Czech
Technical
University
in Prague

Attention Mechanism in Natural Language Processing

Anton Kretov

Supervisor: RNDr. Marko Genyk-Berezovskyj
May 2020

Acknowledgements

I would like to thank ČVUT for giving me an opportunity to work on this project and to try out new technologies that define Natural Language Processing research space in 2020 by applying them to the Czech language. I want to also express my gratitude to Trask solutions a.s. for supporting my NLP activities.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date 22. May 2020

Abstract

Natural language processing (NLP) technologies have always been actively developed to solve tasks in the most widespread languages like English or German. However, there are a few papers and solutions dedicated to smaller Slavic languages like Czech. This paper describes various NLP algorithms of sequence processing. It is primarily focused on modern techniques like Attention mechanism and such architectures like Transformer or Reformer, which are built on it. This thesis examines what advantages this mechanism has compared to conventional recurrent neural networks. It is afterwards tested on two Czech NLP tasks of various degrees of complexity - diacritics correction and abstractive text summarization.

Keywords: neural networks, machine learning, NLP, diacritics, neural machine translation, attention mechanism, Transformer, Reformer, Trax, abstractive summarization

Supervisor: RNDr. Marko Genyk-Berezovskyj

Abstrakt

Technologie pro zpracování přirozeného jazyka (anglicky Natural Language Processing - NLP) se již delší dobu aktivně vyvijí pro nejrozšířenější světové jazyky, např. angličtinu a němčinu. Nicméně, existuje málo prací a řešení pro menší slovanské jazyky, kam patří i čeština. Tato práce je věnována obecně tématu NLP a algoritmům zpracování sekvenčních dat, je zaměřená především na nejnovější techniky, jež je Attention mechanismus a modely Transformer a Reformer, které jsou na tomto mechanismu založené. V práci je podrobně popsáno, jaké má attention mechanismus výhody oproti konvenčním rekurentním neuronovým sítím. Attention mechanismus je pak důsledně otestován na dvou různě složitých NLP úlohách v českém jazyce - doplnění chybějící diakritiky a abstraktní anotaci dlouhých textů.

Klíčová slova: neuronové sítě, strojové učení, NLP, diakritika, neuronové strojové překladače, attention mechanismus, Tranformer, Reformer, Trax, abstraktní anotace

Contents

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Kretov Anton**	Personal ID number:	**474672**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Branch of study:	**Computer and Information Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Attention Mechanism in Natural Language Processing

Bachelor’s thesis title in Czech:

Attention mechanizmus ve zpracování přirozeného jazyka

Guidelines:

Learn Encoder-Decoder model and attention mechanism in recurrent neural networks (RNN) utilised in natural language processing (NLP). Become acquainted, as much as possible, with the current advances in the field. Feasibility of this class of RNN in Czech language processing has not been studied in detail yet. Concentrate specifically on the problem of diacritics correction and the problem of text summarization in texts in Czech language.
Try to identify the topics which should or might be addressed in further developmnt of RNN applications in Czech language processing in the near future, and suggest tools, both existent and non-existent yet, appropriate for the task.
Choose or prepare sufficiently large training data set (corpus). Evaluate the feasibility of the given class of NN using the chosen training data set. Implement an interface for network training and management. If necessary, equip the interface with visualization tools of your choice. Summarize the conclusions based on your experiments.
Supplement your implementation with programmer and user documentation.

Bibliography / sources:

[1] J. Náplava, Natural Language Correction, Master Thesis, Charles University, Prague, 2017
[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L.Jones, A. N. Gomez, L. Kaiser, I. Polosukhin, Attention Is All You Need, arXiv:1706.03762, 2017
[3] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, arXiv:1810.04805, 2019
[4] Dzmitry Bahdanau, KyungHyun Cho, Yoshua Bengio: Neural Machine Translation by Jointly Learning to Align and Translate , arXiv:1409.0473, 2016
[5] Minh-Thang Luong, Hieu Pham, Christopher D. Manning: Effective Approaches to Attention-based Neural Machine Translation, arXiv:1508.04025, 2015
[6] Milan Straka, Nikita Mediankin, Tom Kocmi, Zdeněk Žabokrtský, Vojtěch Hudeček, Jan Hajič: SumeCzech: Large Czech News-Based Summarization Dataset, internal report of Association for Computational Linguistics, 2017, retrieved 27.1.2020 from https://www.aclweb.org/anthology/L18-1551.pdf

Name and workplace of bachelor’s thesis supervisor:

RNDr. Marko Genyk-Berezovskyj, Department of Cybernetics, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **10.01.2020** Deadline for bachelor thesis submission: **25.05.2020**

Assignment valid until: **30.09.2021**

RNDr. Marko Genyk-Berezovskyj	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Chapter 1

Introduction

1.1 Preface

The project I have been working on is my contribution to the NLP (Natural language processing) research conducted by a fringe group of data scientists and researchers in the Czech republic. Considering the ratio of available NLP materials, frameworks, models, solutions, applications and datasets in such languages like English and German compared to the amount of work in smaller Slavic languages like Czech, my aim is to enlighten Czech NLP stack with some new technologies and ideas that have not been presented yet. My second objective is to inspire further exploration of this amazing language in computer science community and achieve motivating results by incorporating existing approaches in Czech NLP and assess their performance.

This project is also a part of vast NLP research conducted by a team of analysts in the Czech technological company Trask solutions a.s. I am a part of. By supporting this particular work, Trask poses a challenge of understanding Czech language with the help of modern machine learning approaches in pursuit of helping businesses to process textual data, understand it and provide a customer with better solutions based on it.

My research consists of two tasks - automatic diacritics completion and abstractive text summarisation. This work aims to show the way how to solve these tasks by applying modern techniques which have arisen in recent years.

Taking into account the fact that this is a bachelor thesis work, I provide exhaustive background of the current Czech NLP research, existing methods and instruments, describe the way NLP evolved, which is essential for establishing background understanding of how NLP looked like for more than a decade. Moreover, I describe all the technologies and frameworks I used to achieve the results presented here.

1.2 NLP

The field of AI research called NLP (Natural language processing) is one of the most challenging and fast-changing among all spheres that are currently in the focus of topic of artificial intelligence. The techniques developed while solving numerous tasks have already been successfully applied in medicine, planning, image recognition, fraud detection and many others fields. One of the most outstanding achievement of NLP research are neural machine translation techniques that are vastly used in other tasks.

Both tasks of diacritics completion and writing summaries can be considered a translation task. We will talk about the more precise definition of these tasks in chapter 3.

Chapter 2

NLP and machine translation techniques

2.1 A short history of NLP

When we talk about machine translation, it is crucial to realise, how the whole industry evolved in last decades. Numerous innovations by some means associated with machine translation have been applied in many spheres outside NLP, accomplishing notorious many tasks.

Since the history of machine translation is one of the richest with its origins traced back to the 9th century, when the Arabic cryptographer Al-Kindi proposed several techniques of basic translation from one language to another, our interest has to be assigned to the 20th century, when humans started to think about digitalization of many agendas. The very first approach to solve the task of automatic machine translation is based on a set of rules how to translate words and phrases in such a way that they preserve their meanings in the target language. Obviously, there are literally countless ways of expressing people's feelings and emotions, that's why this method, despite being the most accurate, is time and resource intensive and requests enormous human effort and consolidation, since translation is considered to be art. Moreover, a language is a living form with many phrases becoming extinct and others occur.

The aforementioned thoughts, computational power increase and data accumulation lead people to think about such problems in statistical terms. The statistical machine translation field arose with various billingual parallel text corpora. Although there were many successful implementations of statistical translators such as CANDIDE [BBDP+94] from IBM or SYSTRAN from Google, they still lacked rare language pairs data. Statistical machine translation incorporates pattern recognition and several sequence modelling techniques like Markov chains or conditional random fields (CRF) [LMP01].

Nevertheless, the methods proposed by statistical and rule-based approaches (including the hybrid ones) still do not equip machines with language understanding, which is part and parcel of how the same thought is interpreted in another language. Moreover, statistical methods such as CRF or SVM [CV95] possess a huge disadvantage of being short-sighted as they are designed to take into account a short window of text at the same time and process text in a "bag-of-words" fashion, which is unacceptable since the word order matters.

With recent development of artificial neural networks and particularly the field of deep learning, there has been an increase of newly proposed algorithms and techniques, which are covered in the next chapter.

2.2 Neural networks in machine translation

2.2.1 Feed-forward neural networds

Artificial neural networks are designed in such a way that they accept a fixed-size vector representing data we want to process and yield another vector. More particularly, neural networks consist of many cells called neurons that run perceptron algorithm [Ros58], which is a dot product of a vector and the neuron's weights followed by a non-linear function, usually softmax or ReLU. With the help of the backpropagation algorithm [RHW88] the network "learns" to map vectors from one linear space to another. Since a feed-forward artificial neural network consists of several perceptrons, it is considered a linear classifier.

With machine translation in mind, it means that the network is able, for example, tolearn how to translate a word or a phrase (represented by a vector in a linear space) from one language to another. Although the idea of encoding each and every word to a finite linear space has made a revolution in NLP after the rise of so-called word embedding with such renowned algorithms as Word2Vec [MSC+13], or GloVe [PSM14], it is still the whole text which bears the meaning of the human's thought, not single words. Do not forget that a language is a more sophisticated structure with synonyms, antonyms, polysemes, etc.

Apparently, for the combinatorial reasons it is impossible to encode all the possible phrases and sentences into any finite space. This is what lead to the rise of recurrent neural networks.

2.2.2 Recurrent Neural Network (RNN)

The term of Recurrent Neural Networks is not brand new and has its history starting from the work of John Hopfield in 1982 [Hop82]. RNN is a class of artificial neural network having a memory unit and forming a directed graph along a temporal sequence with a linear classifier upon it. The main idea is to propose an algorithm of sequence processing and taking the order of the sequence tokens (usually called time stamps) into consideration. The main difference from the aforementioned feed-forward linear network is its ability to process variable-length inputs and generate a so-called feature vector, which is intended to carry information about the whole sequence and represent its context before entering the linear classifier stage. The recurrent network learns by adjusting the inner weights of each RNN cell.

Recurrent neural networks introduced the way to process data spread over time. Nevertheless, they have several critical drawbacks.

2.2.3 Vanishing gradient

Recurrent neural networks are theoretically able to process a sequence of any length. We can process a word, a sentence, or the whole paragraph or an article by the same architecture. Apparently, the amount and the quality of information encoded into a fixed-size vector after all time stamps pass will differ. Since RNNs are trained with the help of the back-propagation algorithm, which is based on partial derivatives and gradients, applying partial derivatives with respect to the weights of farther memory cells (i.e. the first ones in the graph) will make little change and hence be less effective with the growth of the sequence length. This problem is called vanishing gradient - caused by a derivative of a non-linear activation function that adjusts vectors' values typically to the interval $[0, 1]$ or $[-1, 1]$.

2.2.4 Long Short-term memory - LSTM

LSTM is a recurent neural network architecture, which has a more complex structure than a simple RNN and which is designed to solve vanishing gradient problem for longer sequences. They have been considered state-of-the-art and proved to give promising results even on extremely large passages, however, they are hardware unfriendly and sluggish. The way they are defined is particularly sophisticated (showed in figure 2.1), making them impossible to train on a CPU. This is the reason these networks were not present, although introduced in the 20th century by Sepp Hochreiter and Jürgen Schmidhuber [HS97], by the time people learned how to perform large training on Graphical Processing Units (GPUs) effectively, resulting in convolutions and recurrences acceleration. Nonetheless, the problem of their poor architecture persists.

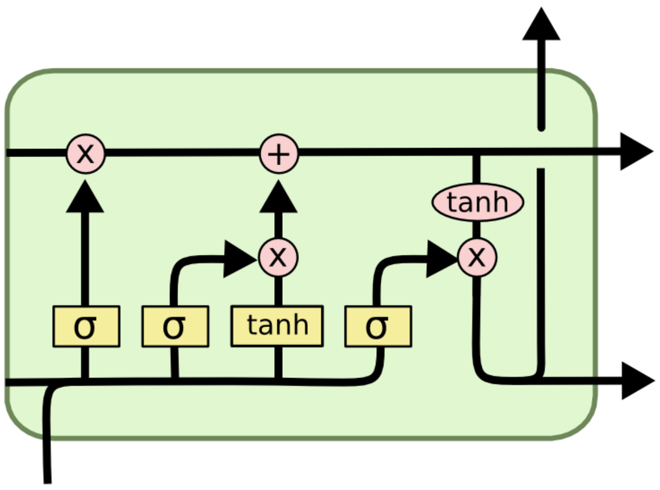


Figure 2.1: LSTM cell architecture, source: [Chr20]

LSTM networks have been widely used in 2010s and have been incorporated into various architectures. Since a machine translation task is defined as a sequence-to-sequence problem, the LSTMs have been used as a mapping algorithm between two sequences.

The major flaw of using LSTMs in such a fashion is in its requirements - the network is incapable of processing variable sized sequences. The maximum length of the input has to be known in advance. Then, all the sequences have to be padded with special "pad" tokens to meet this requirement. All the longer sequences have to be truncated and thus lack information.

2.2.5 Encoder-Decoder architecture

The way a sequence is processed in a LSTM cell was a motivation to create an architecture of a model which is at the moment of writing this work considered state of the art in sequence translation problems. The model is called Encoder-Decoder.

An Encoder-Decoder recurrent neural network consists of two obvious logical parts - an encoder and a decoder. The simplified scheme is depicted in figure 2.2

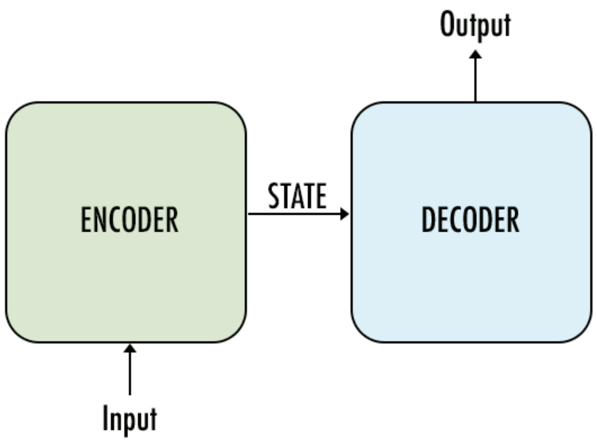


Figure 2.2: Encoder-Decoder basic architecture, source: [JKS18]

As we can see, the input sequence is firstly processed by the encoder block. Then, a vector representing this sequence is passed into the decoder block. The decoder attempts to map the input vector to the target sequence.

It is necessary to realise how both blocks operate with data. Speaking of an encoder block, it consists of a stack of recurrent layers, typically LSTM or GRU (Gated Recurrent Unit), which take the input tokens from left to right or vice versa. The last RNN cell in this chain returns two vectors - h, which is called *a hidden state* and a vector c, which is called *a cell state*. These are lately used in a decoder block. However, this is the case of an encoder consisting of one recurrent layer. In case of a more complex architecture all the layers except for the last one on the top of the stack return a list of hidden vectors from each recurrent cell and pass them as input to the next layer. The last layer takes the output of the preceding layers as input and returns vectors h and c, representing the whole sequence. It is considered that a deeper encoder architecture is capable of learning more complex relations between sequence's tokens and therefore construct more accurate representation.

A decoder is represented by a similar neural network, consisting of LSTM or GRU cells, which process sequence from left to right. The decoder in this architecture works as an autoencoder, i.e. its objective is to use the information gathered during encoding process and to re-use its outputs to generate new sequences in an autoregressive fashion. First of all, the first recurrent cell is initialised with the outputs of the encoder's last recurrent cell. By doing this, the decoder is provided with context of the whole sentence. Then, it accepts a special token, which is usually referred as `BOS` (Beginning Of Sentence) token. `BOS` token starts the decoding phase. At this point, the first recurrent cell is prompted to give the first target token. This token is defined as h_0. Consequently, the vector h_0 is passed to a linear layer with a softmax non-linear activation function which maps it to a probability distribution over all possible output tokens. In terms of translation it could be either a character from the target alphabet or a (sub)word from the target dictionary. In the meantime, the same vector h_0 is being used as an input vector to the next decoder's cell, so that the next cell is "aware" of the previous output. This is an essential aspect of how the decoding phase is implemented. The decoder works iteratively (precisely, in an autoregressive fashion), yielding one time stamp per step and using it as an input token for the next step so that the following decoder's cell can make use of other parts of the encoded sentence and be aware of the current output sequence at the same time.

The complete pipeline is depicted in the figure 2.3.

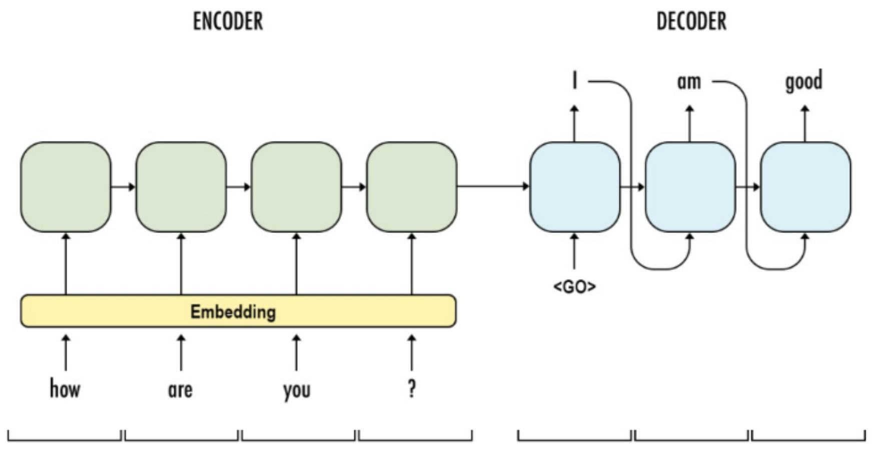


Figure 2.3: Encoder-Decoder basic architecture, source: [JKS18]

Surprisingly, an Encoder-Decoder architecture is what revolutionised the field of machine translation, with such prominent models like Google's NMT (Neural Machine Translation) [WSC+16a] or many dialog systems [ZLLE17], question answering systems [YJL+16] and etc.

