine translation, we want to expand the dimension of an output vector to the size of French language vocabulary. This way, when we can pass that output vector through the softmax function, we will get a probability distribution for the next French word.

3.2.6. BERT

Bidirectional **E**ncoder **R**epresentations from **T**ransformers (BERT; Devlin et al. (2018)) is a transformer-based neural architecture that is often applied to various natural language processing tasks. At its core, BERT is a large version of Transformer's encoders.

Pretraining

The most significant advantage of BERT is that, in its essence, BERT is a large and powerful language model. Essentially, we can give BERT several unsupervised language tasks over a huge text corpus such as the Wikipedia dataset, with the goal of learning distributional information of natural language. We call this process the *pretraining*.

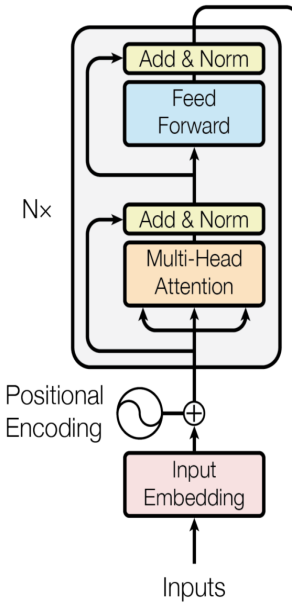

Figure 3.19: BERT architecture (Vaswani et al., 2017)

In practice, we pretrain BERT by using two distinct self-supervised tasks. First, we have a **masked token prediction** task. For this task, we randomly mask a certain percentage of words in the corpus, and it is BERT's job to guess which words are missing. The goal of this taks is learning to read contextual cues.

The second task is a **next sentence prediction** task. For this task, we input two randomly chosen sentences from the dataset, and BERT has to predict whether the second sentence is a continuation given the first sentence. This process is random, and 50% of the time, we take two sequential sentences, and the rest of the time, we pick two random sentences from the dataset. By formulating a task like this, we strive to give BERT the ability to capture dependencies between sentences. This capability was originally motivated by several downstream BERT applications such as *Question Answering* and *Natural Language Inference* that require this proficiency.

BERT's training on these two tasks is done simultaneously. In other words, BERT attempts at the same time to predict which words are missing and whether the two sentences are related. An example of this is given in Figure 3.20. As we can notice, the authors had to introduce *position* embeddings and *segment* embeddings to enable this simultaneous training. Segment embeddings allow BERT to discern the difference between two different sequences. Also, similarly to Transformer, BERT has positional encoding. With those two embeddings, BERT can determine the position and segment of each word in parallel (as opposed to RNNs).

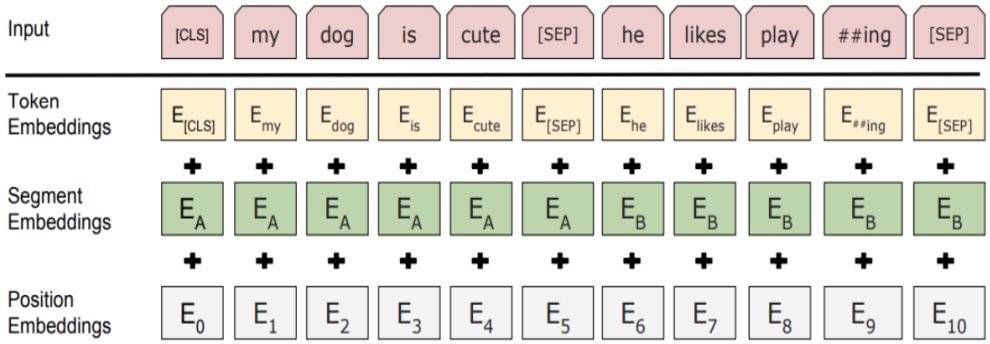

Figure 3.20: The input representation for BERT (Devlin et al., 2018)

Fine-tuning

After long pre-training phase, we can use BERT for numerous downstream tasks. Essentially, pre-trained BERT understand natural language. Now, we can use BERT as an embedding layer for different supervised tasks (Devlin et al., 2018).

As an example, we can use BERT for a sentiment analysis task. In sentiment analysis, the goal is to correctly predict the polarity of a text sequence (usually a sentence or a phrase). A bit more formally, for a sequence $\vec{s_i} = w_1, ..., w_n$ with n words, the goal is to correctly predict y_i ($y_i \in \{0, 1\}$), where $y_i = 1$ means that the sentiment of the sentence is positive.

Usually, the sentiment analysis task would require an encoder that encodes the text sequence into a fixed vector, and a decoder that predicts whether the sequence has positive or negative sentiment. However, now, we can just employ BERT to embed the whole input sequence and add a softmax layer on top of BERT, which would predict whether the input is positive or negative. Furthermore, now, we can train our whole network (BERT + softmax layer) on the entire sentiment analysis dataset and essentially fine-tune the pre-trained BERT specifically for the sentiment analysis task.

However, the applications do not stop on sentiment analysis. BERT has been applied on numerous NLP tasks such as Question Answering, Single Sentence Classification and even Extractive Text Summarization (Liu and Lapata, 2019).

4. Model

4.1. Problem Overview

As we mentioned in the previous chapters, extractive summarization can be categorized as a sequence classification task. Generally, most neural extractive models can be described with three modules - (1) *the sentence encoder*, (2) *the document encoder* and (3) *the decoder*. In practice, researchers introduced numerous improvements for this approach by adding newer architectures, attention mechanisms or merely increasing the number of layers (Zhong et al., 2019). However, most of the approaches share a common problem - **high redundancy of information in summaries**.

Figure 4.1: Example of high redundancy

In a general neural extractive approach, we first use the sentence encoder to encode each sentence s_i into a embedded fixed-length vector d_i. Then, we want to contextualise the

embeddings, so we use the document encoder to encode context into each sentential representation d_i. Lastly, the decoder goes through these contextualized sentence representations and makes an independent binary decision whether the sentence should be included in the summary. The core issue is that the model makes an individual decision for each sentence, which doesn't prevent the selection of sentences with similar content.

For example, in Figure 4.1., we can see how the model might select two sentences that express the same information, just because they are the most salient ones. This example demonstrates the essence of the problem - most of the current neural models focus on independent **sentence-level** decisions, which, in turn, makes them more inclined to select highly generalized sentences while ignoring the coupling of multiple sentences (Zhong et al., 2020).

In Figure 4.1. , we can see that the information density of the summary could have easily been improved by coupling the first sentence with the second sentence, but on the individual sentence level, the second sentence does not carry enough information to be picked by the decoder.

