TransPoemer

Sieben Bocklandt & Ruben Broekx
Catholic University of Leuven
Declarative Languages and Artificial Intelligence research group
3000 Leuven, Belgium

Abstract

This work presents a model that transforms regular text into a poem, hence, this model is likewise called the *TransPoemer*. The implemented model consists of two main parts, the first part will extract the keywords from a sentence, whereas the second part will transform these keywords into a poem. The latter part of the model makes use of the Transformer model that was released by Google in 2017. Since these type models have grown exponentially in size since their first release, we asked the question if it would be possible to train such a model from scratch on a laptop and still obtain satisfactory results. We conclude that this is indeed possible, however, we also state that it's not possible to obtain a model capable of rhyming, given the limited number of training samples we've used.

Background

In 2017, A. Vaswani, N. Shazeer, N. Parmar et al.[1] published the Transformer model, a type of Deep Learning neural network that has been proven to be very effective for common natural language processing tasks. The model is designed to handle sequences of ordered data well, which makes it very applicable for natural language-related tasks. These tasks include translation, summarization, question answering and much more.

The Transformer model is often considered as an improvement over Recurrent Neural Networks (RNN), an older Deep Learning architecture that was previously considered to be state-of-the-art in similar tasks. RNNs are suitable for language modelling tasks since they made correlations between subsequent parts of the inputs, this by means of an internal memory. However, the main shortcoming of this type of network is that they must surpass the full input-sequence of length `N` before being able to predict the `N+1`th element. This prerequisite makes the network inherently sequential, with as result that it needs a lot of memory and is slow to train. A second shortcoming relates to the finiteness of the RNN's memory capacity. Since this internal memory is often rather limited, the network has the tendency to forget earlier seen inputs when long segments are fed into it. Even though the introduction of Long Short-Term Memory[2] (LSTM) and Gated Recurrent Unit[3] (GRU) cells significantly improved the memory capacity of the RNN, albeit with the cost of even slower learning due to the increase in parameters, long segments remained a difficulty. The introduction of an attention mechanism on these RNNs helped to resolve this problem since it tells the network to pay attention to the parts of the inputs that matter, or even more important, it indicates which parts to ignore.

The Transformer model states that this attention mechanism is so powerful that it's the only thing needed to base a prediction on. They go as far as stating that an internal memory and recurrent connections, as seen with RNNs, are obsolete in these Natural Language Processing related tasks. By applying these principles, they obtain some significant advantages over RNNs:

- They obtain better results.
- They train faster since the inherently sequential behaviour as seen with RNNs is lost. This also implies that it is easier to parallelize the training of the model.
- They need less memory.
- They are more interpretable since their attention layers can be visualised by means of a heat-map. By doing so, it's possible to track down to which parts of the input the model was paying attention to for each predicted word.

At last, something interesting to note is that at the time of writing, the limits of these models are not yet known. Every few months, a new and even larger Transformer model is published that outperforms the current state-of-the-art. A general rule surrounding the Transformer model states that: *The bigger they are and the more data you feed them, the better they perform.*

[1] N. Parmar et al. A. Vaswani N. Shazeer. *Attention is all you need.* `https://arxiv.org/abs/1706.03762`. 2017

[2] Sepp Hochreiter. *Long Short-term Memory.* `https://www.researchgate.net/publication/13853244_Long_Short-term_Memory`. 1997

[3] B. van Merrienboer et al. K. Cho. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.* `https://arxiv.org/abs/1406.1078`. 2014

Creative Task

In this work, a model that transforms small texts into poems is implemented, which we'll call the *TransPoemer*. In the ideal scenario, our model should be able to take a regular small sentence as input, extract the core keywords from this text and create a *limerick-like* poem based on these keywords. A limerick is a short, five-line verse, with an `AABBA` rhyming scheme. The main reason why we've settled with this type of poems is that it's empirically easy to check if the model indeed learns how to rhyme.