2.2.6 Drawbacks of the Encoder-Decoder architecture

Nevertheless, this architecture is not flawless. The proposed Encoder-Decoder model, unfortunately, has several limitations. The most significant one is the fact that it is hard to transform all necessary information about long sequences to a fixed-size vector. As is has been proved, the sequences having too many tokens to remember fail to be adequately translated in the decoding phase. This unpleasant feature has been revealed while attempting to apply an Encoder-Decoder model to the text summarisation task, where long passages of texts failed to be properly encoded.

One possible workaround could be to enlarge the size of the fixed size context vector. However, due to the architecture of a RNN cell it is too computationally expensive to add too many of them, making training a burden.

The second major problem resides in the recurrent cells' architecture. As the sequences become longer, the sequential nature of the data flow inside RNN plays a major role here and makes it a huge bottleneck in the training phase, meaning that the time needed for training deeper network is implausible.

2.3 Attention, please!

There is a certain motivation in what Attention in NLP is and why it has been introduced to make one more revolution.

People do not read text in a way machines do. We do not necessarily read from left to right or from right to left. The natural way of perceiving text is rather targeting the information needed right now and not attending the rest. To give an example, when a human notices a preposition in a text, he tries to find the object the preposition is related to. It means finding all the nouns mentioned in the preceding text and attempting to find the described object. The tasks does not really requires to look forward in text, since prepositions are mostly used to refer to already introduced things.

With that in mind, the neural network could mimic the human's behaviour and this is the moment when the attention mechanism has to be introduced.

Attention is an interface between the encoder and decoder that provides the decoder with information from every encoder hidden state. By providing all the hidden states to the decoder, the model learns the alignment between the encoder outputs and the current token to be processed. The term alignment here is defined as [Bus]: "Alignment is a process of matching source and target segments of text in order to create a translation memory. The purpose of alignment is to capture relations of equivalence or correspondence in a translation".

2.3.1 Luong attention

There are mostly two types of attention. These are Bahdanau [BCB16] and Luong [LPM15] attention. Luong attention is a simplified variant of Bahdanau attention and is easier to understand while producing competitive outputs.

Luong attention is depicted in the figure 2.4.

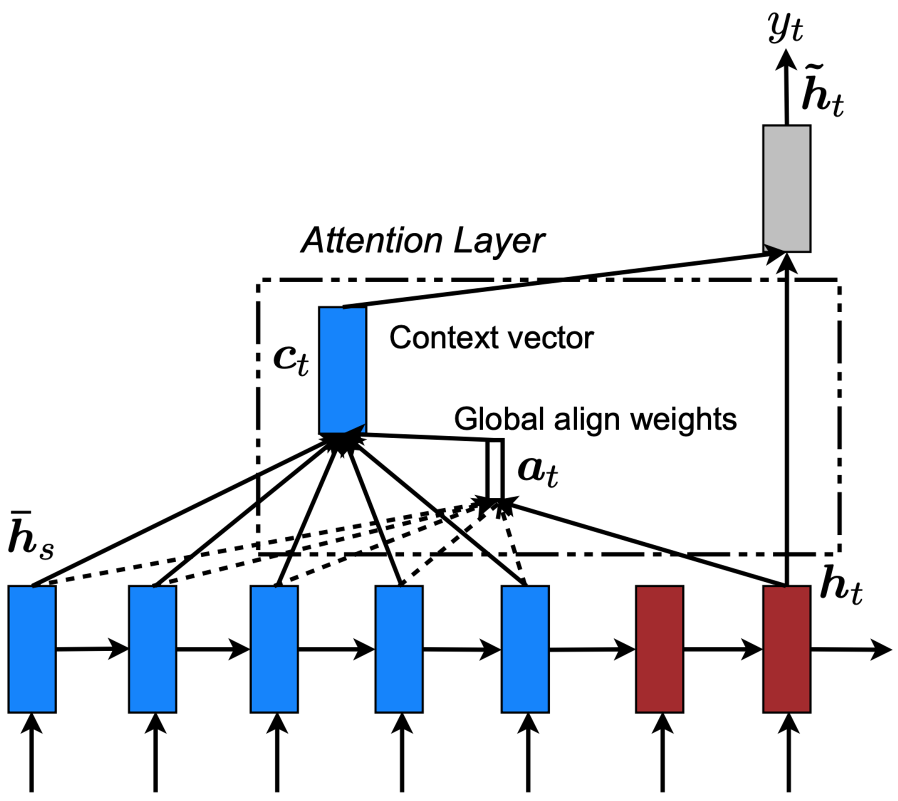


Figure 2.4: Luong Attention mechanism, source: [LPM15]

The idea is to put additional information to the decoder's output in order to help out the linear classification layer in making more accurate decisions. The principle of Luong attention in quite straightforward:

1. Firstly, the decoder outputs hidden states for all input tokens to translate. Denote them as h_t for each timestamp t

2. Then, we calculate score of this decoder output h_t with all encoder outputs, which we denote as $\overline{h_s}$. Scoring function has one of the following forms:

$$score(h_t, \overline{h_s}) = \begin{cases} h_t^T \overline{h_s} & \text{dot} \\ h_t^T \mathbf{W_a} \overline{h_s} & \text{general} \end{cases}$$

   The idea behind these operations is to measure a kind of correlation between each vector to find out, which ones matter more. The use of the general approach is highly recommended due to the fact, that the matrix $\mathbf{W_a}$ represents a linear layer with the weights that are also learned during back-propagation, making the final scoring function more domain-specific. However, in more recent models the simpler "dot" alternative is used.

3. The next step is to convert scores computed previously to the probability distribution over all encoder outputs in order to gain intuition what tokens carry more weight. This is done by a simple function:

$$align(score) = softmax(score) = \frac{exp(score(h_t, \overline{h_s}))}{\sum_{s'} exp(score(h_t, \overline{h'_s}))}$$

4. The last step is to create a context vector, which is done by a dot product:

$$context = align(score)^T \overline{h_s}$$

   Note, that all the aforementioned dot products can be transformed to matrices multiplications, which significantly speeds up the whole attention process.

5. The derived context vector is concatenated with the decoder's output as an additional feature:

$$h_{t'} = [h_t; context]$$

6. The last step of the attention algorithm is to apply one more linear layer above the concatenation to let the whole model know about the contexts around. Formally, this step looks like this:

$$out = \mathbf{W_c} h_{t'}$$

The vector *out* is now the output from the attention mechanism and can be fed to the feed-forward linear classifier.

This simple add-on provides decoder with essential information and is believed to significantly improve the performance of the resulting model. The only drawback of incorporating attention in the neural network is its requirement to hold all hidden states in memory and an increased computation needed to calculate the alignments. Moreover, if the linear trainable transformation is added to the attention block, the total number of weights the model has to learn is increased and the time and effort needed to train the network increase as well.

2.3.2 Attention is all you need

Although the crucial limitation of Encoder-Decoder networks being incapable of handling longer sequences has been circumvented by introducing attention, it is still unreasonably expensive to train deeper networks that solve more complex tasks due to the recurrent cells. This challenge has been accomplished in 2017 with the invention of Transformer [VSP+17].

Transformer is a novel approach of processing sequences. It appeared to be that no recurrence is needed to encode sequential data. The only thing that matters is attention. With the publication of one of the most influential papers in NLP world "Attention is all you need" [VSP+17] by a team of Google Brain scientists, a great NLP revolution came in place and completely changed the way sequences are processed today.

2.3.3 Transformer's architecture

Transformer model is basically derived from an Encoder-Decoder structure. Transformer follows this overall architecture by incorporating only so-called self-attention blocks with a point-wise linear layers. The model scheme is shown in figure 2.5

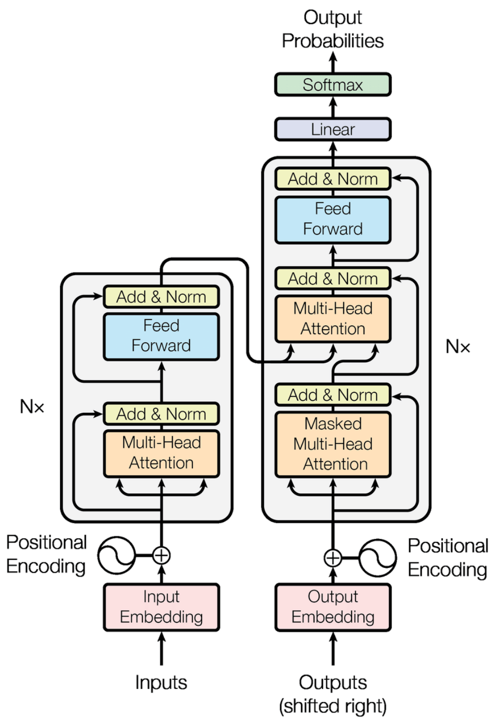


Figure 2.5: The Transformer, source: [VSP+17]

2.3.4 Encoder block

Before an input sequence enters an encoder block, several routines are done. Firstly, each token has to be embedded to the finite vector space. Then, the positional information has to be added to these vectors so that the model takes into consideration the sequential nature of input data. This is done by a special positional encoding function, usually in the form of sine and cosine functions of different frequencies:

$$PE_{(pos,2i)} = sin(\frac{pos}{10000^{\frac{2i}{d_{model}}}})$$
$$PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{\frac{2i}{d_{model}}}})$$

where pos denotes token's position, i denotes the embedding dimension to be encoded, d_{model} is the embedding dimension. Hence, i belongs to the interval $[0, d_{model}]$.

When the positional encoding is calculated, it is piece-wisely added to the embedded vectors. Then the input vectors are prepared to enter the encoder.

The encoder consists of two consequential steps: multi-head attention and linear transformation with non-linear activation function applied to each position separately and identically. The role of the attention blocks here is similar to the recurrent cells, however, with less computational requirements.

2.3.5 The idea behind multi-head attention

Multi-head attention can be seen as a parallel application of several attention functions on differently projected input vectors. The motivation of introducing more alignments instead of one is straightforward - by doing more projections and attention computations the model is enabled to have various perspectives of the same input sequence. It jointly attends to information from different angles, which is mathematically expressed via different linear subspaces. To reduce the impact of more attention computations, attention heads project information to more shallow subspaces, whose dimensions sum up to the model's depth.

Before we define multi-head attention more formally, it is necessary to understand what information is fed to the attention mechanism in this case. The way the attention was introduced in 2.3.1 is slightly modified, since the whole process is run on just one sequence. This is when the term "Self-Attention" is defined.

2.3.6 Self-Attention

Self-Attention is a simplification of a generic attention mechanism, which consists of so-called queries Q, keys K and values V. The origin of such naming can be found in search engines, where a user's *query* is matched against internal engine's *keys* and the result is represented by several *values*. In case of encoder's self-attention module all three vectors come from the same sequence and represent word vectors with positional encoding built in. For the sake of simplicity, we pack the whole sentence into matrices Q, K and V. Moreover, this notation enables parallel attention execution, speeding up the whole system. The alignment is computed on Q and K, which is then applied to V, resulting in the new vectors we attend to.

2.3.7 Scaled Dot-Product Attention

Scaled Dot-Product Attention is a slightly modified version of classical attention, where the scaling factor $\frac{1}{\sqrt{(d_k)}}$ is introduced in order to prevent the softmax function from giving values close to 1 for highly correlated vectors and values close to 0 for non-correlated vectors, making gradients more reasonable for back-propagation.

The mathematical formula of Scaled Dot-Product Attention is

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

2.3.8 Putting it all together

Multi-Head attention applies Scaled Dot-Product Attention mechanism in every attention head and then concatenates the results into one vector followed by a linear projection to the the subspace of the initial dimension. The resulting algorithm of multi-head attention can be formalised as follows:

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O$$

where $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$

The building blocks of the multi-head attention are depicted in figure 2.6

The second step is a simple position-wise fully connected feed-forward network. There is a residual connection around each block, which is followed by a layer normalisation. This is done for several reasons. Firstly, as the information passes deeper into the network, we want to ensure it is not lost, so the residual connections help the network to keep track of data it looks at. Layer normalisation plays an important role in reducing features variance. These operations can be grasped as a special rearrangement inside the batch before entering the next step.

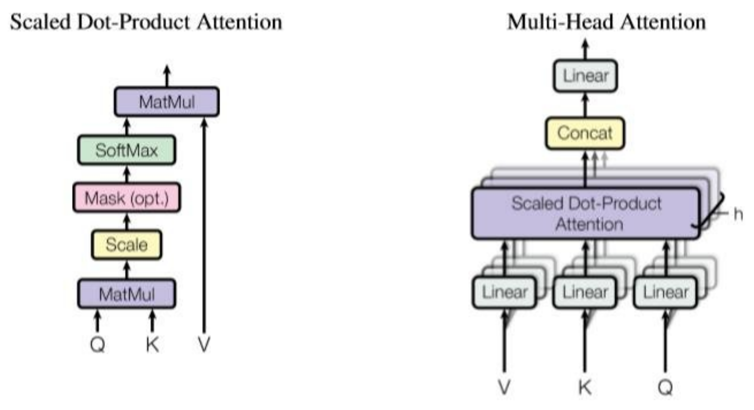


Figure 2.6: Multi-Head attention scheme, source: [VSP+17]

2.3.9 Decoder block

The decoder block looks pretty similar to the encoder block, however it inserts a third attention sub-layer between multi-head self-attention and a piece-wise fully connected layer. The need for the third component becomes evident considering the task a decoder attempts to fulfill. This block is responsible for attending to the encoder's output and aligning it with current decoder's input.

The first block inside the decoder is a modification of the multi-head attention described above. Before the softmax operation is applied, the decoder needs to mask out all the inputs to the right of the current input vector. This prevents decoder from "looking into the future" and makes it operate only with data processed so far. The model is pushed to make decisions which are dependent on the known outputs at each time step.

The second block is equivalent to the multi-head attention block used in encoder with both queries Q and keys K coming from the encoder instead of the same source. The values V are taken from the outputs of the previous block. By doing this, self-attention mechanism learns to link both sources and pay attention to the appropriate information from the encoder for making decisions.

After the alignment is done, the outputs are passed to a piece-wise feed-forward layer and then to the output of the decoder. After the decoder produces the output vector, it is linearly projected to the output's dictionary subspace and the probability scores are assigned to each possible output value.

The one with the highest probability is selected.

Apparently, there are residual connections and layer normalization between all three components of the decoder, so the information is preserved from getting lost.

2.3.10 Transformer versus RNN

At first glimpse, the introduced architecture of Transformer seems too much complicated compared to conventional RNNs. However, it turns out to be more efficient in terms of computational resources it needs for training and more transparent when it comes to the topic of model's interpretability. One of major advantages of conventional machine learning classifiers like SVMs and decision trees over neural networks is that their decisions are interpretable. By excessive feature selection made before training it is possible to argue model's decisions. Nothing similar exists in neural networks. Although it is possible to take a look inside some convolutional layers of computer vision systems and get little insight about how the data is compressed, nothing similar exists in recurrent neural networks. This issue is partially addressed by incorporating attention over all hidden vectors that the recurrent network generates, it is still hard to imagine how the data is flown from left to right.

That's why the model built only on attention blocks can be visualised, since attention generates alignment scores which are probability distributions.

One more problem of recurrent neural networks lies in their sequential nature. They are capable of processing data in only one direction. By adding or concatenating both left-to-right and right-to-left vectors it is still impossible to achieve truly bi-directional encoding. Transformer, which substitutes recurrence for attention block followed by a linear layer, processes all data at once, making encoding bi-directional. The advantage of such a scheme can be seen by comparing two renowned language models ELMo [PNI$^+$18], which consists of a stack of several LSTM blocks, and BERT [DCLT19], which is a slightly modified version of the Transformer model. By leveraging truly bi-directional nature of the underlying Transformer, BERT achieves much higher results in all benchmarks and easily outperforms all models based on recurrent neural networks.

Since recurrent neural networks are sequential, it is impossible to parallelize their training. The information flow in one particular direction is a strong bottleneck. On the other hand, Transformer takes advantage of having simple matrix multiplications, which are easily parallelizable, that's why are faster to train (see p. 8 of [VSP+17] where the number of TFLops needed to calculate several recurrent models and a Transformer model are present in table 2 along with the resulting accuracies.).

2.3.11 Transformer's disadvantages

Though being more efficient during training and inference, Transformers have their own limitations. The most vital one is their memory requirements. Since the recurrence is substituted with linear layers and multi-head attention, the number of parameters a single model has to hold is enormous compared to the recurrent network. While a simple recurrent network requires around 1M parameters to be trained, a model based on the Transformer architecture holds one magnitude more parameters. Moreover, as the sequence length grows, the model has to keep more parameters for both intermediate feed-forward layers and attention, which for a sequence of length L is $\mathbf{O}(L^2)$ in both computational and memory complexity. It indicates that in order to train a deep Transformer model to solve complex tasks, one realistically has to train it in large industrial research laboratories with expensive accelerators, since the whole model is incapable of being fitted to a single GPU or a small GPU pod. Additionally, things get even worse after we realise that there are a few companies and institutions who are capable of training large networks, which can destroy many ML fields, see the blogpost "How the Transformers broke NLP leaderboards".