4.1.1. Possible solutions

Autoregressive decoder Naturally, researchers have been trying to find a solution for this problem. The easiest solution employed is an autoregressive decoder (Jadhav and Rajan (2018); Zhou et al. (2018)) . The autoregressive decoder takes previous outputs in consideration. The idea is that if the decoder "knows" which sentences are in the summary, then it won't choose the sentences with similar content.

Trigram Blocking Another novel idea was the introduction of trigram blocking (Liu and Lapata, 2019), a simple technique for precluding redundancy in summaries. Essentially, the idea is that the decoder shouldn't select sentences that have trigram overlaps with already previously picked sentences. It is a relatively simple heuristic, that surprisingly brought remarkable performance improvements (Zhong et al., 2020).

4.2. MatchSum

As we noted, current extractive approaches are focused on individually selecting salient sentences, which often results in high redundancy of information. We mentioned two possible solutions - an autoregressive decoder or trigram blocking. However, the root of the issue goes deeper than what those fixes can do. Even with those fixes, we still frame extractive summarization as a sentence-level task, which forces the model to make individual sentence-level choices. An improvement would be to formulate extractive summarization as - a search

for optimal candidate *summary*. Effectively, we should change the task from searching the most salient sentences to the most salient summaries.

The main idea of this thesis and also the core concept behind Zhong et al. (2020) is the introduction of a novel **summary-level** framework named **Matchsum** and conceptualization of extractive summarization as a **semantic text matching** task.

4.2.1. Semantic Text Matching

Computers do not understand words or strings. For the computer to comprehend textual data, we have to preprocess the text and assign it some numerical value. In practice, we have many different ways we can do that, from simple vocabulary enumeration to complex contextualized word vectors.

For starters, the simplest method, as we mentioned, is just enumerating the whole language vocabulary. For example, the vocabulary for English language could look something like $\boldsymbol{V}$ = ($the \rightarrow 0$, $dog \rightarrow 1$, $cat \rightarrow 2$, ..., $end \rightarrow 100.000$). This way we can translate a simple sentence "$A\ dog\ sits\ on\ the\ bench$" into a vector $\vec{s_i}$ = $[25, 1, 435, 340, 0, 4442]$. However, we can quickly notice that this approach isn't ideal. Primarily, the most significant issue is that the vectors aren't the same lengths for different sequences. We can solve that by encoding a sentence as a combination of certain words. For example, we make vector's length to be equal to the size of the whole vocabulary, $|\vec{s_i}| = |\boldsymbol{V}|$. Now, for each sentence we set the vector's value to 1, only if the sentence contains the word with that specific index. For example, the sentence "$A\ dog\ sits\ on\ the\ bench$" would be a vector $\vec{s_i}$ = $[1, 1, 0, 0, ..., 0]$.

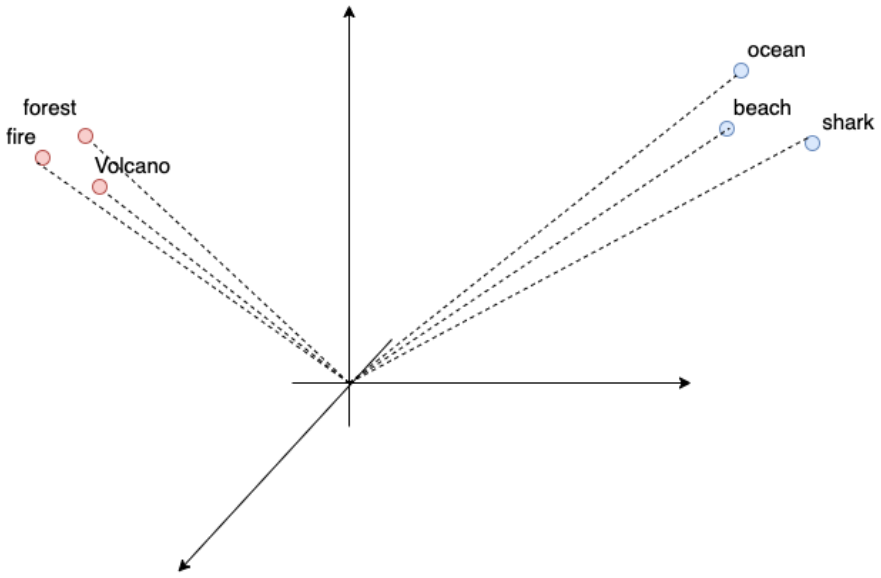

Figure 4.2: Word embeddings in 3D space

Instinctively, we can notice that this approach also isn't optimal. Namely, we have to utilise large and sparse vectors to encode even the shortest sequences. As an example, for a simple one-word sentence, we would have to use a 100.000-dimensional vector that would mostly be empty. This issue would occur with most sequences, and we would be left with large amounts of unused memory. Lastly, an additional problem is that this way of encoding doesn't provide a meaningful way of measuring similarity among sequences. In other words, we can't compare sequences and calculate their syntactic or semantic similarity.

Context-independent word embeddings

Fortunately, there is a way to encode information as dense fixed-length vectors, which can, in turn, be used to measure similarity among sequences. Word embedding is a term used for the representation of words for text analysis, typically in the form of a real-valued vector. These vectors encode syntactic and semantic similarity such that the words that are closer in the vector space are expected to be similar in meaning (Mikolov et al., 2013). The first benefit of word embeddings is that each word gets its corresponding dense vector, whose length can be arbitrarily long. Further, these word embeddings, allow us to express relations between words. For example, now, we can define the following equation:

$$\vec{man} - \vec{woman} + \vec{queen} \approx \vec{king} \tag{4.1}$$

In the expression above, we model the relation between four word-embeddings - *man, woman, queen and king*. As we can see, with these word embeddings, we can model transitive relations between multiple agents. Effectively, the expression implies that the relation between a *man* and a *woman* is approximately equal to the relation between a *king* and a *queen*.

In Figure 4.2., we can also see how words are grouped in the 3-dimensional space. This means that text sequences that are semantically similar in natural language will be closer in this semantic text space. In our example, words $ocean$, $beach$ and $shark$ are close. We can say that these words are often used within the same context.

Another significant advantage of word embeddings is their accessibility and variety. Generally, we can acquire these word embeddings by adding *Embedding layer* to our neural model, which will learn word embeddings for all words in the training dataset. But, we can also use any of the already pre-trained word representations such as GloVe[1] or Word2vec (Mikolov et al., 2013). These corpora of word representations come in a forms of large dictionaries, where each word has a corresponding d-dimensional dense vector representation.