Implementation

Architecture

The model consists of two different parts: first, a keyword-extractor is used to extract the most relevant keywords from the text, then, a Transformer is used to translate these keywords into a poem. These two parts will be discussed in more detail in subsections `Keyword Extraction` and `Transformer` respectively. The overall architecture is as follows:

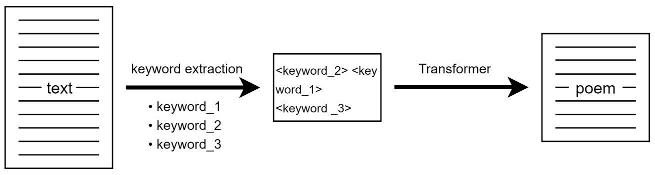


Figure 1: Model Architecture

Design Decisions

Two key design decisions were made that had a large influence on how this project came to an end. First, we wanted to create a model that could be trained on a laptop. In the recent trend of Transformer models continuously growing in size (take for example Microsoft's Turing-NLG model[4] of 17 billion parameters), we were curious to see how a significantly smaller model, i.e. one that could fit in a laptop's GPU, would perform. The hardware on which the model will be trained is an Nvidia GeForce GTX 1060 GPU[5] with 4GB of memory.

The second design decision is to train the model from scratch. This constraint was rather obliged than self-chosen since there are no pre-trained, easy to obtain Transformer models online available that fit a laptop's GPU.

Data

Two different datasets were used throughout the project. A limerick dataset[6] of 90.000 samples was used to train the model on since this relates to our creative task. However, since the model would be trained from scratch and 90.000 samples are rather limited, we also used a quotes dataset[7] of 500.000 samples to pre-train our model on.

Processing The data processing pipeline consists of multiple sub-phases. In order, these phases are:

- Special tokens such as `\t`, `\n` and non-Latin characters are removed.
- Duplicate segments (e.g. `???`) are shortened to only one occurrence (`?`).
- Since the Transformer model has a fixed number of parameters, they also need to receive inputs of a fixed size. When looking at the raw limerick data samples' length distribution, we note that an input-size of 68 suits well (note that the final input-size is then 70 since each fed samples starts and ends with a starting and ending token respectively). To enforce this constraint, samples that exceed this size-limit are split on an intermediate point, question or exclamation mark. By doing so, it is ensured that sentences fed into the model are always complete. If a sample is not splittable, it is discarded.

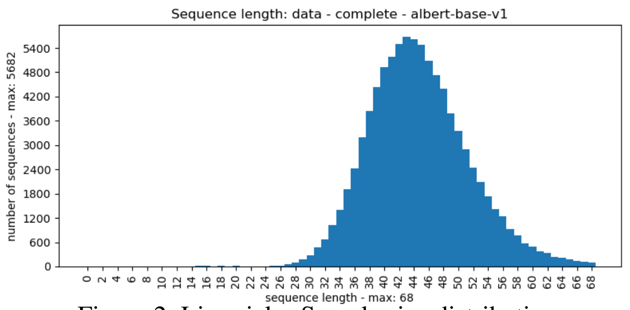


Figure 2: Limerick - Sample size distribution

- At last, it is checked if the remaining samples are indeed in English. This is checked via a language detector library called `langdetect`[8].

After processing, the quotes dataset expands to reach above 600.000 samples due to the split-operation, where the size of the limericks dataset remains unchanged. This processed data is then split into a training, validation and test set. These sets contain 90%, 5% and 5% of the full processed dataset respectively.

Tokenization Before feeding the samples to the network, they need to be tokenized. A first decision we've made is to lower-case all the words, this to reduce the word-variety in our dataset. Next, instead of tokenizing the words on a "complete-word" level (i.e. each unique word would have a unique token), we've decided to use a sub-word tokenizer since this would decrease the numbers of tokens used drastically. It's important to keep the number of used tokens low since these are used to represent the inputs and thus have a direct influence on the number of parameters used to represent the model. The tokenizer we've decided to use is one used by a pre-trained ALBERT Transformer model found

[4]Microsoft. *Turing-NLG: A 17-billion-parameter language model by Microsoft*. `https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/`. 2020

[5]Nvidia. *GEFORCE GTX 1060*. `https://www.nvidia.com/en-in/geforce/products/10series/geforce-gtx-1060`. 2016

[6]sballas8. *PoetRNN*. `https://github.com/sballas8/PoetRNN`. 2015

[7]Manan. *Quotes-500k*. `https://www.kaggle.com/manann/quotes-500k`. 2020

[8]Michal Mimino Danilak. *langdetect 1.0.8*. `https://pypi.org/project/langdetect/`. 2020

in the HuggingFace library[9]. This tokenizer originally uses 30.000 distinct tokens, but we've decided to use only the 20.000 most frequently used tokens due to two reasons:

- To keep the number of parameters used by the model low, since an increase in the input-space results in a significant increase in used parameters.
- The removed tokens were used only twice or less in the limerick training-dataset, hence, the removal of these tokens has little to almost no influence on the performance of our model.