2.3.12 Reformer: The Efficient Transformer

In 2020, the new approach how to overcome Transformer's limitations has been introduced. It is called `Reformer` [KKL20].

Reformer is a modification of yet conventional Transformer model, where the following techniques have been used:

1. Classical Transformer model consists of a stack of N encoder and decoder blocks. Apparently, the activations of each of them are stored for a backpropagation routine. In Reformer, this bottleneck is solved by

incorporating reversible layers, firstly introduced in [GRUG17], which eliminate the need to store all N activations to just a single copy.

2. The problem of huge memory consumption of intermediate feed-forward linear layers is solved by splitting activations into chunks.
3. Attention scores are approximated by locality-sensitive hashing mechanism, which is more propitious with longer sequences.

All the amendments done on the Transformer model do not have negative effect on its performance, making a Reformer model more applicable for those lacking powerful distributed accelerators. Both attention approximation instead of full-fledged computation and reversible layers have negligible effect on Reformer's performance, making it faster to train and infer.

2.3.13 Availability of Transformer and Reformer

At the time of writing, the only framework supporting both Transformer and Reformer models is Trax. Trax is a relatively new package which was initially a part of larger and well-known Tensor2Tensor framework. There is a deep learning library Tensorflow developed by Google lying under the hood of both frameworks. While Tensor2Tensor framework is by 2020 no longer developed and is in a maintenance mode, it is written in Tensorflow 1, which is less effective than newly introduced Tensorflow 2.0 Trax is built on.

Although being extensively used for many deep learning tasks as a command line interface, Tensor2Tensor has become extremely complicated. It is especially hard to set up and tune each and every hyperparameter it has for correct model training. Moreover, after version 1.2.9 Tensor2Tensor has become unstable and exceptionally hard to train, causing many models to diverge at early stages. There are countless issues raised on Tensor2Tensor Github issues page pointing out at this instability. Trax benefits from its simplicity. On the other hand, it is poorly documented and contains a number of bugs I faced while working with it, which are actively fixed by Trax's developers. The major disadvantages of Trax are:

1. The lack of documentation. Lots of modules and inner mechanisms are not sufficiently documented, resulting in a trial-and-error workflow.

2. Attention weights visualisation is not implemented out-of-the-box. I overcome this problem by implementing the attention weights visualisation by myself and preparing to place a pull request in order for others to benefit from this functionality.

3. Third-party modules dependencies. Trax relies on several modules that are implemented and supported by other Google teams. The major ones are `JAX`, which is a module for fast computing on different accelerators and precompilation of neural models so that they run faster, and `gin-config`, which is a module for easier properties configuration. These dependencies together with bugs not caught yet drastically complicate the user experience.

4. Memory leaks. By the time of writing this, Reformer model still has a critical memory leak while running inference with beam search algorithm. There is also an issue with Transformers trained on TPU described below.

5. Beam Search algorithm is incorrectly implemented. The problem I faced is that while used as greedy search (it means by setting beam size to 1), the search algorithm outputs something different compared to the model which is run in a classical encoder-decoder fashion.

6. Yet small community. As soon as you are stuck and cannot move on, there is a huge chance that nobody advises you. It is still extremely young and narrow.

With that said, I hope that my contribution to this framework will make it better and more stable, leading it to be more user-friendly.

Chapter 3

Tasks accomplished in this work

The theory presented in 2.2.5, 2.3, 2.3.3, 2.3.12 is crucial for understanding how modern neural networks handle sequences and what advantages and disadvantages each of them possesses. We have seen many techniques how to overcome some critical limitation these networks had at earlier stages. This section covers two different NLP tasks which are in high demand in Czech NLP community.

3.1 Diacritics correction

The task of diacritics correction can be formalized in this way: each sentence consists of a sequence of characters, we denote it as

$$s = (c_1, c_2, ..., c_n), n \in \mathbf{N}$$

where s is our sentence, $c_i, i \in 1, 2, ..., n$ is ith letter and n is the sentence length.

Each character c is an element of a set of characters called an *alphabet*. We define an alphabet as follows:

$$A = \{c, \exists s_i \in D : c \in s_i, i = \{1, 2, ..., m\}; m \in \mathbf{N}$$

, where D is a set of sentences s_i, m is the size of the dataset D.

Translated to English, A is the set of letters encountered in a list of sentences called a *dataset*. The discussion about the structure and contents of the dataset will be held after we define all neural architectures capable of solving this task. For the sake of simplicity, by saying *dataset* here we bear in mind both correctly and incorrectly written sentences.

Now we are ready to formulate the task a model has to accomplish: given a sentence s' containing $k = 0, 1, 2, ...$ letters with missing diacritics, transform it to the sentence $\overline{s}$ having $k = 0$ letters with missing diacritics. In other words, if a sentence has a letter with missing diacritics, it has to be corrected in the output.

It is important to emphasize how the problem is defined, since there are many ways of doing it considering the problem of diacritics correction. Here we assume that there are *some* incorrect tokens, that's why we will be able to train a model to handle partly-dediacritisized sentence as well as fully-dediacritisized. This technique lets us run the diacritization model recursively: even if the model was not able to fill in all the missing diacritics, we can take the output of the model and use it as the input to the same model. Moreover, such a formulation makes sense, because input partly-dediacritisized sentences resemble the way Czechs write. The Czech alphabet is broader than the English alphabet and definitely cannot be optimally fit into a classical QWERTY keyboard layout. The Czech alphabet consists of 42 letters, with 15 letters having a diacritisized counterpart. That's why it is a challenge to handle 15 additional characters effectively given the same layout. The solution was found by incorporating 2 different strategies:

1. Once you switch to the Czech layout, all the number keys with several keys next to the "Enter" key become diacritisized versions of latin letters.
2. Another way of adding diacritics in a text is to use a shortcut "Shift" + "=", or just "=" key before typing a letter. The next letter you write becomes accented with the selected diacritics sign.

This unhappy situation led Czechs to pay no attention to the diacritics and partly abandoning it, since it can be inferred from the letters or words around a particular character. In addition, several letters are usually written with diacritics, which is given by a habit of each individual.

In view of the aforesaid, the way we defined the diacritics correction task appears to be the most optimal. However, this formulation has a downside: in order to perfectly learn to map sentences, we have to enlarge the dataset.

Fortunately, we can do it by randomly deleting some diacritics from the input sentences while repeatedly using them for training. This pleasant feature will find its usage when we will speak about diacritics dataset representation.

Another point worth mentioning is the way the whole problem is treated. On the one hand, in the light of the topic of machine translation and since the task is well-posed, it is possible to treat diacritics problem as a machine translation task. Czech language in this case can be divided into two subsets "with" and "without" diacritics. Obviously, the transition from the language "with" diacritics in place to the language "without" diacritics is pretty simple, the backward operation is not that trivial.

On the other hand, we realise that it may seem cumbersome to treat such a simple task of mirroring most of the letters as a translation task. This particular problem can be defined in another way: given a letter c, decide whether we need to add a diacritics sign to it. This formulation narrows the task from choosing among all possible letters from the target dictionary (which typically consists of hundreds of symbols including punctuation marks, lower- and upper-cased letters etc.) to assigning several labels indicating whether or not the diacritic symbol is needed there.

3.1.1 Previous works

There are not so many works present in the Czech computer science community that discuss the problem of diacritics correction. One of the most promising works is the Master thesis of Jakub Náplava [Ná17], who uses a simple recurrent neural network with residual connections to solve the diacritics task. It showed high accuracy dealing with data acquired from Prague Dependency Treebank [BHH^{+}13] and several other sources. There is one more work that resulted in a tool named Korektor, which is a much simpler program not utilising neural networks. Therefore, I do not discuss it in the following text.

3.1.2 Neural models for diacritics correction

The experiment I conduct consists of several Transformer models with different parameters Trax frameworks provides. Other architectures (e.g. RNN, Encoder-Decoder and Encoder-Decoder with Luong Attention) due to their limitations, Transformer's proven superiority and time and resource reasons are not tested in this work. These models are more broadly compared in my preceding semestral project work.

3.1.3 Transformer

The Transformer model is build in Trax framework, which provides a relatively convenient interface for configuring the Transformer model. The parameters that can be defined include the number of encoder and decoder blocks, the embedding layer depth, intermediate fully connected layer dimension, the number of attention heads, dropout and the maximum sequence length. That's why I found it essential to create several architectures with different parameters, so it can be clear which one affect training more.

The selected Transformer parameters with the number of trainable weights are presented in table 3.1

Transformer models parameters assuming vocabulary size 212 characters					
Encoders	Decoders	Depth	FF size	Heads	**Trainable weights**
1	1	64	1024	1	**358 612**
1	1	64	2048	1	**622 804**
2	2	64	1024	1	**673 236**
1	1	256	1024	1	**2 017 492**
1	1	256	2048	1	**1 547 135**
1	1	256	1024	2	**2 017 492**
1	1	512	2048	1	**7 704 788**
2	2	512	1024	1	**10 860 756**
2	2	256	1024	1	**3 859 668**

Table 3.1: Different Transformer architectures used in all experiments with the number of trainable parameters for each of them

It is worth mentioning that the original Transformer from consists of 6 encoder-decoder blocks, which means that the number of trainable parameters in Transformer model is much bigger than in conventional recurrent neural networks. However, these parameters are represented by matrices weights in attention blocks and feed-forward layers, compared to the weights parameters in recurrent cells, which are harder to train and which suffer from vanishing gradient problem.

3.1.4 Reformer

I also train the Reformer model on the task of diacritics correction. Since the Reformer is a kind of Transformer's approximation, I will not experiment with all different parameters setup. Instead, I take the best Transformer model among those being tested and change the model architecture and check the behaviour of the Reformer compared to its full-fledged counterpart.

3.1.5 Dataset preparation

The dataset I chose to use in these experiments is SYN2015 [(ed16]. SYN2015 is a representative corpus of contemporary written Czech published in December 2015. It consists of three textual sources: fiction, non-fiction and newspapers. All three are equally presented and pre-shuffled, so that the sentences can be taken "as-is". The dataset comes in a XML format, which is not particularly convenient to work with, so the basic preprocessing and reformatting has been done in order to turn it to simple text lines. The proportion and the quality of data makes it almost ideal for using without any further steps, however in order for the network to learn correct language representation, I needed to filter out those sentences having incorrect beginning and ending.

1. Speaking of correct beginning, I found sentences with the first letter being alphanumerical character and non-capital letter. Also the sentences starting with improper punctuation characters like '[' or '!' have been identified. All incorrect sentences have been erased from the dataset.
2. The same operation has to be done with incorrectly ending sentences. The ones that have incorrect ending are those that end on some alphanumerical character or not end on '!', '.', '"', ')', "'". All these sentences have been erased from the dataset.

As it was stated earlier, I am dealing with a supervised machine learning task, which means that during training a network gets both training data and ground truth data. In my case ground truth data is pretty easy to get, since I already have it (here I assume that there is little percentage of non-diacritisized words in news articles). The way I prepare the training set is a bit different from the way it was created in [Ná17]. I assume, from real life experience, that people do not tend to write either a sentence with no diacritics or sentence with all diacritics rules obeyed. They rather use it as they wish. It's a matter of fact that Czechs finish adjectives with diacritisiced version of letter 'y' - 'ý', but do not put the same letter with diacritics when it's in the middle of the word. Therefore, the way we deal with such occasions is to randomly decide whether to erase diacritics of the particular letter or not. This peculiarity gives model a chance to learn how *correct* sentences might look like and therefore teach it to handle correctly written sentences, which is not the case of the models taught on completely "bare" sentences (these models have never seen diacritics on the input, so their behaviour is undefined or, better to say, is stochastic). The way I deal with training data has one more advantage - by cleverly choosing the way the data is loaded before being fed to the network, I have an opportunity to augment the dataset by randomly clearing diacritics from the sentences. So, by having a sentence with n letters with diacritics, I can create at least 2^n new sentences representing different instances of the same problem.

3.1.6 Training pipeline

The most important part of any machine learning project is to correctly define and implement training and validation pipeline. First of all, a dataset has to be prepared. Since the source for the diacritics correction task is well-established and representative, there is no need in rigorous dataset research that I did in my semestral project work. As I wrote in the section 3.1.5, the only additional routine I applied to all the sentences in the dataset was sentences' beginning and ending validation, so that the model knows basic rules of writing.

SYN2015 consists of slightly more than 8 million sentences, so I have split it into 7.5 million for training, 0.25 million for validation and 20 thousands for testing.

It is traditional for many machine learning supervised tasks to have several pairs of source and target files. The dataset is uploaded from the source file, processed by the model, then the classifier makes a prediction, which is compared to the appropriate entry in the target dataset. Unfortunately, many popular classification tasks including MNIST, CoNLL or regression problems like House Price Regression task have well-established and publicly recognised datasets which are pretty lightweight and can fit almost in every RAM. Deep learning problems like language modelling or summarization are the opposite case. The datasets these models are trained on are extremely large and cannot be simply held in RAM, that's why another approach is needed here.

One more reason why it is not a sensible idea to load the whole dataset into RAM resides in the way the data flows into the neural network. Neural networks are typically fed with mini-batches of a size which is evenly divided by 2. Such a requirement is placed for the hardware reasons. Mini-batches let the model simultaneously process more data and minimize loss function more optimally. These mini-batches have to be padded to the same length, so that all sequences have the same shape and the network can process them in parallel. Once each sequence is padded to the length of the longest sequence in the dataset, we have to encode each letter by a number representing its index in the alphabet. In many cases these numbers have to be one-hot encoded. All these basic rules of mini-batch preparation result in a significant memory load, so we can pretty easily end up storing enormous volume of data in the RAM. For example, if we have 500 thousands sentences each padded to the length of 250 letters and each letter is represented by an integer value (meaning each value has, let's say, 4 bytes), we have $250 * 500000 * 4 = 500000000$ bytes needed to fill it into the RAM (500 000 000 bytes are equivalent to 500MB). It might seem no problem to find free 500MB of RAM to put data in, but we also have to create one-hot encoding of each letter, meaning that if we have an alphabet containing 100 different letters, we have to multiply our 500MB by 100, which means finding **50GB** of RAM just for storing the training set. Multiply it by 2 since we also have to store the ground truth data, and it would be hard to find a machine meeting these requirements.

An effective approach how to prepare the dataset for training is to incorporate generators. Generator is a function that behaves like an iterator, i.e. it can be used in a for loop. Each iteration has to preload a mini-batch, vectorize it and pad to the same length. Generator functions solve a problem of dataset preloading at the expense of slightly longer training, since generators are considered lazy and are invoked only once needed, resulting is small I/O overhead at each forward step.

One more great advantage of using generators is the ability to augment input data. This is what I did. Instead of having source and target sentence pairs, I have only target sentences, from which I randomly wipe diacritics and use the "cleaned" sentences as input.

3.1.7 Tensorflow Dataset API

Neural networks are mostly implemented and trained in Python. Python is one of the most appropriate language for such experiments, since it is extremely easy to learn and adopt, it has a broad community and supports precompiled C++ libraries, which are vastly used when fast computing is taken into consideration.

For my experiments, I have chosen Python 3.6, created several virtual environments via mini-conda package. I decided to use Google's Tensorflow 2.0 framework for implementing and running all neural networks. There are several reasons for choosing Tensorflow:

1. It is abstractive, meaning that unless you run training, no real data or no real network is initialised, making it much simpler to experiment.
2. Since a 2.0 update, it has become much easier to work with.
3. It natively supports Keras.
4. It has the biggest community with lots of contributors and modifications.
5. Tensor2Tensor and Trax are built on it, so there is no need to install more third-party packages.
6. It comes with lots of simplifications like Estimator API and Dataset API with many basic functions each neural pipeline consists of (i.e. preprocessing data, model deployment and profiling, etc.), making all necessary preparations faster and more consistent.
7. It supports TensorBoard, which helps in monitoring model's training process.
8. The models built in Tensorflow can be universally trained on any device: CPU, GPU (see 3.1.9) or even Android devices.
9. It supports training on Google Cloud TPU, for more information see 3.1.10.
10. Distributed training is supported out of the box.

3.1.8 Accelerators

Neural networks are trained on different accelerators. Since computation power increases rapidly following Moore's law, shallow networks having several hundred thousands parameters can be trained on a single CPU on a personal computer in a couple of days. However, there are two more ways to drastically accelerate training.

3.1.9 GPU

One of the most widespread accelerator for training deep neural networks is a GPU - Graphic Processing Unit. It has been proven that these circuits are extremely effective in performing repetitive matrix multiplications and have solid bandwidth, so they are capable of processing large batches at the same time and speed up training by several magnitudes. Since most neural networks consist of parallelizable operations like dot products and convolutions, GPUs have become a standard in neural networks training.

3.1.10 Google's cloud TPU

What makes Tensorflow special is its affiliation to Google, who designed a new processing unit specifically intended to be used for fast matrix multiplication. This unit is called a TPU (Tensor Processing Unit). It is a relatively new technology, which has been internally used in Google since 2015. Thankfully, it became available for third party use via Google's cloud computing platform. This chip has been designed exclusively for Google's TensorFlow framework and is proprietary with some models commercially available.