[1] https://nlp.stanford.edu/projects/glove/

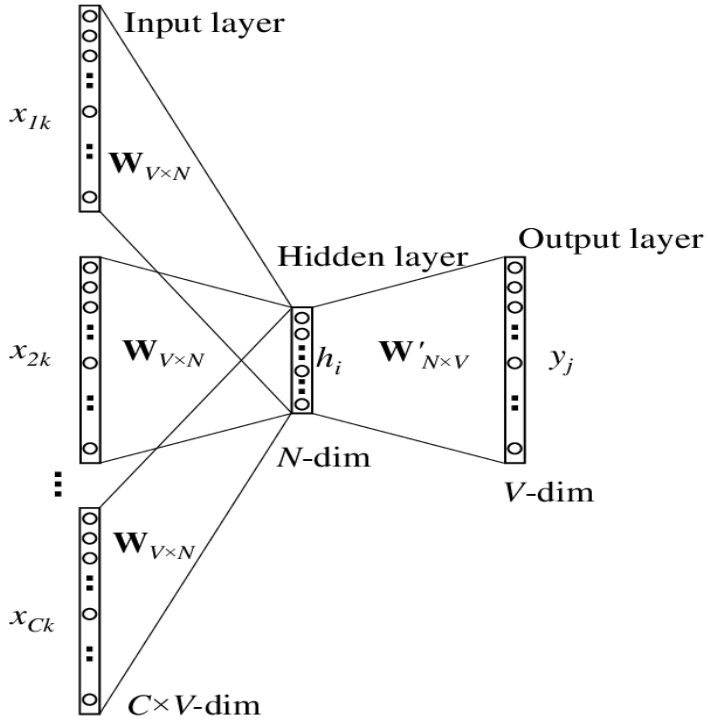

Figure 4.3: Continuous bag of words (CBOW) (Mikolov et al., 2013)

To generate these pre-trained word representations, authors leveraged shallow neural networks specifically trained to learn word embeddings. They do it by forcing the model to understand the context. One of the popular ways to do it is by using a continuous bag of words (CBOW). In the CBOW technique, the network receives a sentence (represented as a sequence of words) with one word removed as an input, and as an output, the network has to predict which word is missing. In the case of word2vec pre-trained vectors, authors employed a one-layer neural network and CBOW technique to acquire word embeddings for all words in several languages (Mikolov et al., 2013).

In Figure 4.3. , we can see a shallow neural network used to acquire word embeddings by using a CBOW technique. As we mentioned, the goal for this network is to guess which word y_i is missing from the sentence (x_{1k}, x_{2k}, ..., x_{Ck}). This way the network has to learn how the words relate to each other and which words are used in similar contexts. The hidden layers of these networks contain word embeddings for specific words.

However, these shallow word embeddings have their limitations. One of the main limitations is that words with multiple meanings are conflated into a single representation.[2] In other words, polysemy and homonymy are not handled properly. For example, in the sentence *"Pass me the bat!"*, it is not clear if the term bat is referring to a *bat (an animal)*, *bat (a baseball bat)* or any other sense that bat might have. Because of this, these shallow

[2]en.wikipedia.org/wiki/Word_embedding

neural-based word embeddings are often referred to as *context-free* or *context-independent* word embeddings.

Contextualized word embeddings

Fortunately, with recent developments of large and powerful neural architectures such as transformers, the context infused embeddings have been developed. These embeddings use a word's context to disambiguate polysemes. BERT, described in the previous chapter, takes into account the context for each occurrence of a given word.

As an example, whereas the vector for *"running"* would have the same *word2vec* vector representation for both of its occurrences in the sentences *"He is running a company"* and *"He is running a marathon"*, BERT, on the other hand, will provide a contextualized embedding that will be different according to the sentence.

4.2.2. Summary-level Approach

The main goal of this novel architecture is to raise the extractive summarization from a sentence-level task to a summary-level task. To achieve that, we have to completely disregard the standard way of doing extractive summarization.

For starters, if we want to frame it as a summary-level task, the model has to decide based on the candidate summary instead of making the decision based on individual sentences. One possible approach is to **generate candidate summaries** and then search for the one that scores highest on evaluation metrics. Essentially, we employ a *two-stage summarization*, where in the first stage, we generate the candidate summaries from the original document, and then in the second stage we search for the most optimal candidate. The first stage is focused on generating summaries, and the second stage is a standard machine learning task, where we want to teach a network to find the most optimal summary from a pool of generated ones. We can see an illustration of the summarization process in Figure 4.4.

Generating candidate summaries

The most important question about the generation of candidate summaries is how do we generate summaries that are already high-quality and not just a random selection of sentences?

Instinctively, we consider that we could just generate all possible sentence combinations and that way we are sure that we have covered all possible candidate options, including high-quality ones. However, this approach can create a massive amount of data samples extremely fast. As an example, let's assume that a standard summary has only three sentences. Also, let's assume that a standard document in this dataset contains only 20 sentences. By applying our reasoning and generating all possible summary combinations, we would have to generate

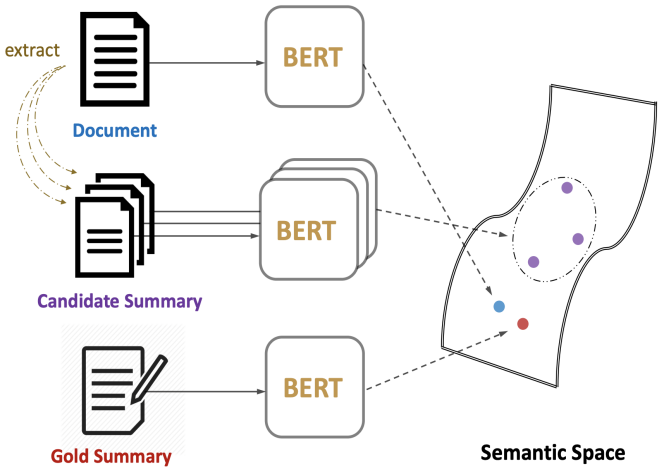

Figure 4.4: MatchSum text matching concept (Zhong et al., 2020)

1140 candidate summaries per document! Obviously, this approach wouldn't scale, and we have to find a way to prune this candidate generation.

The solution proposed by Zhong et al. (2020) is to leverage another state-of-the-art sentence-level model to select top n most salient sentences and then use just these salient sentences to generate candidates. In their paper, researchers utilised the last state-of-the-art sentence-level model - *BertExt* Liu and Lapata (2019). Liu and Lapata (2019) model is a standard neural extractive model that follows the already established sentence-level framework. In short, they utilise BERT as a sentence encoder, and then, they use those sentential representations as input to a standard classifier that determines whether to or not to include the sentence.

The generation of summary candidates follows the combinational function $O(\frac{ext!}{sel!(ext-sel)!})$, where ext is the number of salient sentences, and sel is the number of sentences we want in the candidate summary. Specifically, for every n salient sentences and k sentences we want in generated summaries, we will have to generate $\binom{n}{k}$ summaries. For example, if we choose the six most salient sentences, and each summary would consist only of three sentences, we would have to generate a total of 20 candidate summaries.