Unknown (discarded) tokens are simply removed instead of substituted with the `<unk>` token. This is done with the idea that the model would learn no relation with the next (most likely vaguely correlated) token, where it could learn a relation to when an `<unk>` token would occur.

Keyword Extraction

The first part of our model is where the keywords are extracted from the regular text. To do so, we've used the Rapid Automatic Keyword Extraction[10] (RAKE) algorithm, which is based on the Natural Language ToolKit[11] (NLTK). The RAKE algorithm is based on the Term Frequency-Inverse Document Frequency (TF-IDF) rule that statistically reflects how important a word is based on how often it is used in other text corpora relative to the current corpus.

From each sample, the ten most important keywords are extracted and inserted back into the corpus in order. Consider the following example sentence: "*this is a piece of example text fed into the model that will be used to train the model on*". The keywords extracted by the algorithm are: "*used*", "*train*", "*piece*" and "*model*". The resulting keyword-sentence, which will be the input to the second part of our model, is then: "*piece model used train model*" since this is the order in which the keywords occurred in the original sample.

Transformer

The implemented Transformer model is mainly influenced by the `pytorch-seq2seq` GitHub repository[12]. Their implementation is based in part on the HuggingFace[13] implementation of the Transformer, which is implemented in PyTorch[14].

Our model has a hidden vector-space of dimension 128, eight heads for both the encoder and the decoder, with each of these heads consisting out of three layers. The feedforward dimension of both the encoder as the decoder is equal to 256. At last, a dropout-rate of 0.1 was applied at the embeddings of both sides. This all, combined with the input-size of 70 and the tokenizer-dimension of 20.000, results in a model that has 8.7 million parameters.

[9]HuggingFace. *ALBERT*. `https://huggingface.co/transformers/model_doc/albert.html`. 2020

[10]Vishwas B Sharma. *rake-nltk 1.0.4*. `https://pypi.org/project/rake-nltk/`. 2018

[11]Natural Language Toolkit. `https://www.nltk.org/`. 2020

[12]bentrevett. *pytorch-seq2seq*. `https://github.com/bentrevett/pytorch-seq2seq`. 2020

[13]HuggingFace. *Transformers*. `https://huggingface.co/transformers`. 2020

[14]PyTorch. *PyTorch*. `https://pytorch.org/`. 2020

Training

The only part in our model that needs training is the Transformer. This will be done in an auto-encoding fashion, in which the same sample fed into the (complete) model is used as the target output. In other words, each sample that is fed into the model will have its keywords extracted using the RAKE algorithm, these keywords are then given to the Transformer-part of the model which will try to translate these keywords back to the original sample.

As stated before, since the model is trained from scratch, it will undergo pre-training on the quotes dataset before being fine-tuned on the limerick samples. Two different training-sessions will be executed and compared afterwards:

- A model that is pre-trained on the quotes dataset, and then fine-tuned on the limericks dataset. We'll call this model "`Model PF`" (Pre-Fine) from now on.
- A model that is solely trained on the limericks dataset. We'll call this model "`Model F`" (Fine) from now on.

It is also interesting to see how the pre-training only model would perform as a baseline to the other models. The model obtained after solely pre-training will be called "`Model P`" from now on.

The training itself is executed identically for both the pre-training as the fine-tuning part. A batch-size of 64 is used since this was the biggest batch-size that would fit in the GPU's memory. Since the task is a classification task (i.e. predict which word is most likely to occur next), the cross-entropy is calculated on the final softmax-layer to represent the loss function. As optimizer, we've opted to use the Adam optimizer with a static learning rate of 0.0005.

During training, a trade-off must be made between originality and quality. If we want to keep the predictions of our model *original*, then we must ensure that the model will not overfit during training. Overfitting is scenario where the model is trained too much on its given training-set, that it will start reproducing samples seen during training, instead of predicting truly *original* content. On the other hand, if we want to produce quality outputs then it could be an advantage to have this overfit-bias present in our model since this would imply that parts of the training data would be present in the prediction. These predictions would then be of good quality based on the assumption that the samples fed to the model during training also were of decent quality. Since we want to achieve a model that is *creative*, we've put the focus on the former option. To enforce this, an early stopping mechanism was applied that will terminate the training session if no improvement is seen in the validation loss for the last three epochs. After termination of the training, the fittest model will be loaded back into memory.