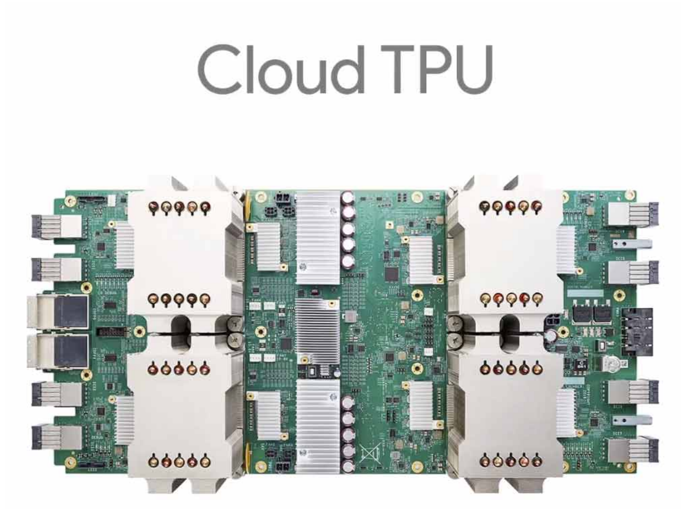


Figure 3.1: Google Cloud TPU chip, source: [goo]

Google's TPUs are much more efficient and faster in terms of training than GPUs. The network which is trained on a GPU for two weeks can reach the same accuracy while being trained on a TPU for several hours.

Training a network on a cloud TPU has its own specifics. Since cloud

TPUs are designed exceptionally for parallel computing, the data that comes to the chip has to meet certain requirements. One of the most crucial one is that all sequences have to be padded to the same length and the length has to be a multiple of 128. These are the requirements of the underlying matrix multiplication circuit that accepts 128×128 matrices. In case when a batch has another shape, a TPU has to internally convert it to the compatible shape and size, which prolongs the whole training process. Moreover, the whole network has to be pre-compiled using JAX, which is partially created for effective mathematics inside a TPU. With that said, a developer is narrowed to a short set of operations he can make use of without deteriorating training speed.

The need to pad all the dataset to the same length is a major drawback of a TPU. When trained on a GPU, the network can adjust the batch padding according to the longest sequence in a batch, not the whole dataset. Tensorflow takes advantage of this feature and is capable of shuffling sequences in a way that minimum padding is needed.

3.1.11 GPUs and TPUs availability

All the experiments that I run in this work are held in Google Colab environment. It is an online version of Jupyter Notebooks, which are widely used for experimenting in data science field. Google Colab is more advantageous than Jupyter notebooks as it is run on Google's virtual machines dedicated for Python experimenting and the notebooks are accessible via Google Drive. It means that a developer is freed from a painful process of packages installation and incompatibility issues. Almost everything is preinstalled and since it is a product from Google, there is no need to additionally install any other Google's frameworks. Nevertheless, the most important feature Google Colab comes with is GPUs and TPUs availability free of charge. Everyone can simply connect to a cloud accelerator and run experiments on them. This is the main reason why I chose Google Colab as my primal training and analysis environment. TPUs and GPUs are however preemptive, meaning that they are reserved for a short period of time (12 hours) and are primarily assigned to those who utilize them less. The reason for such a policy is to make accelerators available to everybody so that nobody can monopolise them.

Despite being free of charge, cloud TPUs available in Colab have only 8GB of RAM and are effective just in case when fully utilized. Training on a TPU usually means creating batches of size 2048 or 4096. This is not always possible. For example, running Trasformer in "big" configuration from [VSP$^+$17] is not possible. This is the reason why I run summarization experiments on Reformer model.

Due to the fact that I run many experiments and sometimes it is impossible to get a TPU assigned for 12 hours, I needed a storage where the models' checkpoints are saved. Since all the experiments are run in Google's environment, I make use of Google Cloud Platform free account with free 300 dollars credit. For storing checkpoints I use Google Cloud Storage with `gsutil` tool integrated into Colab. While training networks, I run automatic transfer of all saved data from my Colab instance (which is temporal) to the Google storage. As soon as I get a TPU assigned, I prefetch the last checkpoint with the datasets from my persistent cloud storage and continue training.

3.1.12 Tensorboard

One more advantage of running experiments in Tensorflow and via Google Colab environment is Tensorboard's integration. Tensorboard is an application run as a background process (so that it does not block Colab's I/O), which provides the visualization and tooling needed for machine learning experimentation. The key features are real-time metrics and loss tracking, model's graph visualization, profiling and weights projections. It is pretty simple to use and it drastically improves developer's insight on how the model behaves during training, what are the major metric values and whether the model converges or diverges. It is impossible to imagine training any classifier without such a tool.

Tensorboard, however, is still incapable of profiling TPU assigned via Colab, so there is a difficulty deciding how much data fits in it.

3.1.13 Dataset handling

First of all, it is important to preload datasets. This can be done in many ways with the help of Dataset API:

1. The first one is to prepare both source and target pairs in advance and store them in efficient containers called TFRecords. Tensorflow Dataset API provides an interface called `tf.Example`, which is a way of storing

and processing numerical and categorical features. It is recommended to use it together with `TFRecords`, which are responsible for examples serialization, storing, effective loading and distributing in case when there are more accelerators forming computational cluster where the data flow is usually a bottleneck. Thankfully, by storing data in `tf.Example` format, a developer does not need to carry about many peculiarities distributed training possesses. I make use of `TFRecords` while training summarization networks.

2. The second way of processing data in Tensorflow is to preload dataset into Dataset API format by calling a procedure, which transforms the generator's output to Tensorflow's internal format called Tensors. This is what is done in case of diacritics task. Tensorflow provides an extensive guide how to use *Dataset API effectively*, so that the whole training is not occasionally hampered by inefficient data preprocessing.

I opted for the second variant and after learning how to correctly operate with data, I came up with such a preparation pipeline:

Listing 3.1: Dataset preprocessing using Dataset API

```
import tensorflow as tf

dataset_train = tf.data.Dataset.from_generator(generator, (tf.
    ↪ int64, tf.int64), (tf.TensorShape([None]), tf.
    ↪ TensorShape([None])))

dataset_train.padded_batch(batch_size, padded_shapes=([max_len
    ↪ ], [max_target_len]), drop_remainder=True).repeat()
```

According to Google's guidelines, this dataset preparation pipeline is the most optimal in this particular task. Here is what is done in this code:

1. The dataset is preloaded by a generator function `generator`.
2. The generator function pre-loads a text line, applies a random dediacritization procedure described in 3.1.5, substitutes each character in both sequences with a corresponding index in the vocabulary (the vocabulary is defined as a mapping `{key: value}`, where a `key` is a character and a `value` is its index in the vocabulary). Two special characters

BEGIN_OF_SENTENCE and END_OF_SENTENCE are added as the first and the last values respectively. As a result, both Python lists of integer values are returned with a statement yield.

3. The generator function is passed as the first argument to the Tensorflow's Dataset API static method from_generator, which invokes the generator only once, controls the data types it receives against those specified in the second argument. The data types there must be from tf.dtypes.DType package. As soon as this requirement is met, the data Tensorflow gets as input can be safely cast to the underlying Tensorflow's C++ data types and the framework guarantees efficient processing. The third argument specifies generator's output shape. Tensorflow must know the shape of each item in the output tuple. When the data is of variable length, the shape has to be left undefined (e.g. defined as [None]).

4. After the dataset is pre-loaded, it is split into batches. This is done by a function padded_batch. The optional parameter drop_remainder ensures the last batch, which may not have enough data to have the predefined size, is dropped. This flag is crucial in case of TPU training, since TPU has strict shapes requirements that have to be obeyed in order to get maximum out of this accelerator.

   The function padded_batch is typically called without specifying the padded_shapes parameter the way I did. The usual way is to define each shape with [None] values and let Tensorflow find the longest sequence in the batch and pad others to its length. This is apparently done in case of GPU training. However, when your model is trained on a TPU, you still have to explicitly set these lengths.

   There is one more function that can be used both with TPUs and GPUs in case the dataset is already padded. It is called batch, which is only a lightweight version of padded_batch. Both these functions in such a case can be used interchangeably.

5. The last routine is to repeat the whole pipeline, so that as soon as the last batch is read, the dataset can be iterated over again. The function comes with an optional parameter count, specifying how many times you want to iterate over the dataset. The default value is -1 meaning that the dataset is always repeated.

All these functions are applied in this particular way and it is not recommended to change the order of their invocation unless you do not mind diminished accelerator's utilization efficiency.

3.1.14 Experiments

The models are primarily run in Trax with these notes:

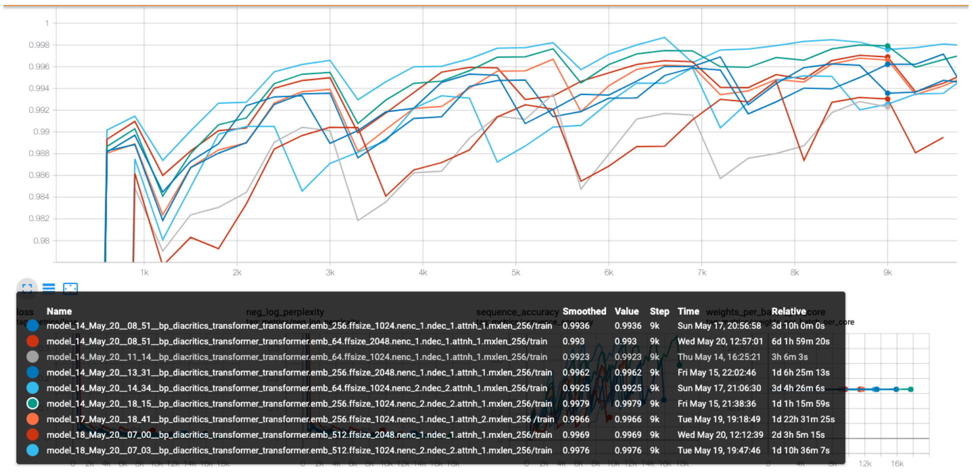
1. Tranformer model has been implemented in both Tensor2Tensor and Trax, however, after many days of attempting to make Transformer in Tensor2Tensor converge, I had no success. Many issues on Github confirm that there are bugs in hyperparameters setup and since Tensor2Tensor is not developed and is kept in maintenance mode, there was no hope anybody could have helped me with all issues I had with this framework. Many authors reporting promising results with Transformer trained it in version 1.2.9, which is too old and since there is Trax with at least stable training, I decided not to struggle with Tensor2Tensor any more. Additionally, Tensor2Tensor only supports Tensorflow 1, which is, in my opinion, excessively complicated to get acquainted with, especially when the newer Tensorflow 2 has been many times optimised and since the end of 2019 is more stable and reliable. I hope all my attempts and efforts to fix Trax's bugs will be useful and this library will become better.
2. Reformer model is also implemented in Trax. This is one more reason why I chose Trax. Despite having many bugs and memory issues, Trax is what is going to be more popular in upcoming months, so it seemed to me that it was a right moment to try it out. Additionally, in order to change the model's architecture from Transformer to Reformer, you only need to slightly change import statements, which is what I loved about Trax.

The final dataset after filtering all incorrect sentences had

1. 6 186 214 sentences for training
2. 210 790 sentences for validation
3. 20 000 sentences for testing

Other training settings like optimizer, initial learning rate, warmup steps after several experiments were set to their default values as they seem to be the most optimal for selected architectures.

Here is how all the transformer models look like considering training accuracy, see figure 3.2.



	Name	Smoothed	Value	Step	Time	Relative
●	model_14_May_20__08_51__bp_diacritics_transformer_transformer.emb_256.ffsize_1024.nenc_1.ndec_1.attnh_1.mxlen_256/train	0.9936	0.9936	9k	Sun May 17, 20:56:58	3d 10h 0m 0s
●	model_14_May_20__08_51__bp_diacritics_transformer_transformer.emb_64.ffsize_2048.nenc_1.ndec_1.attnh_1.mxlen_256/train	0.993	0.993	9k	Wed May 20, 12:57:01	6d 1h 59m 20s
●	model_14_May_20__11_14__bp_diacritics_transformer_transformer.emb_64.ffsize_1024.nenc_1.ndec_1.attnh_1.mxlen_256/train	0.9923	0.9923	9k	Thu May 14, 16:25:21	3h 6m 3s
●	model_14_May_20__13_31__bp_diacritics_transformer_transformer.emb_256.ffsize_2048.nenc_1.ndec_1.attnh_1.mxlen_256/train	0.9962	0.9962	9k	Fri May 15, 22:02:46	1d 6h 25m 13s
●	model_14_May_20__14_34__bp_diacritics_transformer_transformer.emb_64.ffsize_1024.nenc_2.ndec_2.attnh_1.mxlen_256/train	0.9925	0.9925	9k	Sun May 17, 21:06:30	3d 4h 26m 6s
●	model_14_May_20__18_15__bp_diacritics_transformer_transformer.emb_256.ffsize_1024.nenc_2.ndec_2.attnh_1.mxlen_256/train	0.9979	0.9979	9k	Fri May 15, 21:38:36	1d 1h 15m 59s
●	model_17_May_20__18_41__bp_diacritics_transformer_transformer.emb_256.ffsize_1024.nenc_1.ndec_1.attnh_2.mxlen_256 /train	0.9966	0.9966	9k	Tue May 19, 19:18:49	1d 22h 31m 25s
●	model_18_May_20__07_00__bp_diacritics_transformer_transformer.emb_512.ffsize_2048.nenc_1.ndec_1.attnh_1.mxlen_256/train	0.9969	0.9969	9k	Wed May 20, 12:12:39	2d 3h 5m 15s
●	model_18_May_20__07_03__bp_diacritics_transformer_transformer.emb_512.ffsize_1024.nenc_2.ndec_2.attnh_1.mxlen_256/train	0.9976	0.9976	9k	Tue May 19, 19:47:46	1d 10h 36m 7s

Figure 3.2: Transformer models training accuracy in Tensorboard UI

It is better seen from the tabular perspective in 3.2.

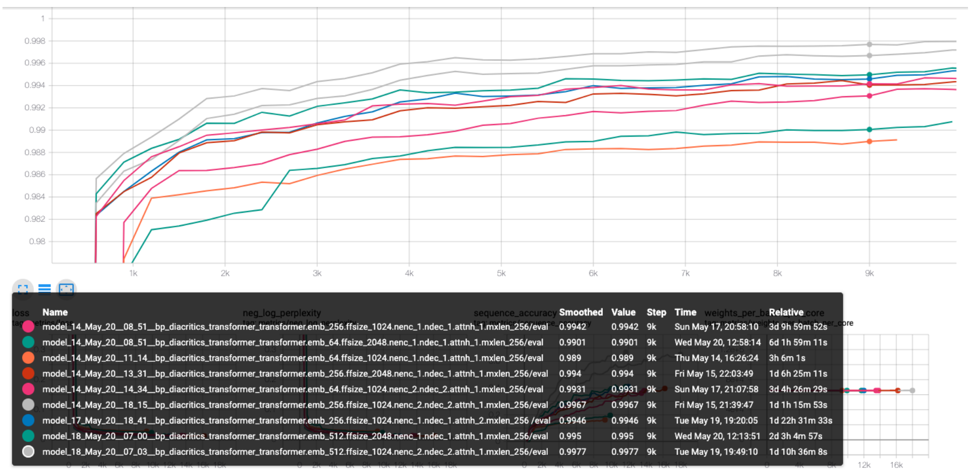
Transformer diacritics models - training accuracy							
Enc	Dec	Depth	FF size	Heads	3K	6K	9K
1	1	64	1024	1	0.9869	0.9897	0.9915
1	1	256	1024	1	0.9901	0.9931	0.9949
1	1	64	2048	1	0.9864	0.9880	0.9915
1	1	256	2048	1	0.9915	0.9927	0.9950
1	1	512	2048	1	0.9930	0.9950	0.9965
1	1	256	1024	2	0.9917	0.9941	0.9962
2	2	64	1024	1	0.9856	0.9912	0.9931
2	2	256	1024	1	0.9934	0.9960	0.9976
2	2	512	1024	1	**0.9947**	**0.9969**	**0.9980**

Table 3.2: Different diacritics models accuracy scores on the training set after 3K, 6K and 9K training steps

Evaluation metrics are depicted in the figure 3.3

3.1.15 Results discussion

There is no particular difference in training times, each epoch consists of 300 train steps and 102 evaluation steps, so that the model covers the whole training dataset in 3 epochs, whereas after each epoch is evaluated on the whole dev set. 9K steps from the table mean that the model ran 30 epochs, iterating over our huge training dataset for 10 times. It is sufficient to evaluate all models' performance after several full iterations since the overall trend



	Name	Smoothed	Value	Step	Time	Relative
●	model_14_May_20__08_51__bp_diacritics_transformer_transformer.emb_256.ffsize_1024.nenc_1.ndec_1.attnh_1.mxlen_256/eval	0.9942	0.9942	9k	Sun May 17, 20:58:10	3d 9h 59m 52s
●	model_14_May_20__08_51__bp_diacritics_transformer_transformer.emb_64.ffsize_2048.nenc_1.ndec_1.attnh_1.mxlen_256/eval	0.9901	0.9901	9k	Wed May 20, 12:58:14	6d 1h 59m 11s
●	model_14_May_20__11_14__bp_diacritics_transformer_transformer.emb_64.ffsize_1024.nenc_1.ndec_1.attnh_1.mxlen_256/eval	0.989	0.989	9k	Thu May 14, 16:26:24	3h 6m 1s
●	model_14_May_20__13_31__bp_diacritics_transformer_transformer.emb_256.ffsize_2048.nenc_1.ndec_1.attnh_1.mxlen_256/eval	0.994	0.994	9k	Fri May 15, 22:03:49	1d 6h 25m 11s
●	model_14_May_20__14_34__bp_diacritics_transformer_transformer.emb_64.ffsize_1024.nenc_2.ndec_2.attnh_1.mxlen_256/eval	0.9931	0.9931	9k	Sun May 17, 21:07:58	3d 4h 26m 29s
●	model_14_May_20__18_15__bp_diacritics_transformer_transformer.emb_256.ffsize_1024.nenc_2.ndec_2.attnh_1.mxlen_256/eval	0.9967	0.9967	9k	Fri May 15, 21:39:47	1d 1h 15m 53s
●	model_17_May_20__18_41__bp_diacritics_transformer_transformer.emb_256.ffsize_1024.nenc_1.ndec_1.attnh_2.mxlen_256 /eval	0.9946	0.9946	9k	Tue May 19, 19:20:05	1d 22h 31m 33s
●	model_18_May_20__07_00__bp_diacritics_transformer_transformer.emb_512.ffsize_2048.nenc_1.ndec_1.attnh_1.mxlen_256/eval	0.995	0.995	9k	Wed May 20, 12:13:51	2d 3h 4m 57s
○	model_18_May_20__07_03__bp_diacritics_transformer_transformer.emb_512.ffsize_1024.nenc_2.ndec_2.attnh_1.mxlen_256/eval	0.9977	0.9977	9k	Tue May 19, 19:49:10	1d 10h 36m 8s

Figure 3.3: Transformer models validation accuracy in Tensorboard UI

Transformer diacritics models - evaluation accuracy							
Enc	Dec	Depth	FF size	Heads	3K	6K	9K
1	1	64	1024	1	0.9831	0.9881	0.9889
1	1	256	1024	1	0.9892	0.9934	0.9940
1	1	64	2048	1	0.9828	0.9888	0.9901
1	1	256	2048	1	0.9889	0.9927	0.9941
1	1	512	2048	1	0.9905	0.9942	0.9949
1	1	256	1024	2	0.9890	0.9935	0.9946
2	2	64	1024	1	0.9854	0.9912	0.9928
2	2	256	1024	1	0.9912	0.9954	0.9966
2	2	512	1024	1	**0.9928**	**0.9966**	**0.9976**

Table 3.3: Different diacritics models accuracy scores on the evaluation set after 3K, 6K and 9K training steps

is apparent. As the the model becomess more complex in its architecture and have more parameters, there is still some room for further improvement. However, we are still dealing with quite simple task, that's why it is not always the final accuracy value that matters most.