Ultimately, this generative phase of the model is flexible. The number of extracted salient sentences, the number of sentences per summary and even the type of model we use to select these salient sentences are all hyperparameters we can adjust and see which values and models work the best. However, this approach does raise an unusual prospect on summary-level extractive summarization. In our first idea, where we generated every possible candidate summary, we would certainly generate the optimal summary. And, in our pruned version, we might miss it because we are limited by ability of our sentence-level selection model. Essentially, the generation of these candidates creates a new space where improvement of

sentence-level models can actually improve the performance of summary-level models.

Semantic Text Matching

The core idea of this novel architecture is that a more salient summary should be semantically more similar to the original document (Zhong et al., 2020). This means that after we have generated all the candidate summaries, the goal is to find which summary is the most similar to the original document. For this task,Zhong et al. (2020) employed a Siamese-BERT architecture that serves as a summary (document) encoder. We can see the full architecture in Figure 4.5.

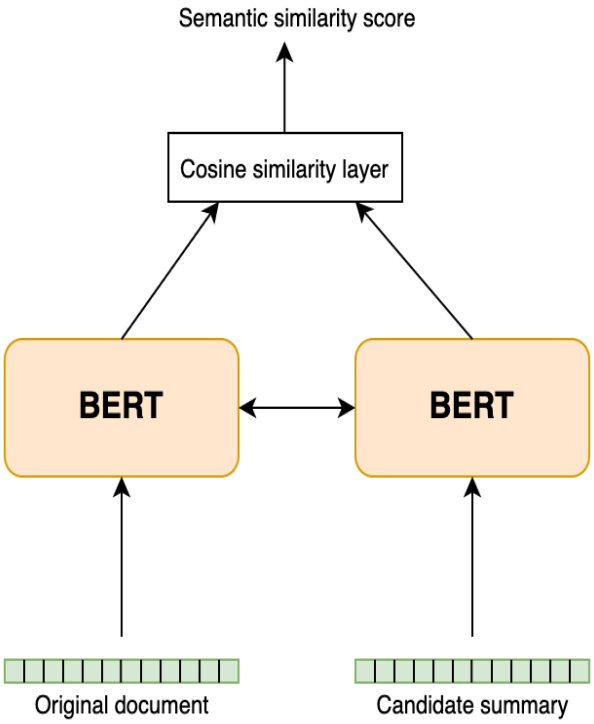

Figure 4.5: MatchSum Text Matching architecture

As we can see, the model is based on the Siamese-BERT architecture with tied weights. This Siamese-BERT model simultaneously functions as a summary and document encoder. In the original paper, the authors used a $base$ version of BERT without changing any of the input layers. Essentially, they used out-of-the-box version of BERT to to derive semantically meaningful embeddings of document D and summary candidate C. Let $\boldsymbol{e_D}$ and $\boldsymbol{e_C}$ denote the embeddings of the document D and candidate summary C. Their similarity score is measured by $f(D, C) = cosine(\boldsymbol{e_D}, \boldsymbol{e_C})$, where $cosine$ is *cosine similarity*, defined as

$$cosine(\boldsymbol{A}, \boldsymbol{B}) = \frac{\boldsymbol{A} \cdot \boldsymbol{B}}{||\boldsymbol{A}||||\boldsymbol{B}||} \tag{4.2}$$

Cosine function is one of the functions that we can employ on word embeddings to measure their semantic similarity. In order to fine-tune Siamese-BERT, we use a margin-based triplet loss to update the weights. The first assumption is that logically the gold summary C^* should be semantically closest to the source document. This assumption is implemented in the first part of the loss function:

$$\mathscr{L}_1 = max(0, f(\boldsymbol{D}, \boldsymbol{C}) - f(\boldsymbol{D}, \boldsymbol{C^*}) + \gamma_1) \tag{4.3}$$

where C is a candidate summary from document D, and γ_1 is a margin value. Besides, they also design a pairwise margin loss for all the candidate summaries. We sort candidate summaries in descending order of ROUGE scores with respect to the gold summary. Naturally, the candidate pair with a larger ranking gap should have a larger margin, which is the second principle to design loss function:

$$\mathscr{L}_2 = max(0, f(\boldsymbol{D}, \boldsymbol{C_j}) - f(\boldsymbol{D}, \boldsymbol{C_i}) + (j - i) * \gamma_2) \tag{4.4}$$

where C_i represents a candidate summary ranked i-th and γ_2 is a hyperparameter to distinguish between good and bad candidate summaries. Finally, the margin-based triplet loss can be written as:

$$\mathscr{L} = \mathscr{L}_1 + \mathscr{L}_2 \tag{4.5}$$

The basic idea is to let the gold summary have the highest matching score, and at the same time, a better candidate summary should obtain a higher score compared with the unqualified candidate summary (Zhong et al., 2020).

4.2.3. Implementation

The original source code from the authors has been published publicly and is available for use. However, although the original code provides valuable insights on certain aspects, it is not written and documented in English. Additionally, the code depends on multiple external libraries that are also not written and documented in English. Consequently, I haven't used their code and the entirety of my code was written from scratch. My implementation is split into two modules - preprocessing and summary generation module and Siamese-BERT fine-tuning module.

Candidate Summary Generation Module

As we mentioned earlier, the pruned candidate generation process requires a different state-of-the-art sentence-level model to locate the top n most salient sentences from each data

sample. In the context of this thesis, we relied on the recommended $BertExt$ BERT-based neural model from Liu and Lapata (2019). This process doesn't rely fall in within the scope of the original source code, as researchers already preprocessed several dataset this way. This preprocessing includes finding and locating indices for top n most salient sentences. After we index the original dataset, we are left with two data files - *indexes.json*, which contains a list of top n sentence indexes for each document and *original.json*, which contains documents and their corresponding gold summaries. We can see the example of this process in Figure 4.6.

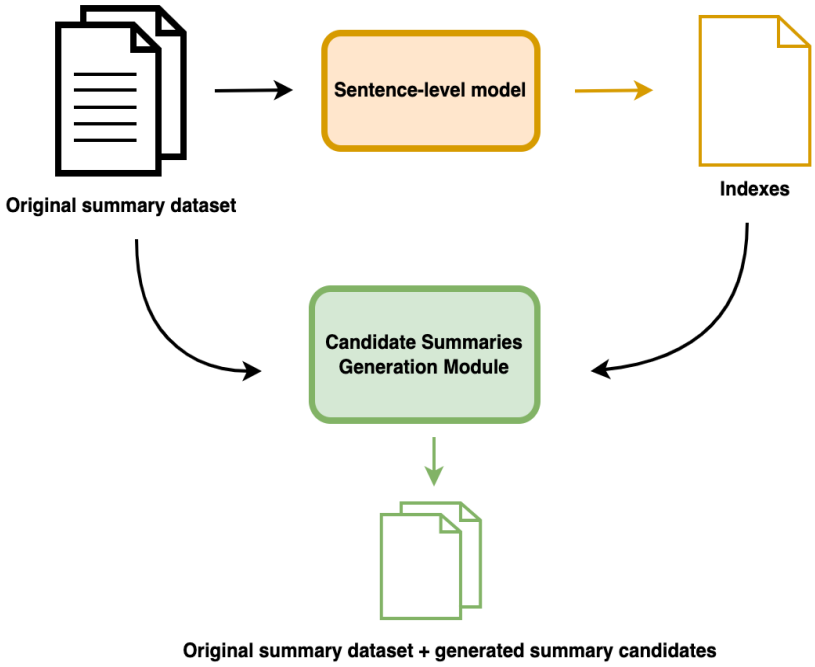

Figure 4.6: Generation of summary candidates

As we can notice in Figure 4.6., the candidate generation module uses both the original summary dataset file and the preprocessed index file. Essentially, inside the module, we go through each document and extract the sentences with indices we find in the *indexes.json* file. Then, based on the number of sentences we are going to use in the each summary, we generate candidates based on formula $\binom{ext}{sel}$. Furthermore, during this generation process each document is scored based on the ROUGE score(Lin, 2004) between the candidate and gold summary.

As an example, we have a document for which the sentence-level model has determined that the first $n = 4$ most salient sentences have indices $[0, 3, 5, 6]$. We also define that each of our generated summaries will have $k = 2$ sentences. We generate $\binom{4}{2} = \frac{4*3}{2*1} = 6$ index combinations - $[0, 3]$, $[0, 5]$, $[0, 6]$, $[3, 5]$, $[3, 6]$, $[5, 6]$. Now, we extract those $n = 4$ sentences and we generate six summaries with those index combinations.

In the context of this thesis, the number of salient sentences used was $ext = 5$. The

number of selected sentences, on the other hand, is determined by the dataset and the average number of sentences in gold summaries. For the *CNN/Daily Mail* dataset, the number of sentences in gold summaries is between two and three. Based on that, the number of selected sentences was set to both two and three, thus I would generate $\binom{ext=5}{sel=3}$ + $\binom{ext=5}{sel=2}$ = 20 for each data sample in the *CNN/Daily Mail* dataset.

BERT Fine-tuning Module

In the BERT module, the architecture in Figure 4.5 is implemented. For the implementation of the BERT and RoBERT encoders, I have used *transformers* library. [3] For both encoders, *base-uncased* versions with 110 million parameters were used. Further, for model implementation and all machine learning support, *PyTorch 1.10* was used. [4] The project in its entirety was written in *Python 3.5*. [5]

A deviation from the original model was in the choice of the optimization procedure. In the original paper, the authors used the *Adaptive Moment Estimation (Adam)* optimization procedure. However, for a reason that was not apparent, my implementation was not performing well with the Adam optimizer. Essentially, the training was stagnating, and the loss function was decreasing. Replacing replaced Adam with *stochastic gradient descent (SGD)* caused the training to become smooth. A clear reason why Adam was underperforming was not determined, but related work claims that adaptive optimization methods such as Adam generalize substantially worse than simpler methods like stochastic gradient descent (SGD) (Hardt et al., 2016).

The training was done on two *NVIDIA GeForce GTX 1080* GPUs. However, as BERT and RoBERT are huge neural networks, even with two powerful GPUs, the training time on a full dataset required on average of 4.5 days. Furthermore, due to large memory consumption, the batch size was limited to 2.

[3] https://huggingface.co/docs/transformers/index

[4] https://pytorch.org/

[5] https://www.python.org

5. Dataset

The commonly used dataset in extractive text summarization is the **CNN/ Daily Mail** (Hermann et al., 2015) dataset. *CNN/ Daily Mail* as a dataset was originally introduced by Hermann et al. (2015), but then later modified by Nallapati et al. (2016). The dataset is comprised of news articles and associated highlights as summaries. Apart from this dataset, in the original paper, the researchers have experimented on several other datasets such as *PubMed*, *XSum*, *WikiHow* and *Reddit*. However, in scope of this thesis, because of the long model training time, the main and only dataset used for evaluation is the CNN/Daily Mail dataset.

Table 5.1: The CNN/Daily Mail dataset overview

Source	**News articles**
Type	**Single Document Summarization**
Training set	**287,084**
Validation set	**13,367**
Testing set	**11,489**
Average # of tokens in the document	**766.1**
Average # of tokens in the summary	**58.2**
Average # number of sentences in the summary	**3.75**

5.1. Dataset Analysis

Earlier, we claimed that summary-level should outperform a more greedy-like sentence-level summarization. However, this hypothesis is only based only on weaknesses of previous sentence-level models. In this section, we will analyse the CNN/ Daily News dataset and additionally demonstrate the claim of the inherent gap between sentence and summary level models.

Formally, we can define $D = \{s_1, s_2, ..., s_n\}$, where D is a single document with n sentences, and $C = \{s_1, s_2, ..., s_k | s_i \in D\}$ as a candidate summary including k $(k \leq n)$ sen-

tences extracted from a document. Given a document D with its gold summary C^*, we measure a candidate summary C by calculating the ROUGE value between C and C^* in two levels - **sentence-level** and **summary-level** score (Zhong et al., 2020).

$$g^{sen}(\boldsymbol{C}) = \frac{1}{|\boldsymbol{C}|} \sum_{s \in \boldsymbol{C}} R(s, C^*) \tag{5.1}$$

In Equation 5.1., we can see an expression for sentence-level score. We define it as an average overlap between each sentence in C and the gold summary C^* (Zhong et al., 2020). s represents a sentence in candidate summary C, and $|C|$ represents the number of sentences. $R(\cdot)$ represent the mean F_1 score of ROUGE-1, ROUGE-2 and ROUGE-L.

$$g^{sum}(\boldsymbol{C}) = R(\boldsymbol{C}, C^*) \tag{5.2}$$

Equation 5.2. represents a summary-level score, where $g^{sum}(C)$ considers sentences in C as whole and then calculate the ROUGE score with the gold summary C^*.

We want to prove that the summary-level approach is more advantageous than the standard sentence-level approach. To prove that, Zhong et al. (2020) introduce **Pearl summary**. Pearl summary is a candidate summary whose *summary-level* g^{sum} score is higher than *sentence-level* g^{sen} score. Formally, we define pearl summary as candidate summary C if there exists another candidate summary $C^{'}$ that satisfies the inequality: $g^{sen}(C^{'}) > g^{sen}(C)$ while $g^{sum}(C^{'}) < g^{sum}(C)$. In other words, pearl summaries are the summaries that would benefit from the summary-level extractive model. Further, the authors also defined the term **Best summary**. The Best summary is defined as a candidate summary that has a maximum $g^{sum}(C)$ score of all candidates.