Moreover, the graphs 3.3 above show that even though some setups may be less accurate at earlier stages, they become better over time. It is a kind of a trade-off between the training speed and thus the price for buying GPU/TPU hours needed for the training and the model's complexity. These have a tendency to learn faster when having more trainable weights. On the other side, as the model grows in its parameters number, it runs longer and takes more space on disk, which is in many cases critical.

With that in mind, it depends on how the model is going to be utilized after being trained. Whether it is supposed to run on a low-performance device and thus the additional restriction is the model's size on disk and in RAM, or it is served in a cloud container (e.g. Microsoft Azure, GCP, AWS) and these requirements are relaxed. So, not only the final result matters, but also the price we pay for achieving it.

3.1.16 Best model analysis

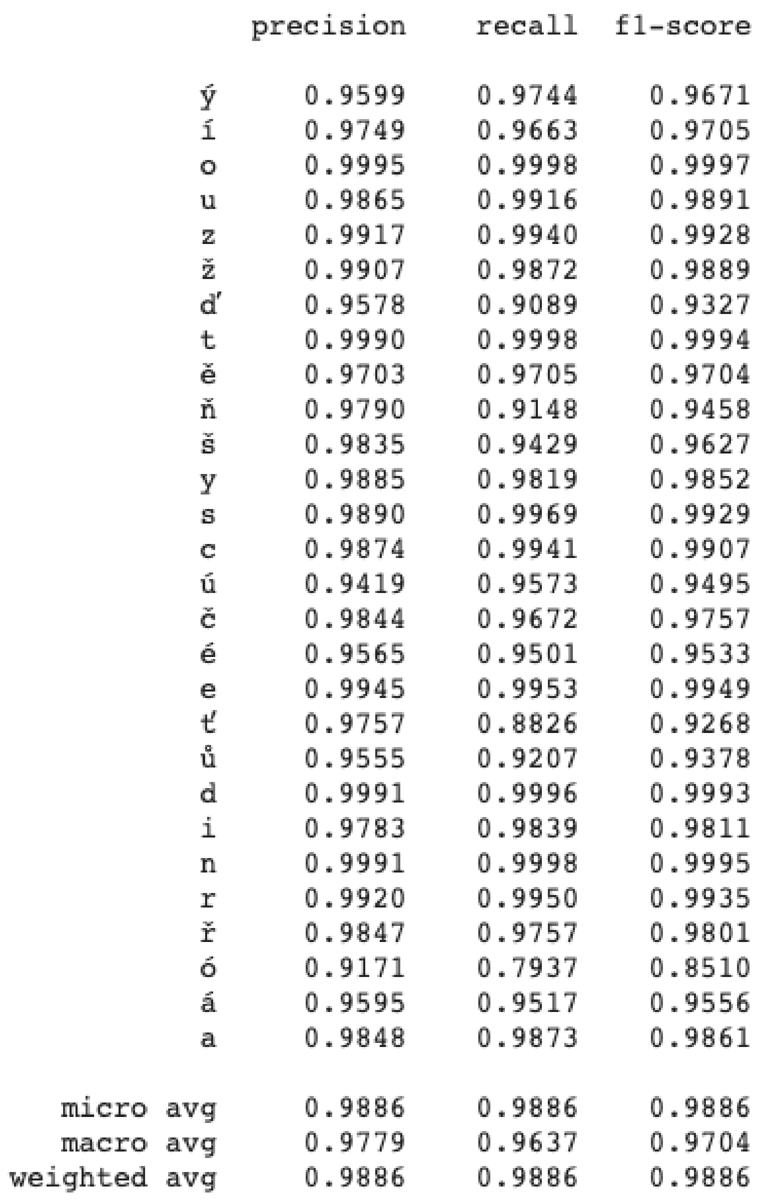
The next leg of the diacritics experiment is to take a look at more statistics the model provides. I have taken the best one so far (having `2 - 2 - 512 - 1024 - 1` setup from the table 3.1) and ran a small benchmark on the test set. Since the task of diacritics correction is mostly about copying those letters having no diacritised equivalent, in the table 3.4 I show only those letters that were mistakenly misclassified. Note that there are no letters having no diacritisized equivalent in the report, which indicates that the model perfectly copies every input character and do not mix them. In my opinion, since the model embedded all input letters to the linear subspace of higher dimension than the length of the alphabet, it learned how to map them unambiguously.

I also calculate confusion matrix 3.5 of those misclassified letters in order to take a look at what was the mistake.

There can be seen from 3.5 that the model really learned how to "copy" all input letter to their equivalents to the output. In case of mistake we can be sure that the model does not yield another erroneous letter. This is not the case for models with significantly slower embedding size, which sounds logical - the model struggles with projecting several hundreds letters to a subspace of dimension less than the size of the vocabulary. Hence, a letter represented by some vector can be considered to be a linear combination of the vectors from some basis.

3.1.17 Attention weights visualisation

It is not an easy task to visualise attention in Trax. The problem is that Trax does not store internal attention alignment while being trained or run in inference mode. However, it is possible to dig into the code and make the attention layers store the alignments which could be then read out from the model's properties. The model in Trax consists of two basic wrappers: `Serial` and `Parallel`. They tell the JAX compiler how to handle them while



```
                precision    recall  f1-score

           ý       0.9599    0.9744    0.9671
           í       0.9749    0.9663    0.9705
           o       0.9995    0.9998    0.9997
           u       0.9865    0.9916    0.9891
           z       0.9917    0.9940    0.9928
           ž       0.9907    0.9872    0.9889
           ď       0.9578    0.9089    0.9327
           t       0.9990    0.9998    0.9994
           ě       0.9703    0.9705    0.9704
           ň       0.9790    0.9148    0.9458
           š       0.9835    0.9429    0.9627
           y       0.9885    0.9819    0.9852
           s       0.9890    0.9969    0.9929
           c       0.9874    0.9941    0.9907
           ú       0.9419    0.9573    0.9495
           č       0.9844    0.9672    0.9757
           é       0.9565    0.9501    0.9533
           e       0.9945    0.9953    0.9949
           ť       0.9757    0.8826    0.9268
           ů       0.9555    0.9207    0.9378
           d       0.9991    0.9996    0.9993
           i       0.9783    0.9839    0.9811
           n       0.9991    0.9998    0.9995
           r       0.9920    0.9950    0.9935
           ř       0.9847    0.9757    0.9801
           ó       0.9171    0.7937    0.8510
           á       0.9595    0.9517    0.9556
           a       0.9848    0.9873    0.9861

   micro avg       0.9886    0.9886    0.9886
   macro avg       0.9779    0.9637    0.9704
weighted avg       0.9886    0.9886    0.9886
```

Figure 3.4: Misclassified letters and their scores

been run on different accelerators. These wrappers contain a computational graph which is accessible from the outside. That's why I added several necessary lines to the framework so that it saves attention weights while run in inference mode. The reason to save attention alignments only during inference is straightforward - Transformer model is enormously large and may not fit into RAM of the accelerator it is trained on. Since attention weights are formally a $N \times N$ matrix, where N is the length of a sequence, it is really hard to store additional floats for each batch. Moreover, there are 3 attention modules in one encoder-decoder setup. Moreover, it is useless to save attention weights while training, unless there is no specific need to

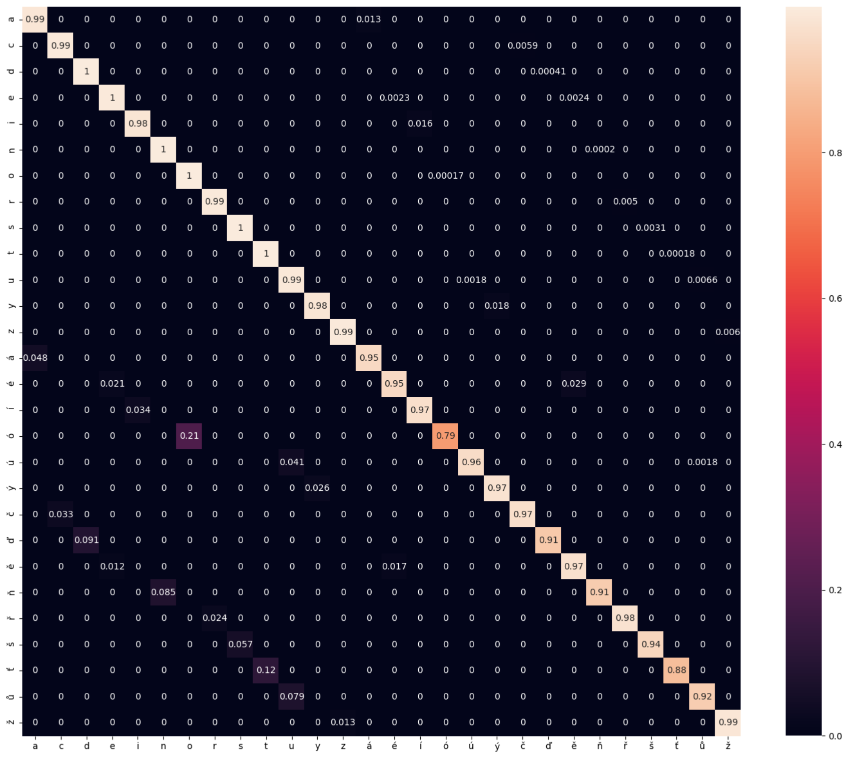


Figure 3.5: Confusion matrix for misclassified labels

track the training process by looking at how attention alignments change over time, which could be beneficial for understanding how backpropagation really works in such architecture.

As stated above, there are several attention modules in Transformer. In Trax, there are two main classes responsible for calculating attention:

1. `trax.layers.PureAttention`
2. `trax.layers.DotProductCausalAttention`

I modified them by introducing new class properties, which are then read after each model inference step. The reading itself is a tedious process which looks like shown in listing 3.2:

Listing 3.2: Attention layer object instance destination is Trax's Transformer model

```
alignments = encoder_blocks[0]._sublayers[0]._sublayers[-1].
    ↪ _sublayers[-1]._sublayers[1]._sublayers[-1]._sublayers
    ↪ [1].alignment[0, :, :, :]
```

I chose to show the best Transformer model with 2 encoder-decoder blocks and 1 attention head. Since there are two decoding blocks, there will be two different views on the generated sequence. The attention I show is the one which has both keys and values coming from the encoder, so that we see how the model really makes use of the encoded information.

An alignment matrix A is a $N \times M$ matrix, where N is the length of an input sequence and M is the length of the output sequence. For each token $i, i \in [0, M]$ the cell $a_{j,i}, j \in [0, N]$ shows how much attention is given to the jth input token for making the final decision.

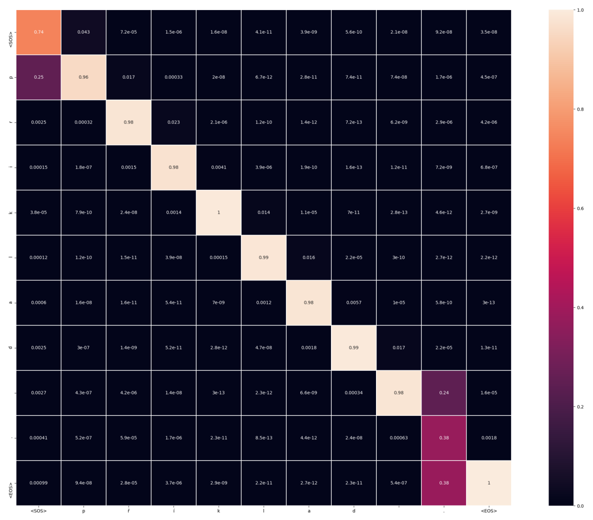


Figure 3.6: Attention weights from the first decoder block for sentence "priklad"

The input string is "priklad", which has to be transformed to "příklad" (Engl. example). First of all, by taking a look at the first (lower) decoder block 3.6, we see that the model basically attends to the same letter in the encoder. However, by looking at the same module in the second (upper) decoder block 3.7, we notice how the model applies context of the letter it tries to generate. This behaviour corresponds to what is thought to be typical for deep neural networks - more shallow layers learn the basic rules, whereas those on top of them operate with more complex definitions and abstractions. This is exactly what happens inside our model.

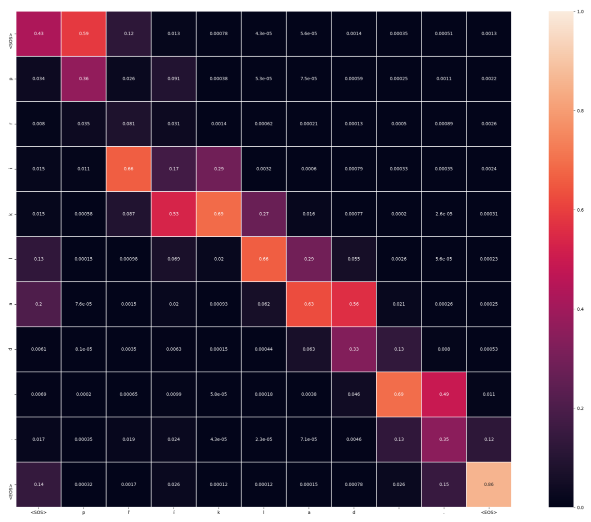


Figure 3.7: Attention weights from the second decoder block for sentence "priklad"

The topic of looking inside the model's is very popular at the moment. One of the most prominent paper about how the attention works in Transformer is "What Does BERT Look At? An Analysis of BERT's Attention" [CKLM19].

3.1.18 Training Transformer in Trax with TF Dataset loaded from generators

For some unknown reason, while the model was training on a TPU, it crashed without even outputting any error message or stack trace. It always happened after 20-25 epochs and it is not connected to Colab's usage and idle notebooks restrictions. Even when I trained iteratively for 15 epochs and then reloaded my notebook, the same problem occurred after 5-8 epochs. The whole notebook crashed and all temporary data was erased. Colab provides developer with runtime logs, however they are updated with a huge timeout and moreover, as the notebook crashed unexpectedly, the runtime log was no more available.

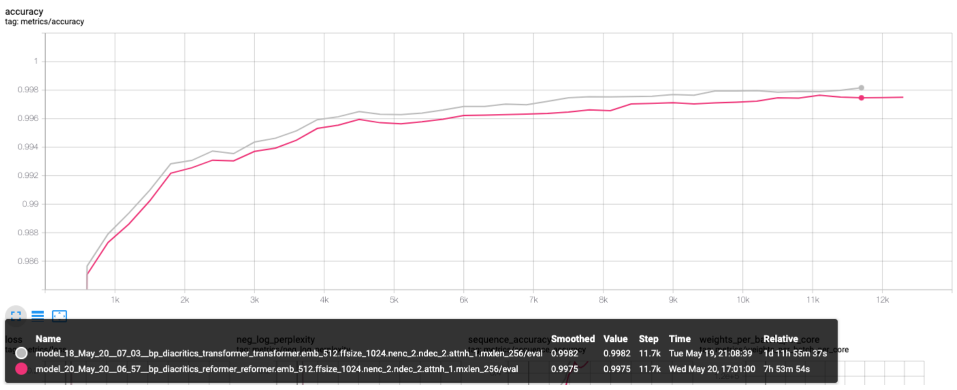
My opinion is that there is an inner bug in Trax's logic of data handling and model's compilation. To be trained on a TPU and to take advantage of all its computational power, the model has to meet strict requirements even in the set of mathematical operations that have to be adapted to be efficiently run in such a specific environment. Taking into consideration the relative novelty of this framework and the novel approach to compile models, I assume there is one more bug I cannot even trace. I have opened an issue on their GitHub page and duplicated it to the Trax's Gitter chat.