The goal of the analysis is to demonstrate that if we use sentence-level summarization, we might overlook a substantial proportion of best summaries. To start, for each data sample, we define a list of candidate summaries. Then, we take sentence-level scores for each candidate and sort them in descending order. Lastly, we define rank z as the rank of the best summary.

1. If $z = 1$, it implies that the best summary has the best sentence-level and summary-level score. This means that this summary can be found even with the sentence-level model.

2. If $z > 1$, it implies that the best summary is not detected by the sentence-level model. Moreover, as the z increases, it means that it gets harder for the sentence-level model to find the best summary.

As we can notice in Figures 5.1. and 5.2., the CNN/Daily Mail dataset is not well-suited for sentence-level extractors. Specifically, we can notice in Figure 5.1. that almost 80% of best summary-level summaries were not recognized by the sentence-level extractors as the

best summary. This means that 80% of the time, the sentence-level model will not find the best summary.

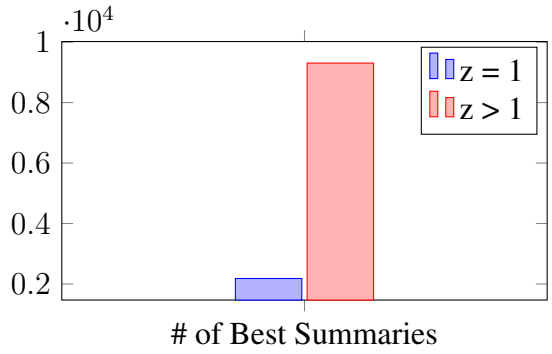

Figure 5.1: Ratio between best summaries that were found be by sentence-level model and the number of pearl summaries

This issue gets more highlighted as the rank of the best summary starts to increase. In Figure 5.2., we can see the distribution of ranks for the CNN/Daily Mail dataset. As we can notice, only 2180 best summaries were identified as the best summary purely based on the sentence-level score.

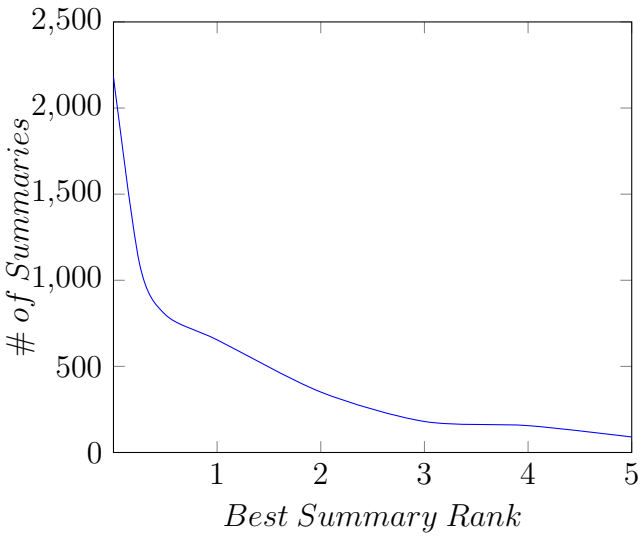

Figure 5.2: The distribution of best summary rank for the CNN/Daily Mail dataset

Naturally, these graphs highlight the absolute difference between the two approaches. Sentence-level extractors do apply some strategies to mitigate problems of redundancy, and summary-level models are still in their early stages of development. However, based on these insights plus the additional research done by the original authors, we can see that summary-level approaches offer a lot of room for progress.

6. Results

6.1. Evaluation metrics

Text summarization evaluation is a broad and elaborate domain. The fact that makes things a bit more complex is that there is no natural upper bound on the quality of summarization systems, and even humans are excluded from performing optimal summarization. In Figure 6.1., we can observe the various sub-fields of evaluation techniques for text summarization.

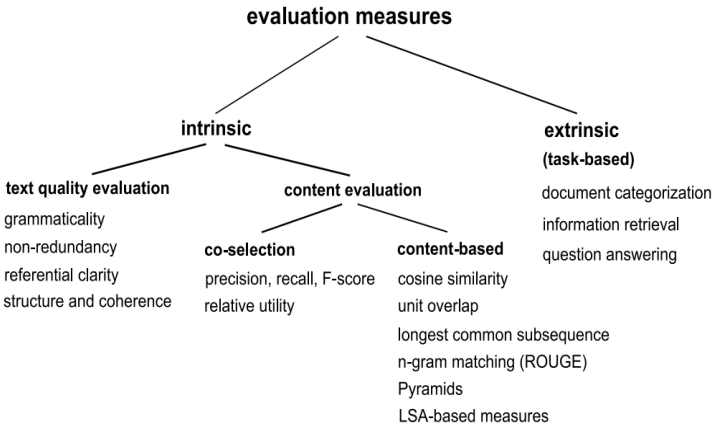

Figure 6.1: The taxonomy of summary evaluation measures (Steinberger et al., 2009)

As we can notice, text evaluation methods are divided between *intrisic* and *extrinsic* measures. Extrinsic measures are primarily suited for the tasks that use summaries for later downstream tasks such as question answering or document categorization (Steinberger et al., 2009). Since in the context of this thesis we focus exclusively on single-document, generic and extractive summarization, we will concentrate only on intrinsic evaluation measures.

Intrinsic evaluation methods are focused on evaluation *inherent* qualities of text. In the case of extractive text summarization, we deal with the pre-written and presumably grammatically and semantically correct text. Consequently, we only care how the contents of predicted summaries compare with the gold summaries. In other words, we will only concentrate on content-based measures that focus on measuring the overlap of content between summaries. Their advantage is that we can compare human-made with automatically extracted summaries.

6.1.1. ROUGE

The most prevalent intrinsic content-based evaluation metric for text summarization is ROUGE (Lin, 2004). ROUGE or Recall-Oriented Understudy for Gisting Evaluation is by far the most popular automatic method for evaluating the content of a summary. ROUGE includes measures to automatically determine the quality of a summary by comparing it to other (ideal) summaries created by humans (Lin, 2004). The measures count the number of overlapping units such as n-gram, word sequences, and word pairs between the computer-generated summary to be evaluated and the ideal summaries created by humans. ROUGE measures include ROUGE-N, ROUGE-L, ROUGE-W, and ROUGE-S measure.