I managed to work around this bug by saving checkpoints after each training epoch to my free Google Storage bucket. After the environment crashed, I simply downloaded my model from this storage and continued training from the last saved checkpoint.

3.1.19 Reformer model

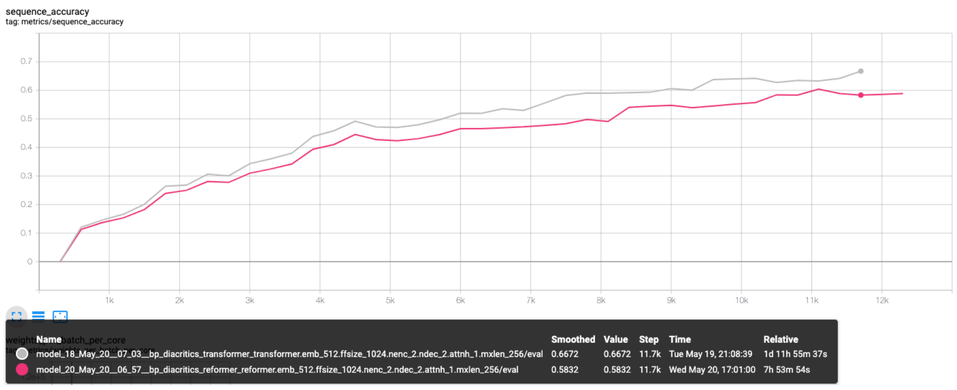
I also trained the Reformer model with the same hyperparameters as the best Transformer model from 3.1 in terms of evaluation accuracy (see 3.3). The graph 3.8 shows the difference in their final accuracy on the evaluation set. Moreover, I show sequence accuracy metric, which particularly interesting in the context of this task, in figure 3.9

Apparently, the Reformer model behaves exactly according to the results from [KKL20] - it is an approximation of Transformer, however, which is easier to train and fit into RAM.



	Name	Smoothed	Value	Step	Time	Relative
⚪	model_18_May_20__07_03__bp_diacritics_transformer_transformer.emb_512.ffsize_1024.nenc_2.ndec_2.attnh_1.mxlen_256/eval	0.9982	0.9982	11.7k	Tue May 19, 21:08:39	1d 11h 55m 37s
🔴	model_20_May_20__06_57__bp_diacritics_reformer_reformer.emb_512.ffsize_1024.nenc_2.ndec_2.attnh_1.mxlen_256/eval	0.9975	0.9975	11.7k	Wed May 20, 17:01:00	7h 53m 54s

Figure 3.8: Reformer model compared to its Transformer counterpart having the best evaluation accuracy with the same hyperparameters in terms of classical accuracy.



	Name	Smoothed	Value	Step	Time	Relative
⚪	model_18_May_20__07_03__bp_diacritics_transformer_transformer.emb_512.ffsize_1024.nenc_2.ndec_2.attnh_1.mxlen_256/eval	0.6672	0.6672	11.7k	Tue May 19, 21:08:39	1d 11h 55m 37s
🔴	model_20_May_20__06_57__bp_diacritics_reformer_reformer.emb_512.ffsize_1024.nenc_2.ndec_2.attnh_1.mxlen_256/eval	0.5832	0.5832	11.7k	Wed May 20, 17:01:00	7h 53m 54s

Figure 3.9: Reformer model compared to its Transformer counterpart having the best evaluation accuracy with the same hyperparameters in terms of sequence accuracy.

3.1.20 Final thoughts on diacritics correction task

The task of diacritics correction seems simple even for less sophisticated architectures like many-to-many LSTM network, which was trained and tested in my semestral project work. However, the idea of attention behind Transformer makes it possible to think about this task in another way, which I consider as the most important point.

However, Transformer model shows that it is more universal and is capable of solving many sequential tasks. The biggest challenge is to formulate the problem so that it can be solved by Transformer, which has been successfully done and the results of the trained models look promising, providing that the networks were trained for several hours instead of days or weeks.

To sum up, even though Transformer has this capability to solve many different tasks, its potential is more evident while applied to more complex tasks like text summarization, which is the second part of this thesis (see 3.2), where I train Reformer model and compare its performance against previous results.

3.2 Text summarization

Another task which is considered to be a machine translation problem is text summarization. Summarization is an approach to fit a large piece of text into a small concise set of sentences called *summary*, preserving key information and overall meaning. It is an extremely challenging task which has not been solved to a proper level even in English yet. The task, in contrast to diacritics correction, is more complicated and has many forms which are differently defined and hence accomplished.

There are two different variants of text summarization.

1. **Extractive summarization** is a task of selecting the most appropriate pieces of text (typically sentences) carrying most crucial information from the original text. This problem can be seen as a classification problem, precisely assigning each sentence a label "yes" or "no", meaning whether the sentence is enough concise to be included in the summary. The methods how to accomplish this problem exist from the end of the 20th century and include such prominent and well-posed algorithms as Latent Semantic Analysis [SJ04], TextRank algorithm [Mih04], topics representations and frequency analysis (TF-IDF [HL98]), Bayesian Topics Model [Nom05], graph methods [THP08] and machine learning approaches. Interestingly, even language modelling approaches have been tried on the extractive summarization problem [KMTD14].

2. **Abstractive summarization** is an advanced way to create summaries based on text generation approach and general language understanding. It is far more sophisticated and hence unexplored concept, which interested me and which is primarily demonstrated in this work. To make a concise summary, a model must show signs of orienting in the language. It has to be able to do *coreference resolution*, understand, what is been narrated about in the sentence, paragraph, or in the whole text. What's more, a model must be able to hold long-term dependencies and be consistent. That's why this is not the task for shallow networks.

3.2.1 Abstractive summarization - task definition

The task of abstractive text summarization can be formalised in a following way. Given a list of sentences $\mathbf{S} = [s_1, s_2, ..., s_n], n \in \mathbf{N}$ representing a text piece to write summary of, create another list of sentences $\mathbf{S'} = [s'_1, s'_2, ..., s'_k], k \in \mathbf{N}$ representing its summary.

This formulation may sound trivial, it is still important to define a sentence. There are three ways how to do it:

1. $s = [c_1, c_2, c_3, ..., c_m]$, where $c_i \in \mathbf{A}, i \in 1, 2, ..., m$. Here, s is a list of characters c_i from an alphabet $\mathbf{A}$, m is the length of the list.
2. $s = [w_1, w_2, ..., w_m]$, where $w_i \in \mathbf{V}, i \in 1, 2, ..., m$. Here, s is a list of words w_i from a vocabulary $\mathbf{V}$, m is the length of the list and the sentence length at the same time.
3. $s = [p_1, p_2, ..., p_m]$, where $p_i \in \mathbf{W}, i \in 1, 2, ..., m$. Here, s is a list of word-pieces p_i from a word-piece vocabulary $\mathbf{V}$, m is the length of the list and the sentence length at the same time.

3.2.2 Definition based on data representation

The first and the second formulations sound logical from different perspectives. We can handle a sentence either as a sequence of letters or a sequence of words. For the task of summarization, though, the second definition could be more accurate for two reasons - when we operate with single words, the model can better learn the dependencies on a word level, whereas while trained with a character-level representation, the challenge is to firstly understand what a word is and how it is built. Moreover, the primal task is to learn the semantics in the text, although the lexical structure of the words also matters. On the other hand, the model is trained on a long list of sentences, which means that in case of character-level representation the sequences themselves will be significantly longer and may not fit in GPU/TPU memory or the whole training will take much longer.

Another argument against word-level representation lies in its poor flexibility. As soon as the model encounters unseen word, it is not handled properly and the information this particular word carries will be not taken into consideration, which is in the light of summarization exceptionally vital. Imagine a long article with many names of people or companies mentioned. A summarization model trained on a word level will probably not find many of these names in its vocabulary and will assign an `UNKNOWN` label to each word it has not seen yet. This is the case when the most essential information is getting lost even before entering the model. A character level model, however, will not struggle with it, since the probability of encountering an unknown letter is significantly lower. Still it has to be intensively trained to understand what a name is.

3.2.3 Word pieces

In the light of the aforementioned, both methods seem illogical to utilise in summarization models. There is however a third way how to handle data, which combines both of the described methods and attempts to take all their advantages. The third formulation almost mirrors the second one by effectively solving the problem of Out-Of-Vocabulary (OOV) words. Still, the major difference lies in the vocabulary used here. The best way to connect both word level and character level representations is to find a trade-off somewhere in the middle - in the word pieces. Word piece method is an approximation of both word-level and character-level representations, which operates with a vocabulary of a pre-defined size N. A word piece representation splits each word in a set of smaller chunks that are presented in the vocabulary. For example, the word "nejvhodnějšího" (Eng. the most appropriate) could be split into chunks "nej", "vhodn", "ější", "ho". Two signs in front of tokens starting from the second position indicate that this piece belongs to the piece from the left, so that the word can be reconstructed after being split.

A program dealing with word pieces is called `a tokenizer`. A tokenizer is responsible for building the word-piece vocabulary, splitting an input sentence into words and then words into word pieces. It has to be able to rebuild the sentence from the list of its word pieces.

There are many ways a tokenizer can build the vocabulary. This routine is called *training a tokenizer*. There are several widely used training algorithms, with many of them having a Byte Pair Encoding algorithm under the hood [SHB16]

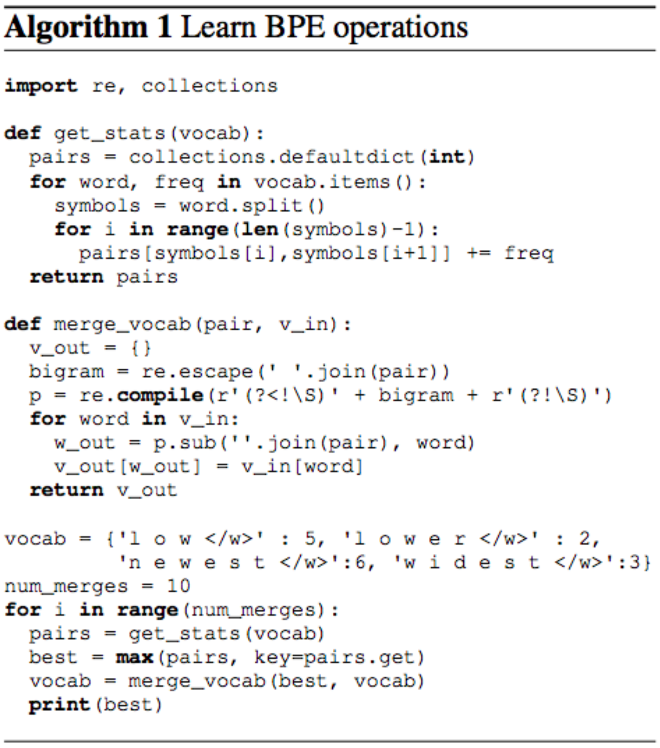
3.2.4 Byte Pair Encoding algorithm

Byte Pair Encoding algorithm consists of several steps:

1. Prepare the largest dataset of textual data possible.

2. Define the size of the word piece vocabulary to build.
3. Split each word into a sequence of characters.
4. Append suffix `</w>` to the end of every word with word's frequency, which is computed over the dataset from step 1. For example, given the frequency of "dobrý" (Engl. good) is 5, the algorithm transforms it to `"d o b r ý </w>": 5`
5. Generate a subword according to the highest frequency.
6. Repeat the previsou step until reaching subword vocabulary size which is defined in step 2 or the next highest frequency pair is 1.

The algorithm is depicted in figure 3.10.



Algorithm 1 Learn BPE operations

```
import re, collections

def get_stats(vocab):
  pairs = collections.defaultdict(int)
  for word, freq in vocab.items():
    symbols = word.split()
    for i in range(len(symbols)-1):
      pairs[symbols[i],symbols[i+1]] += freq
  return pairs

def merge_vocab(pair, v_in):
  v_out = {}
  bigram = re.escape(' '.join(pair))
  p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
  for word in v_in:
    w_out = p.sub(''.join(pair), word)
    v_out[w_out] = v_in[word]
  return v_out

vocab = {'l o w </w>' : 5, 'l o w e r </w>' : 2,
         'n e w e s t </w>':6, 'w i d e s t </w>':3}
num_merges = 10
for i in range(num_merges):
  pairs = get_stats(vocab)
  best = max(pairs, key=pairs.get)
  vocab = merge_vocab(best, vocab)
  print(best)
```

Figure 3.10: BPE algorithm code listing. Source: [SHB16]

The idea of BPE lies in WordPiece algorithm [WSC+16b], which is widely used in language modelling tasks like BERT and which I chose for tokenization.

The algorithm consists of these steps:

1. Do steps 1-3 from BPE.

2. Build a language model on the character data.
3. Choose the new word unit out of all the possible ones that increases the *likelihood* on the training data the most when added to the model.
4. Go to step 2 until a predefined limit of word units is reached or the likelihood increase falls below a certain threshold.

For my summarization experiments, I decided to use the same tokenizer used in training BERT model with WordPiece algorithm with vocabulary size of 30000 subwords. Its training and usage is made simple by an open-source library `transformers` implemented by HuggingFace. The advantage of using it instead of own implementation is that all tokenizers in `transformers` library are implemented in Rust and are extremely fast to train and run. Moreover, they are easily integrated to the sequence processing pipelines thanks to lots of useful features which make the whole dataset preparation process much easier.

3.2.5 Summarization dataset

There are a few small datasets with human-generated summaries available in Czech language for accomplishing summarization task: Czech part of the MultiLing dataset [GCK+17] consisting of 40 WIkipedia articles, and SummEC [RČ13] with 50 news articles. In 2018, Institute of Formal and Applied Linguistics released SumeCzech - Large Czech News-Based Summarization Dataset [SMK+18] containing over 1 million news articles with aggregated summaries from 5 different sources. This dataset was obtained with a semi-automatic processing of news articles, abstracts and headlines. Unfortunately, it does not contain human-generated summaries and heavily relies on the abstracts. The assumption made by the authors is that an abstract in some way holds the article's summary, however, this is a really strong assumption which is not always satisfied, since the abstract's partial aim is to provoke reader's interest to read the whole text. That's why my opinion is that any metrics calculated on such data have to be taken with a grain of salt. Nevertheless, it is still the only source of large amount of data available to run experiments on with pretty feasible basis, so hopefully there will be more datasets in the nearest future, as the problem of summarization becomes more urgent.

The SumeCzech dataset is stored in JSON line format with each line containing a headline, an abstract and the article itself, which makes 6 different pairs of textual data and hence 6 different summarization tasks. The authors introduced 3 setups:

1. abstract to headline - generate a one-sentence headline basing on several sentences in the abstract.
2. full text to head line - generate a one-sentence headline basing on the whole article
3. full text to abstract - generate a multi-sentence summary of the whole article.

It is also important to notice the size of each of three integrants of this dataset. The authors provide the following statistics:

	Q1	Median	Q3	Mean	Stddev
Headlines	7	9	11	9.4	2.9
Abstracts	33	42	51	42.2	14.8
Texts	265	378	553	470.1	365.3

Table 3.4: Statistics of lengths of headlines, abstracts and texts in words. Q1 and Q3 are the first and the third quartile, respectively.

It can be seen that all three items follow normal Gaussian distribution.

The authors of SumeCzech report that they trained Transformer from [VSP$^+$17] on the simplest (in terms of data size) setup on a single Nvidia GTX 1080 Ti for 15 days with batch size of 1700. Unfortunately, no more precise information is available, so the throughput, the number of steps and epochs for this experiment remain unknown and can be only estimated basing on the recommendations from the ÚFAL's parer with Transformer training tips [PB18]. The authors claim, that the throughput of the Transformer 'base' setup on English-to-German translation problem for a batch of size 1500 is 33.4M subwords/hour and for a batch of size 2000 is 33.7M subwords/hour on a single GPU. It means that for a batch of size 1700 the throughput has to be approximately 33.5 subwords/hour. However, the tasks differ significantly. While the English-to-German translation task consists of 2 sentences for each example, in the context of summarization there are approximately 3 sentences for a typical abstract and 1 sentence for a headline. For the sake of simplicity I dropped the throughput of the Transformer model on summarization on 30M subwords/hour. Then, the authors state that dataset tokenization roughly enlarges each example by a factor of 1.5, so the assumed throughput of 30M subwords/hour is a highly optimistic value.

The batch size of 1700 claimed in the paper does not specify whether there were 1700 abstract-headline pairs or just 1700 subwords as the authors of [PB18] state. I assumed the latest, because Transformer with SumeCzech has been trained in Tensor2Tensor framework, so this size might reflect the batch logic applied in the framework.

By doing simple calculations, I came to the following result: assuming the average length of an abstract is 42 words, the tokenized version has an average length of 63 subwords. Then, it means that a single batch contains $1700/63 \approx 26$ abstract-headline pairs. Given the throughput calculated above, I suppose the model could process $30000000/1700 \approx 17647$ batches per hour, meaning that it processed $17647 * 26 = 458822$ abstract-headline pairs per hour. This implies approximately 165M abstract-headline pairs for the experiment. Since the training set consists of 867596 such pairs, the dataset has been fully iterated for 190 times.

That's why, I decided to experiment only with "abstract to headline" setup for the purposes of this work and due to many GPU/TPU restrictions in Google Colab environment.

3.2.6 Previous works

To my best knowledge, no other attempts have been made to solve summarization tasks in Czech language except for [SMK^{+}18]. There was no feasible dataset in Czech language to train models on, which is the main reason of the absence of other works.

3.2.7 Metrics

There are several evaluation metrics widely used in translation tasks. The most popular one is ROUGE [Lin04]. The name ROUGE stands for `Recall-Oriented Understudy for Gisting Evaluation`. This metric compares summaries or translations generated by an evaluated model against a human-written reference summary or translation. The basic idea behind ROUGE is to measure the number of overlapping words in both texts with respect to different sources. In particular, ROUGE metric makes use of two scores:

1. Precision score in the context of ROUGE is defined in the following way:

$$precision = \frac{number\_of\_overlapping\_words}{total\_number\_of\_words\_in\_system\_summary}$$

   The motivation to calculate precision is to understand how many *unnecessary* words are added to the summary. Simply put, if a model copies the whole input text to the output proclaiming that this is a summary, the precision score will approach zero, since the denominator of the precision formula is large. Another way of thinking about precision is to ask how many words in the system's summary are relevant.

2. Recall score in terms of ROUGE is defined in the following way:

$$recall = \frac{number\_of\_overlapping\_words}{number\_of\_words\_in\_the\_reference\_summary}$$

   The aim of the recall score is to show how much of the reference summary is covered by the system's summary. This quantity reflects whether the information presented in the system's summary is concise and cover the main idea of the text.

It is obvious that both scores have to go hand-in-hand, since it is pretty easy to fool each of them separately. That's why the main indicator of the summarization (and translation) quality is the harmonic mean of both precision and recall, which is `F1 score` and is defined as follows:

$$F1 = 2 * \frac{precision * recall}{precision + recall}$$

In case when summaries are forced to be as concise as possible, only recall is used to evaluate the system's performance.

ROUGE metric is in fact a set of several similar metrics with a different degree of strictness to the model's output:

1. ROUGE-N - this metric calculates an overlap of n-grams of length **N**. In case **N = 1** the metric is equivalent to calculation of an overlap of words. Otherwise an overlap of **N** consecutive words is taken into consideration.
2. ROUGE-L - measures an overlap in terms of Longest Common Subsequence (LCS) problem, i.e. the longest matching sequence. The advantage of this metric is that it measures sentence level structure similarity that reflect the word order. Moreover, it includes longest in-sequence common n-grams.

3. ROUGE-W - a modification of the preceding metric which weights the ROUGE-L score in favor of consecutive LCS.
4. ROUGE-S - a skip-gram n-gram metric model which allows arbitrary gaps between words. It is typically called skip-gram concurrence. For example, skip-bigram measures the overlap of word pairs that can have a maximum of two gaps in between.

The SumeCzech dataset comes with a script for summaries evaluation which consists of a full-length language-agnostic F1 ROUGE metric with no stemmer or stop words implemented. This metric is denoted as $ROUGE_{raw} - N$ and $ROUGE_{raw} - L$ with equivalent meaning as described above.

3.2.8 Experiments

I trained a WordPiece tokenizer on all articles in this dataset, extracted and tokenized headlines and abstracts and saved them in TFRecords format with each shard containing 10000 abstract-headline pairs. Then, I uploaded the processed dataset to Google Storage buckets so that there is minimal file transfering delay between the model run on TPU and the instance storing the data (all dataset operations are run on a CPU). The advantage of uploading TFRecords dataset to Google storage is that both TPU and Storage physical device are mapped to the same data center, so that the file transfer delay is minimised.

The model I selected is the Reformer model with hyperparameters set as reported in [KKL20] in "Experiments" paragraph. This setup together with the WordPiece vocabulary consisting of 30000 word pieces has 68 293 936 trainable parameters.

I also I came up with several hyperparameters optimization as a result of extensive experimenting. These are:

1. The number of warmup steps is set to 600.
2. Learning rate Scheduler is a `MultifactorSchedule` object with two factors: `"constant * linear_warmup"`. The constant value is 0.1
3. Optimizer's factor is set to 0.1

In case any of the parameters is increased, the model diverges. In the opposite case the model converges a little bit slower.

There is no need to additionally train a Transformer for this task since the authors of SumeCzech have already done it. The experiment is run in Trax 1.2.3 on a cloud TPU.

Since the experiment is run on a Cloud TPU v2 instance, Reformer is the ideal candidate, because it can fit the 8GB TPU's RAM and handle large batches. The batch size is set to 4096, which in my case means 4096 abstract-headline pairs. Because of TPU's padding requirements, the abstracts are padded to the length of 128 with those having more than 128 subwords truncated (the statistics showed there were not so many of them), the headlines are padded to the length of 64. This preparation barely affected headline-abstract pairs and meets all the TPU's requirements, so that it can be maximally used.

Considering the batch size of 4096, in order to train my model with the same amount of data as the authors of SumeCzech did, my model had to make $165000000/4096 \approx 40284$ training steps. An epoch in Trax is defined by a number of steps, so in order to get intermediate insight about how my model improves, I chose 250 steps for training, which means that the epochs number is 160. SumeCzech's dev (or validation) set consists of 44454 entries, so I set up the model in the way that after each epoch the model is evaluated on the whole dev set. The training process is constantly monitored via Tensorboard and the intermediate checkpoints and weights are automatically sent to my Google Storage account so that in case somebody preempts my Colab virtual instance, I am capable of continuing training from the last checkpoint.

3.2.9 Results

The training results are accessible via Tensorboard's interface:

It is obvious that after performing the first 15K steps, the model reached the plateau and its validation dataset accuracy did not improve while the training accuracy went steadily higher. One could think of a classical case of overfitting, however I consider this behaviour to be partially one of the indicators of a poor dataset. Starting from scratch, the model is unaware of the problem it has to solve. During the first 15K steps the model presumably learns not only the language in the training set, but also how to generate

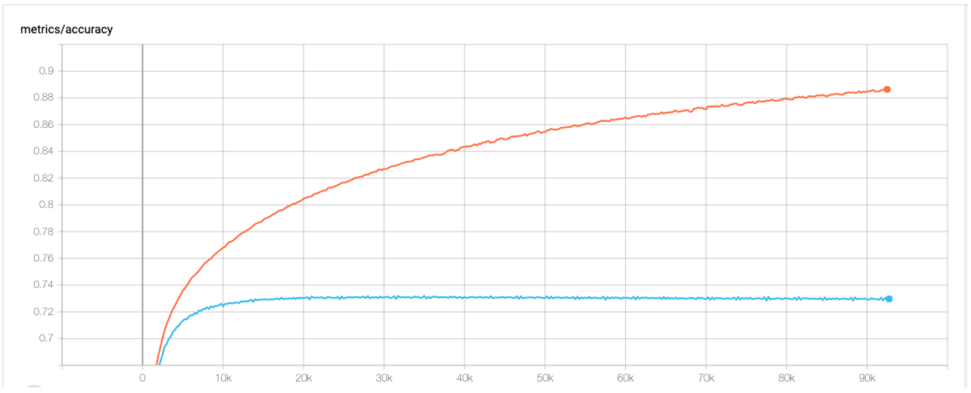


Figure 3.11: Summarization model accuracy
The orange curve depicts the model's performance on the training set, whereas the blue one depicts the model's performance on the evaluation dataset.
The x-axis resembles the number of training steps, the y-axis shows the model's accuracy on this step.

language. Recall that the target summaries in SumeCzech are not human-written, meaning that they are noisy. I presume that after making the first steps towards understanding what is in these abstracts, the model gradually learned how to compress them in general. After these 15K steps the model is definitely learning to adjust the summaries to what is considered to be a summary in the training set and still learns language, which is seen below.

By saying this, I do not state that this is not the case of overfitting. It still could be, however, more in-depth analysis has to be done to understand what causes the evaluation set to stop improving after first 15K steps of training.

One more argument is that when the model overfits, its performance on the evaluation set degrades. In case of this experiment, the overall accuracy is almost constant.

Nevertheless, if we take a look at the graph 3.12, we will see how the model's behaviour differs in terms of the loss function value. Recall that the loss function here is cross-entropy loss. Cross-entropy loss is defined as average number of total bits to represent an event from one distribution instead of another distribution. With that said, it means that the model has an improving tendency to be "not so far away from" the targets in the training set, whereas the targets in the evaluation set are becoming more "distant". My hypothesis is that this behaviour is given by the model's tendency to make summaries shorter over time, increasing model's recall score.

One more metric worth taking into consideration is shown in figure 3.13. Here we see sequence accuracy metric. The model is assigned a score as

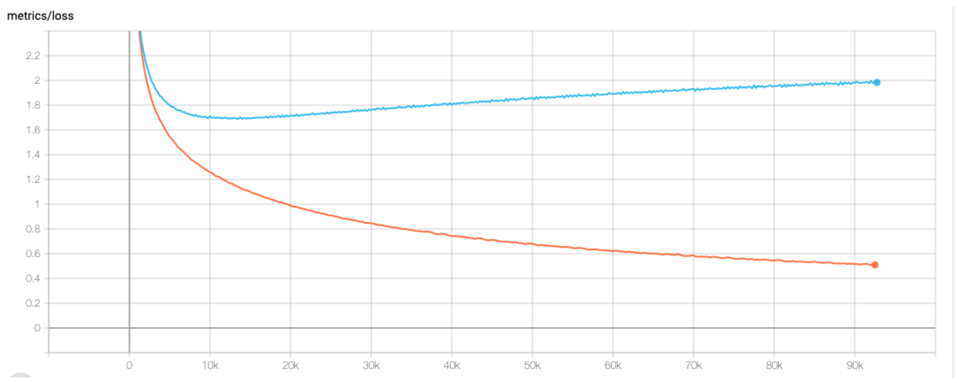


Figure 3.12: Summarization model loss
The orange curve depicts the model's performance on the training set, whereas the blue one depicts the model's performance on the evaluation dataset.
The x-axis resembles the number of training steps, the y-axis shows the model's loss function value.

soon as the whole output summary is identical to the one in the target set. Obviously, as the value reaches 1, we can claim the model is overfit. However, the values (scaled by a natural logarithm) are close to zero. The reason the graph is shown with y-axis in a logarithmic scale is that it clearly shows what happens when the model hits the plateau near 15K steps. The sequence accuracy on training data starts to converge to a constant value, which is somewhere around 0.1. Since it tends to flatten and become constant, I highly doubt the model started to overfit after 15K steps and stopped training.

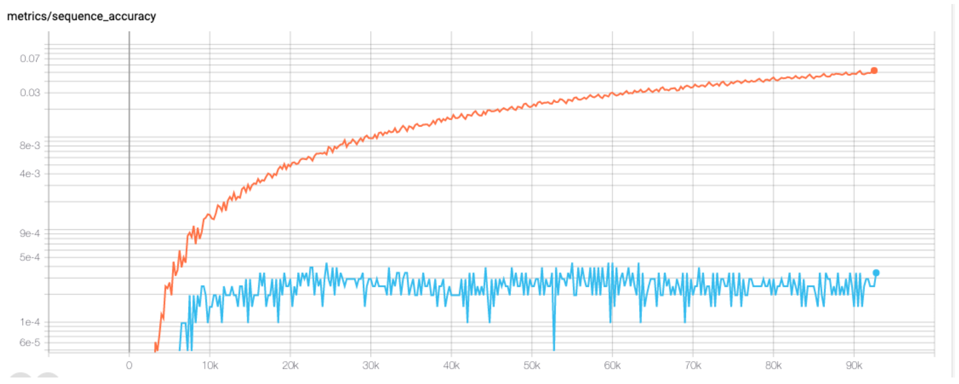


Figure 3.13: Summarization model sequence accuracy
The orange curve depicts the model's performance on the training set, whereas the blue one depicts the model's performance on the evaluation dataset. The x-axis resembles the number of training steps, the y-axis shows the model's loss function value and is in a logarithmic scale.

The last graph 3.14 depicts how the learning rate is scheduled while training. We see how fast it grows and then decays linearly with a factor of 0.2. Based on my experiments, this could be the most optimal and the fastest schedule, since every further manipulation with the scheduling parameters towards faster learning rate growth and slower decay caused model to diverge after several hundreds of steps, though showing gradual improvements over the first steps.

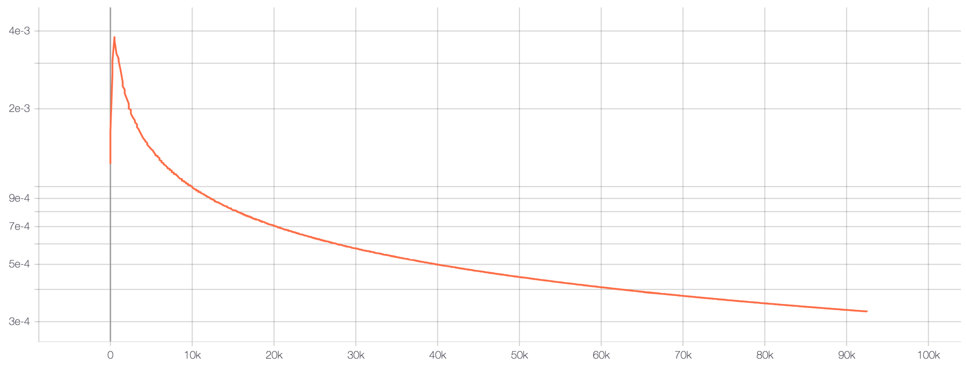


Figure 3.14: Summarization model learning rate scheduling
The x-axis represents the number of training steps, the y-axis shows the model's learning rate value.

3.2.10 Evaluation

To get a better insight on how the model performs after being trained for a while, I evaluate them by using ROUGE metric scores implemented in SumeCzech dataset.

Unfortunately, models trained in Trax are still unstable while being run in inference mode. First of all, there is an illogical misconception about the states a model can be run in. To train a model, one needs to set its mode to 'train', which sounds logical. When the model is been evaluated, its mode is set to 'eval', which from the technical perspective means the dropout scores are set to 0 and no activations are being stored. There is an additional 'predict' mode, which I was not able to run due to some errors in inner logic of the communication between layers. Somebody in Gitter, which is the main communicational platform to solve issues in Trax, supposed that this 'predict' mode is designed to run a model in an autoregressive fashion without incorporating the encoder, which is exactly what is implemented in several public Colab notebooks available from the official Trax repository on Github.

Taking into consideration the aforementioned, all the inference tests are executed in 'eval' mode. The model is loaded as follows from the snippet 3.3:

Listing 3.3: The way the model has to be loaded

```
import trax
model_path = 'path_to_model_pkl_file'

inference_model = trax.models.Reformer(mode='eval')
inference_model.init_from_file(model_path, weights_only=False)
```

Another thing worth mentioning is that the model is initialised with the parameter `weights_only` set to `False`. This setup also makes Trax load the model's states. As soon as the model is initialised together with all its states, the summaries look more consistent and correct.

One more unpleasant experience I had with Trax is that it is not completely optimised to run fast inference. Since the model is loaded in the evaluation mode with all its states, it takes considerably much more time to run tests even on a GPU. That's why I firstly compare several checkpoints against each other on the same extract from the test set included in SumeCzech, which had 100 batches, each of them containing 128 abstract-headline pairs. As I have noticed, all the models tend to converge to some concrete value and no significant changes happen afterwards. Trax models are capable of handling batches while being run in inference mode, that's why I run 100 inference steps with 128 input pairs per batch. Then, I choose the best one and run tests on the whole testing set in order to be able to compare my model with the one reported in [SMK+18].

Each test is run on Tesla P4 GPU provided by some heuristics in Colab and each batch took approximately 60-80 seconds to process, which means that 1 abstract-headline pair is handled in 0.5-0.625 seconds. Recall that the tested Reformer model had 68 293 936 trainable parameters, which is almost the same as 'base' Transformer's setup parameters number.

The table 3.5 gives an insight how the several checkpoints performed on the testing set extract.

Step	$ROUGE_{RAW}-1$			$ROUGE_{RAW}-2$			$ROUGE_{RAW}-L$		
	P	R	F	P	R	F	P	R	F
10000	18.0	16.5	16.7	4.4	4.0	4.1	16.2	14.9	15.0
12500	**18.4**	**17.1**	**17.2**	**4.7**	4.3	4.3	**16.5**	**15.4**	**15.4**
15000	18.1	16.9	17.0	4.6	4.3	4.3	16.3	15.2	15.3
47500	18.0	16.8	16.9	4.7	**4.4**	**4.4**	16.1	15.1	15.1
70000	17.7	16.6	16.6	4.6	4.4	4.3	15.9	15.0	15.0

Table 3.5: The results of the test set extract evaluation on different Reformer model's checkpoints in terms of ROUGE metric described in 3.2.7.

Indeed, it is evident that the best behaviour according to the ROUGE metric is reached at the 12500th step. However, it is still worth looking at some examples of the model's outputs to get better insight.

Model	Headline
Headline	Lidé upozornili spolucestujícího v tramvaji, že nemá roušku, napadl je pěstí. *People warned a fellow-passenger in the tram that he didn't have a mask, he punched them.*
5000	Muž napadl v tramvaji pěstí, pak ho napadl a napadl ho. *A man punched in a tram, them attacked him and attacked him.*
12500	Agresivní muž napadl v tramvaji pěstí, pak kopal do obličeje pěstí. *An aggressive man punched in a tram, then hit in the face with a fist.*
25000	Agresivní muž napadl pěstí a kopal do obličeje pěstí, pak kopal do obličeje. *An aggressive man punched and hit in the face with a fist.*
30000	Agresivní muž napadl v tramvaji pěstí, praštil ji pěstí do obličeje. *An aggressive man punched in a tram, smacked her in the face with a fist.*
35000	Agresivní muž napadl v tramvaji pěstí, praštil ji do obličeje. *An aggressive man punched in a tram, smacked her in the face.*
40000	Agresivní muž napadl agresivního řidiče tramvaje, útočník ho zkopal. *An aggressive man attacked an aggressive tram driver, the attacker kicked him.*

Abstract
Dva cestující v jedné z olomouckých tramvají napadl agresivní osmadvacetiletý muž poté, co ho upozornili, že nemá roušku. Muž nejprve udeřil několika ranami pěstí do obličeje šestačtyřicetiletého muže a poté i dvaapadesátiletou ženu, která se napadeného zastala. Konflikt vyřešila až přivolaná policie, útočníkovi za výtržnictví hrozí až dva roky vězení. *An aggressive 28 years old man attacked two passengers in one of Olomouc trams after being notified of not wearing masks. The man firstly left several wounds on the face of a 46 years old man and then punched a 52 years old woman who defended him. The conflict has been resolved by the police. The attacker is facing up to 2 years in prison for disturbing the peace.*

Table 3.6: The first example of summaries generated by the models after completing different number of training steps.