In practice, commonly, text summarization relies only on two measures - ROUGE-N and ROUGE-L. Formally, ROUGE-N is a measure of *n*-gram recall between a candidate summary and a set of reference summaries (Lin, 2004). As an example, ROUGE-n = 40% means that 40% of n-grams in the $reference$ summary are also present in the $candidate$ summary. In extractive summarization, a reference summary is often just one human-written summary of the original document. The ROUGE-n score of a candidate summary is computed as follows:

$$ROGUE - n = \frac{\sum_{C \in RSS} \sum_{gram_n \in C} Count_{match}(gram_n)}{\sum_{C \in RSS} \sum_{gram_n \in C} Count(gram_n)} \tag{6.1}$$

where the RSS is a reference summary set and $Count_{match}(gram_n)$ is the maximum number of *n*-grams co-occurring in a candidate summary and a set of reference summaries. Although the length of an *n*-gram can be arbitrary, the researchers in text summarization typically employ only ROUGE-1 and ROUGE-2. Essentially, ROUGE-1 measures the overlapping of individual words, whereas ROUGE-2 measures the overlapping of bi-grams.

Apart from ROUGE-N, researchers also use the ROUGE-L measure. ROUGE-L is a measure of the longest common subsequence. Formally, a sequence $Z = [z_1, z_2, ..., z_n]$ is a subsequence of another subsequence $X = [x_1, x_2, ..., x_n]$, if there exists a strict increasing sequence $[i_1, i_2, ..., i_k]$ of indices of X such that for all $j = 1, 2, ..., k$, we have $x_{ij} = z_j$ (Lin, 2004). Given two sequences X and Y, the longest common subsequence is a common subsequence with maximum length.

6.2. Experimental results

In this section, we will go over the experimental results. In Table 6.1., we can observe how the implemented model compares with the original and the rest of the state-of-the-art models.

Firstly, we can notice that the implemented model performs relatively well compared to the previous sentence-level state-of-the-art models such as BERT-based *BertExt* and RNN-based *NeuSum*. However, we can also see that the implemented model didn't surpass the original author's implementations. These results were expected since their model was trained on much more powerful machines, which enabled multiple runs with different hyperparameters and longer training time.

The main issue that limited the ability of a much deeper model optimisation and analysis was an incredibly long training time. As mentioned earlier, the training time was between four to five days on the entire dataset. This limitation prevented multiple full-dataset tests and forced hyper-parameter exploration on a much smaller scale. Another potential issue might have been the lack of memory. Namely, the batch size had to be reduced to the size of two. This reduction on its own probably didn't impact a lot, but some research shows that larger batch size can improve convergence and generalization of the model.

Ultimately, based on initial theoretical implications, later dataset analysis and these practical results, we can conclude that the summary-level approach is a promising research direction in extractive summarization.

Table 6.1: The CNN/Daily Mail dataset overview

Model	**R-1**	**R-2**	**R-L**
LEAD	40.43	17.62	36.67
ORACLE	52.59	31.23	48.87
SUMMARUNNER (Nallapati et al., 2017)	39.60	16.21	35.3
SWAPNET (Jadhav and Rajan, 2018)	41.60	18.30	37.70
NEUSUM (Zhou et al., 2018)	41.59	19.01	37.98
PNBERT+RL (Zhong et al., 2019)	42.69	19.60	38.85
BERTEXT (Liu, 2019)	42.57	19.96	39.04
BERTEXT + Tri-Blocking	43.23	20.22	39.60
MATCHSUM + BERT (Zhong et al., 2020)	44.22	20.62	40.38
MATCHSUM + RoBERT (Zhong et al., 2020)	**44.41**	**20.86**	**40.55**
MATCHSUM + BERT (My implementation)	43.81	20.36	40.04
MATCHSUM + RoBERT (My implementation)	44.01	20.63	40.44

6.3. Future work

MatchSum is a two-stage summarization architecture. During this thesis, I tried to identify the weak points of the architecture and potentially improve the results.

The second stage of the architecture is a fine-tuning of a large Transformer-based encoder that performs semantic text matching. In this stage, the apparent improvement is increasing the size of the encoder. Initially, we used the BERT-base version with only 110 million parameters. However, personally, this wasn't an interesting direction of research, since there isn't any creativity involved.

On the other hand, the summary generation stage offers multiple areas for improvement. As mentioned earlier, there is a gap between generating all possible candidates and pruning the candidates with state-of-the-art models. When we prune candidates, we accept the fact we might lose a few best summary candidates. In light of that, one of the potential areas is to find a model that could improve the selection of sentences. However, the original authors already applied the best state-of-the-art sentence-level summarization model.

Another potential avenue of improvement was the way of scoring candidates. Namely, in the initial model, candidates are scored based on the ROUGE score with respect to the gold summary. Then, this score is used when we calculate the *Margin Ranking Loss* during the training phase. Inspired by the work of Liu and Liu (2021), I tried to replace the ROUGE score, with a similar semantic similarity score we use in the training phase. Specifically, the idea is to score candidate summaries based on the semantic similarity to the original document, not the gold summary. This way, we adjust the scores to the later downstream task. The idea was never properly tested because of the lack of time and long training time the model requires.

7. Conclusion

Text summarization is devoted to creating salient and concise subsets of textual data from the original content. As such, it is one most well-researched domains of natural language processing. As of now, text summarization is heavily influenced by deep learning, as almost all recent architectures rely on some form of artificial neural network. In case of the extractive approach, most neural frameworks follow a similar architectural pattern - extract sentences, score them and pick the most salient ones. However, this approach tends to oversaturate summaries with the most salient and general sentences that often encode same information. In other words, neural models that follow this pattern often create summaries with redundant sentences.

In this thesis, we present the problems of modern neural approaches, introduce and explain a novel architecture that mitigates redundancy issues and analyse the benefits of this new system. In the beginning, the fundamental neural models that are commonly used within the current extractive frameworks were described. The problems were identified with those neural frameworks and the solution was presented - a summary-level semantic text matching framework. Zhong et al. (2020) proposed this novel paradigm where instead of choosing salient sentences individually (sentence-level summarization), the focus is on simultaneously generating and picking the most salient summaries (summary-level summarization). The model reimplementation of this novel approach demonstrated the intricacies behind the semantic text matching model. The dataset dependent analysis for this new approach was also performed. The experimental results showed that the reimplemented model was on par with the original authors, although the training was limited by less powerful hardware.

Ultimately, based on the experimental results and dataset insights, we can conclude that this novel approach offers a lot of area for improvement of neural extractive systems. In the scope of this thesis, we discussed one tangible way of potentially improving performance by constructing the candidate summaries' score function more similarly to the downstream task.

BIBLIOGRAPHY

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Yoshua Bengio, Ian Goodfellow, and Aaron Courville. *Deep learning*, volume 1. MIT press Massachusetts, USA:, 2017.

Jaime Carbonell and Jade Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 335–336, 1998.

Jianpeng Cheng and Maria Lapata. Neural summarization by extracting sentences and words. In *54th Annual Meeting of the Association for Computational Linguistics*, pages 484–494. Association for Computational Linguistics, 2016.