It can be seen from the table 3.6 that even in case of pretty reasonable overlap with real abstract, models that stopped training at lower steps struggle with language generation, whereas the last model, despite the fact that it misunderstood the object of the attack, was better in language understanding and generation. One more example is presented in table 3.7.

Model	Headline
Headline	Tesla v Kalifornii obnovila výrobu navzdory zákazu. Klidně mě zavřete, vzkazuje Musk *Tesla resumed production in California despite the prohibition. Feel free to close me, tells Musk.*
5000	Tesla zahájil výrobu v Kalifornii. Kvůli prodeji Tesla. *Tesla launched production in California. Due to Tesla sale.*
12500	Tesla zahájil výrobu v Kalifornii, kvůli poruše nesmí zatrhnout. *Tesla launched production in California, due to malfuction may not forbid.*
15000	Tesla zahájil výrobu v Kalifornii, výrobu pozastavil výrobu. *Tesla launched production in California, the production stopped production.*
45000	Tesla znovu rozjel výrobu elektromobilů. Navzdory embargu. *Tesla again started up production of electric automobiles. Despite the embargo.*
90000	Tesla znovu rozjel výrobu elektromobilů. Navzdory kritice. *Tesla again started up production of electric automobiles. Despite the criticism.*

Abstract
Americký výrobce elektromobilů Tesla znovu zahájil výrobu v kalifornském Fremontu, uvedl server listu Financial Times. A to navzdory tomu, že podle úředního nařízení výrobu obnovit nesmí. Elon Musk úřady vyzval, aby ho zatkly. *Despite the fact that according to the official order it may not restart production, American electric automobiles manufacturer Tesla resumed production in Fremont, California, says Finantial Times. Elon Musk called to be arrested.*

Table 3.7: The first example of summaries generated by the models after completing different number of training steps.

This is not the only case of how the models that have been longer trained understand language. My assumption is that it probably overfitted and hence performs worse in terms of ROUGE metric on the evaluation and test datasets. However, I see the ability of the model to learn sophisticated semantics and be more aware of the language itself. It can be seen from both two tables 3.6

and 3.7. In the first case, the tram driver was not explicitly mentioned in the abstract, however, the model somehow knew that the driver is connected to the public transport (here - the tram). The same is true when we notice that there is a word "útočník" (Engl. attacker), which has been used in the end of the sentence and carried an additional information not reflected in the summary. However, it helped the model understand that the main figure of the abstract is the attacker and managed to use the new word to represent it in its output.

The second example is even more fascinating: after much longer training we see not only better language understanding (the word embargo is not used in the abstract, however, the model even on a small training dataset managed to handle synonyms), but also the bias of the news articles. It learned to write short second sentences typically used to stress some particular point from the first one. This is exactly what we see.

I have to mension that all the tests and inference were made in a greedy manner. Unfortunately, Trax has a confirmed bug in the beam seach algoritm, which yields inconsistent results that in case of beam of size 1 do not correspond to what is obtained by autoregressive inference as described in 2.2.5.

3.2.11 Comparison with SumeCzech's model

I ran the best summarization model from the table above on the whole test set on Tesla P100-PCIE-16GB GPU, which is the best GPU available in Colab. Its strongest side is that it has 16GB of RAM, so it is capable of handling very large batches. Hence, one batch consisted of 256 abstract-headline pairs and it took from 90 to 120 seconds to process it. Since the experiments were run in Colab, I also had to create regular dumps so that in case I got disconnected from the GPU, I did not lose all intermediate calculations.

The results are presented in the table 3.8:

As it is seen in the table, my model outperforms the baseline in terms of $GOUGE_{RAW} - 1$ and $GOUGE_{RAW} - L$ metrics, both in recall and F1 score, which is in full correspondence with what I said about summarization metrics. SumeCzech's model is more 'precise'. It is however not always desirable to have higher precision over recall, that's why it is possible to say that I have reached state-of-the-art behaviour on this dataset with such a setup.

Method	$GOUGE_{RAW} - 1$			$GOUGE_{RAW} - 2$			$GOUGE_{RAW} - L$		
	P	R	F	P	R	F	P	R	F
Test set									
SumeCzech	**19.3**	15.4	16.6	**6.2**	**4.8**	**5.2**	**17.9**	14.3	15.4
My model	18.1	**17.2**	**17.1**	4.6	4.4	4.4	16.3	**15.5**	**15.5**
Out of domain test set									
SumeCzech	**18.9**	14.8	**16.0**	**6.8**	**5.0**	**5.5**	**17.7**	13.9	**15.0**
My model	16.8	**15.9**	15.7	4.8	4.6	4.5	15.5	**14.7**	14.5

Table 3.8: Models' comparison in terms of ROUGE metric described in 3.2.7

It is also important to say that SumeCzech's model obtained these numbers with the help of a beam search algorithm. The authors followed the recommendations from [VSP$^+$17] and chose a beam of size 4, which is considered to significantly improve the final translation's quality. Unfortunately, the Beam Search module in Trax is still buggy and gives incorrect results, so my numbers could be a little bit better if the beam search was applied. However, the model's behaviour corresponds to what was reported in [KKL20], where the Reformer setup slightly outperformed existing translation modules. It proves, that even when the model is assumed to be an approximation of another model, it does not necessarily mean that this model is worse in terms of final accuracy.

The model I trained and that is present in table 3.8 is a checkpoint after 12500 steps of training, which took less than 4 hours of TPU training. Recall that SumeCzech's model was trained for 14 days on a GPU. So, one more contribution of my work here is a novel approach how to train deep models in a shorter time span.

3.2.12 Final thoughts on summarization

Abstractive summarization is an extremely fascinating and challenging NLP task. It is even more complex than classical language to language translation, since the whole text and the underlying idea must be taken into consideration. I have conducted several small experiments towards solving this task, which I hope will be soon solved by more robust models trained in a much smarter way. Here are some ideas I suggest trying out:

1. Create much larger dataset which will be generated applying more strict heuristics and trying to avoid those texts without real summary but

sentences making a reader want to read the article (e.g. typical headlines in mass media:

a. Proč podvádějí a kradou? (Engl. *Why do they cheat and steal?*)
b. Pacientům lhal, že mají rakovinu. K soudu ale kvůli médiím nepřišel. (Engl. *Lied to patients that they had cancer. Didn't come to court due to mass media.*)
c. Perličky z krajských voleb na východě Čech. (Engl. *Funny facts about regional elections in the East of Czechia.*)
d. Ale fuj, tak brzo v práci, ohodnotila studentka zaměstnání bankéře. (Engl. *Yuck, so early at work, rated a student banker's occupation.*)

It is a very tough task which is not trivial to accomplish. One possible way of dealing with it is to train a bigger Transformer (or Reformer) model on the existing data, find those summaries in the training set having the lowest ROUGE values and try to filter them out. By cleaning up the dataset, it could have higher ratio of high-quality data, which can be then iteratively used for training new model. This is a way how to partially accomplish the lack of data.

2. Consider incorporating a language model before the data flows to the summarization model. The major flaw of many approaches is in the lack of prior knowledge. This is the reason why the approach from computer vision field called *transfer learning* is getting more popular in other spheres, including NLP. Having understanding of the language the summaries are written in is critical for this task, so I suggest taking a glance at currently state-of-the-art model BERTSum [Liu19], which is a modification of the BERT language model tuned for the summarization task. However, it does not need to be pretrained. This model is based on an assumption that by incorporating BERT's idea, the language model is learned hand-in-hand with the capability of making summaries. My suggestion is to introduce an encoder that will firstly generate sentences embedding on several levels: producing one fixed-size vector representing the sentence as Universal Sentence Encoder [CYK+18] does, combining it with word-level sentence embedding (e.g. sentence represented as a matrix).

3. Do not rely on Transformer's embedding while training on large volumes of data. Keep in mind that as soon as the data is projected to a smaller subspace, some information is inevitably lost, that is why it has to be recovered by other sources (features).

4. A new metric has to be introduced. Czech language cannot be evaluated in the same fashion as English or German. The new metric has to take into consideration many language aspects like lexical features (cases, persons, numbers) or semantic features (named entities, specific lexicon from particular field), be able to reduce the richness of Slavic languages so that it can be processed more easily.

5. Use beam search with semantic features unless the language model is incorporated.

Chapter 4

Conclusion

Losts of words have been said in this work and even more words remained concealed. I hope that my contribution to the field of Czech NLP will be not left unattended. In this work, I attempted to show how attention mechanism changed the way NLP community accomplishes many yet unsolved tasks and why is it crucial to be aware of how the ideas changed in course of time. Then, I conducted several experiments on two totally different tasks of diacritics correction and abstractive summarization. My endeavour was to convince a reader that NLP is a fascinating field with so dynamic pace that has to be kept up with. I showed how newly introduced models like Transformer and Reformer attempt to solve these problems and what are their advantages and disadvantages. I also made little contribution to Trax by raising many issues and detecting its weaknesses. Moreover, I introduced a feature of attention heads visualisation by making several additions to the existing code. I hope to continue working in this direction and making Czech NLP systems more accurate and more robust towards genuine language understanding.

Appendix A

Bibliography

[BBDP+94] Adam L. Berger, Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, John R. Gillett, John D. Lafferty, Robert L. Mercer, Harry Printz, and Lubos Ures, *The Candide System for Machine Translation*, Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994, 1994.

[BCB16] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio, *Neural Machine Translation by Jointly Learning to Align and Translate*, arXiv:1409.0473 [cs, stat] (2016), arXiv: 1409.0473.

[BHH+13] Eduard Bejček, Eva Hajičová, Jan Hajič, Pavlína Jínová, Václava Kettnerová, Veronika Kolářová, Marie Mikulová, Jiří Mírovský, Anna Nedoluzhko, Jarmila Panevová, Lucie Poláková, Magda Ševčíková, Jan Štěpánek, and Šárka Zikánová, *Prague dependency treebank 3.0*, 2013.

[Bus] Steven Bussey, *Alignment of Translation.*

[Chr20] Olah Christopher, *Understanding LSTM Networks – colah's blog*, 05 2020.

[CKLM19] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning, *What Does BERT Look At? An Analysis of BERT's Attention*, arXiv:1906.04341 [cs] (2019), arXiv: 1906.04341.

[CV95] Corinna Cortes and Vladimir Vapnik, *Support-vector networks*, Machine Learning **20** (1995), no. 3, 273–297 (en).

[CYK$^+$18] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil, *Universal Sentence Encoder*, arXiv:1803.11175 [cs] (2018), arXiv: 1803.11175.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova, *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, arXiv:1810.04805 [cs] (2019), arXiv: 1810.04805.

[(ed16] Cvrček, Václav Richterová, Olga (eds), *cnk:syn2015 — příručka Čnk*, 2016, [Online; accessed 20-May-2020].

[GCK$^+$17] George Giannakopoulos, John Conroy, Jeff Kubina, Peter A. Rankel, Elena Lloret, Josef Steinberger, Marina Litvak, and Benoit Favre, *MultiLing 2017 Overview*, Proceedings of the MultiLing 2017 Workshop on Summarization and Summary Evaluation Across Source Types and Genres (Valencia, Spain), Association for Computational Linguistics, April 2017, pp. 1–6.

[goo] *Cloud TPU.*

[GRUG17] Aidan N. Gomez, Mengye Ren, Raquel Urtasun, and Roger B. Grosse, *The Reversible Residual Network: Backpropagation Without Storing Activations*, arXiv:1707.04585 [cs] (2017), arXiv: 1707.04585.

[HL98] Eduard Hovy and Chin-Yew Lin, *Automated Text Summarization and the Summarist System*, TIPSTER TEXT PROGRAM PHASE III: Proceedings of a Workshop held at Baltimore, Maryland, October 13-15, 1998 (Baltimore, Maryland, USA), Association for Computational Linguistics, October 1998, pp. 197–214.

[Hop82] John Hopfield, *Neural networks and physical systems with emergent collective computational abilities*, Proceedings of the National Academy of Sciences of the United States of America **79** (1982), 2554–8.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber, *Long Short-Term Memory*, November 1997.

[JKS18] Aditya Jain, Gandhar Kulkarni, and Vraj Shah, *Natural language processing*, International Journal of Computer Sciences and Engineering **6** (2018), 161–167.

[KKL20] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya, *Reformer: The Efficient Transformer*, arXiv:2001.04451 [cs, stat] (2020), arXiv: 2001.04451.

[KMTD14] Mikael K\aagebäck, Olof Mogren, Nina Tahmasebi, and Devdatt Dubhashi, *Extractive Summarization using Continuous Vector Space Models*, Proceedings of the 2nd Workshop on Continuous Vector Space Models and their Compositionality (CVSC) (Gothenburg, Sweden), Association for Computational Linguistics, April 2014, pp. 31–39.

[Lin04] Chin-Yew Lin, *ROUGE: A Package for Automatic Evaluation of Summaries*, Text Summarization Branches Out (Barcelona, Spain), Association for Computational Linguistics, July 2004, pp. 74–81.

[Liu19] Yang Liu, *Fine-tune BERT for Extractive Summarization*, arXiv:1903.10318 [cs] (2019), arXiv: 1903.10318.

[LMP01] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira, *Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data*, Proceedings of the Eighteenth International Conference on Machine Learning (San Francisco, CA, USA), ICML '01, Morgan Kaufmann Publishers Inc., June 2001, pp. 282–289.

[LPM15] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning, *Effective Approaches to Attention-based Neural Machine Translation*, arXiv:1508.04025 [cs] (2015), arXiv: 1508.04025.

[Mih04] Rada Mihalcea, *Graph-based ranking algorithms for sentence extraction, applied to text summarization.*

[MSC+13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean, *Distributed Representations of Words and Phrases and their Compositionality*, arXiv:1310.4546 [cs, stat] (2013), arXiv: 1310.4546.

[Nom05] Tadashi Nomoto, *Bayesian learning in text summarization*, 01 2005.

[Ná17] Bc. Jakub Náplava, *Natural Language Correction*, 2017.

[PB18] Martin Popel and Ondřej Bojar, *Training Tips for the Transformer Model*, The Prague Bulletin of Mathematical Linguistics **110** (2018), no. 1, 43–70, arXiv: 1804.00247.

[PNI+18] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer, *Deep contextualized word representations*, arXiv:1802.05365 [cs] (2018), arXiv: 1802.05365.

[PSM14] Jeffrey Pennington, Richard Socher, and Christopher Manning, *Glove: Global Vectors for Word Representation*, Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP) (Doha, Qatar), Association for Computational Linguistics, October 2014, pp. 1532–1543.

[RČ13] Michal Rott and Petr Červa, *Summec: A summarization engine for czech*, Text, Speech, and Dialogue (Berlin, Heidelberg) (Ivan Habernal and Václav Matoušek, eds.), Springer Berlin Heidelberg, 2013, pp. 527–535.

[RHW88] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams, *Learning representations by back-propagating errors*, Neurocomputing: foundations of research, MIT Press, Cambridge, MA, USA, January 1988, pp. 696–699.

[Ros58] Frank F. Rosenblatt, *The perceptron: a probabilistic model for information storage and organization in the brain.*, Psychological review (1958).

[SHB16] Rico Sennrich, Barry Haddow, and Alexandra Birch, *Neural Machine Translation of Rare Words with Subword Units*, arXiv:1508.07909 [cs] (2016), arXiv: 1508.07909.

[SJ04] Josef Steinberger and Karel Jezek, *Using Latent Semantic Analysis in Text Summarization and Summary Evaluation*, 2004.

[SMK+18] Milan Straka, Nikita Mediankin, Tom Kocmi, Zdenek Zabokrtský, Vojtech Hudecek, and Jan Hajic, *Sumeczech: Large czech news-based summarization dataset*, LREC, 2018.

[THP08] Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel, *Efficient aggregation for graph summarization*, Proceedings of the 2008 ACM SIGMOD international conference on Management of data (Vancouver, Canada), SIGMOD '08, Association for Computing Machinery, June 2008, pp. 567–580.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin, *Attention Is All You Need*, arXiv:1706.03762 [cs] (2017), arXiv: 1706.03762.

[WSC+16a] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean, *Google's Neural Machine Translation System: Bridging the Gap between Human* *and Machine Translation*, arXiv:1609.08144 [cs] (2016), arXiv: 1609.08144.

[WSC+16b] ______, *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*, arXiv:1609.08144 [cs] (2016), arXiv: 1609.08144.

[YJL+16] Jun Yin, Xin Jiang, Zhengdong Lu, Lifeng Shang, Hang Li, and Xiaoming Li, *Neural Generative Question Answering*, arXiv:1512.01337 [cs] (2016), arXiv: 1512.01337.

[ZLLE17] Tiancheng Zhao, Allen Lu, Kyusong Lee, and Maxine Eskenazi, *Generative Encoder-Decoder Models for Task-Oriented Spoken Dialog Systems with Chatting Capability*, arXiv:1706.08476 [cs] (2017), arXiv: 1706.08476.