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

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

Som Gupta and SK Gupta. Abstractive summarization: An overview of the state of the art. *Expert Systems with Applications*, 121:49–65, 2019.

Moritz Hardt, Ben Recht, and Yoram Singer. Train faster, generalize better: Stability of stochastic gradient descent. In *International Conference on Machine Learning*, pages 1225–1234. PMLR, 2016.

Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. Teaching machines to read and comprehend. *Advances in neural information processing systems*, 28:1693–1701, 2015.

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

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Aishwarya Jadhav and Vaibhav Rajan. Extractive summarization with swap-net: Sentences and words from alternating pointer networks. In *Proceedings of the 56th annual meeting of the association for computational linguistics (volume 1: Long papers)*, pages 142–151, 2018.

H. Jing and K.R. McKeown. Cut and paste based text summarization. 2000.

Mikael Kågebäck, Olof Mogren, Nina Tahmasebi, and Devdatt Dubhashi. Extractive summarization using continuous vector space models. In *Proceedings of the 2nd Workshop on Continuous Vector Space Models and their Compositionality (CVSC)*, pages 31–39, 2014.

Chris Kedzie, Kathleen McKeown, and Hal Daumé III. Content selection in deep learning models of summarization. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1818–1828, 2018.

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

Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

Yang Liu. Fine-tune bert for extractive summarization. *arXiv preprint arXiv:1903.10318*, 2019.

Yang Liu and Mirella Lapata. Text summarization with pretrained encoders. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3730–3740, 2019.

Yang Liu, Ivan Titov, and Mirella Lapata. Single document summarization as tree induction. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 1745–1755, 2019.

Yixin Liu and Pengfei Liu. Simcls: A simple framework for contrastive learning of abstractive summarization. *arXiv preprint arXiv:2106.01890*, 2021.

H.P. Luhn. The automatic creation of literature abstracts. *IBM Journal*, 1958.

Ryan McDonald. A study of global inference algorithms in multi-document summarization. In *European Conference on Information Retrieval*, pages 557–564. Springer, 2007.

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

Marvin Minsky and Seymour A Papert. *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

Ramesh Nallapati, Bowen Zhou, Caglar Gulcehre, Bing Xiang, et al. Abstractive text summarization using sequence-to-sequence rnns and beyond. *arXiv preprint arXiv:1602.06023*, 2016.

Ramesh Nallapati, Feifei Zhai, and Bowen Zhou. Summarunner: A recurrent neural network based sequence model for extractive summarization of documents. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

Shashi Narayan, Shay B Cohen, and Mirella Lapata. Ranking sentences for extractive summarization with reinforcement learning. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1747–1759, 2018.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. PMLR, 2013.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

Josef Steinberger et al. Evaluation measures for text summarization. *Computing and Informatics*, 28(2):251–275, 2009.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

Juan-Manuel Torres-Moreno. *Automatic text summarization*. John Wiley & Sons, 2014.

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

Andreas Zell. *Simulation neuronaler netze*, volume 1. Addison-Wesley Bonn, 1994.

Ming Zhong, Pengfei Liu, Danqing Wang, Xipeng Qiu, and Xuan-Jing Huang. Searching for effective neural extractive summarization: What works and what's next. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1049–1058, 2019.

Ming Zhong, Pengfei Liu, Yiran Chen, Danqing Wang, Xipeng Qiu, and Xuan-Jing Huang. Extractive summarization as text matching. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6197–6208, 2020.

Qingyu Zhou, Nan Yang, Furu Wei, Shaohan Huang, Ming Zhou, and Tiejun Zhao. Neural document summarization by jointly learning to score and select sentences. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 654–663, 2018.

Deep Neural Models for Extractive Text Summarization

Abstract

Large amounts of publicly available textual data made possible the rapid developments of neural natural language processing (NLP) models. One of the NLP tasks that particularly benefited from large amounts of text, but which at the same time holds promise for solving the problem of data overabundance, is automated text summarization. In particular, the goal of extractive text summarization is the production of a short but informationally rich and condensed subset of the original text. The topic of this thesis is to research and (re)implement a new approach within the extractive summarization field, which focuses on framing extractive summarization as a semantic text-matching problem. As of now, most of the neural extractive summarization models follow the same paradigm: extract sentences, score them and pick the most salient ones. However, by choosing the most salient sentences, we are often left with a summary where most sentences are redundant. Zhong et al. (2020) proposed a novel paradigm where instead of choosing the salient sentences individually (sentence-level summarization), the focus is on simultaneously generating and picking the most salient summaries (summary-level summarization). The objective of the thesis is to reimplement this novel paradigm, research the flaws of previous models, and potentially improve the capabilities of this new summarization approach. All references must be cited, and all source code, documentation, executables, and datasets must be provided with the thesis.

Keywords: natural language processing, deep learning, machine learning, text summarization, extractive text summarization

Duboki neuronski model za ekstraktivno sažimanje teksta

Sažetak

Velika količina javno dostupnih tekstnih podataka omogućila je brz razvoj dubokih neuronskih modela za obradu prirodnog jezika. Jedan od zadataka obrade prirodnog jezika koji je posebno profitirao od velika količina teksta, a koji istovremeno ima potencijal riješiti problem prekomjernosti podataka, jest strojno sažimanje teksta. Posebice, cilj ekstraktivnog sažimanja teksta jest stvaranje kratkog ali informacijskog bogatog i sažetog podskupa izvornog teksta. Tema ovog rada jest istražiti i (re)implementirati nov pristup unutar područja ekstraktivnog sažimanja, koji se fokusira na ekstrastivno sažimanje u kontekstu semantičkog podudaranja teksta. Do sada je većina neuronskih ekstraktivnih modela sažimanja slijedila istu paradigmu: izdvoji rečenice, ocijeni ih i odaberi one koje nose najviše informacija. Međutim, odabirom samo najistaknutijih rečenica često dobivamo sažetak u kojem je većina rečenica suvišna. Zhong i suradnici (2020.) predložili su nov pristup u kojem se, umjesto pojedinačnog biranja najistaknutijih rečenica (sažimanje na razini rečenice), naglasak stavlja na istovremeno generiranje i odabir najistaknutijih sažetaka (sažimanje na razini sažetka). Cilj rada jest nadograditi ovu novu paradigmu, istražiti nedostatke prethodnih i trenutnih modela te potencijalno poboljšati izvedbu ovog novog pristupa ekstrativnog sažimanja. Radu priložiti izvroni i izvršni kod modela, skupove podataka i programsku dokumentaciju te citirati korištenu literaturu.

Ključne riječi: obrada prirodnog jezika, duboko učenje, strojno učenje, sažimanje teksta, ekstraktivno sažimanje teksta