`Model PF` trained for five epochs on the pre-training dataset and another five epochs on the fine-tuning dataset before being terminated by the early stopping mechanism. `Model F` trained for twelve epochs on the limericks dataset before its loss started back increasing. Finally, `Model P` was extracted from `Model PF` right after pre-training, which means that it had only five epochs of training on the quotes dataset.

Evaluation

Due to the lack of a clear outlined task given to the model, it is hard to evaluate the model's performance. Consider the following; given only at most ten keywords, it is hard if not impossible for even a human to predict the original sentence. Furthermore, it isn't necessarily the case that we want our model to predict the right target sentence, we just want *a sentence* that contains the given keywords and also happens to rhyme using the `AABBA` rhyming scheme.

To evaluate our model, we considered the following metrics:

- The cross-entropy loss, as defined during training.
- The GLEU[15] score (Google BLEU). The reason why we preferred the GLEU score over the traditional BLEU score often seen in translation tasks is that the GLEU score is better suited when there is only one target sample to compare with. In other words, BLEU is meant to be used when there are multiple *correct* samples to which one prediction can be compared to. The GLEU score returns a score between 0 and 100, with 100 being the "*perfect prediction*".

We've opted to use libraries that check how well a sample is written (e.g. grammatical checkers), but these lead often to misleading results. Since these checkers were no good representation on the overall quality of the samples we've decided not to mention these further in the report.

The two models fine-tuned for the task at hand obtained similar results on the loss-metric, but `Model P` outperformed `Model PF` on the GLEU score, which implies that `Model P` was slightly better at predicting the correct subsequences of tokens (i.e. it was better at predicting which tokens correlated to each other). The exact scores were:

- `Model PF`, loss: 3.537, GLEU: 13.17
- `Model P`, loss: 3.791, GLEU: 14.17

As stated before, these metrics only give a vague overview of the models' performance. To get a better view of how well both models perform, an empirical check on their predictions is mandatory, which is done in the next section.

[15] Z. Chen Q.V. Le M. Norouzi et al. Y. Wu M. Schuster. *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation.* `https://arxiv.org/abs/1609.08144`. 2016

Results

The following example gives a clear representation of the overall quality of the two models:

Keywords break sprout thighs snout
Model PF if you break a small metal, you'll sprout from your thighs, and your snout is a big deal. you'll find something drastic, and you'll find something drastic, and you'll see.
Model F when you break out of your thighs, you can sprout, and your thighs are quite clear. if you're out of your snout, you're a pig, and you'll find yourself a big mistake.

Surprisingly, the two models perform very similar, even though `Model F` was trained on dataset less than 1/7th the size of the dataset used to train `Model PF` on. It's interesting to see that both models succeed to include all of the provided keywords into their prediction. Overall, both predictions can be classified as *English gibberish*, meaning that they both look and read like regular English sentences, but don't care that much of meaning. Unfortunately, the predictions don't follow the `AABBA` rhyming scheme, or even rhyme at all.

Another often occurring *symptom* that only `Model PF` has, is that it tends to loop. A clear example of such a loop is as follows:

Keywords someone peeling okay montage cool collage creation
Model PF someone's peeling, okay, and the montage is cool, collagen, collagen, collagen, collagen, collagen, collagen, collagen, collagen, collage, collage, collage, collage, collage, collage, collage, collage. creation.
Model F someone's peeling a montrature, okay? it's a montage. it's cool, collagen's creation. it's a popular thrill.

A remarkable observation is that out of the first twenty samples in the test-set, `Model PF` contained a loop for three of those samples, where `Model F` never looped.

For a third experiment, we were curious to see how the models would perform if only one keyword or even none were given to them, "*pretty bad*" sums it up nicely:

Keywords rain
Model PF the "the "old" is a "the """""""" "it's a """""""" "it's a "no," "it's a "no," "it's a "it"."
Model F the "the rain" is the sun, and the sun's wet. it's a clear demonstration of rain, and the sun's a good place, and the sun's got a good place to behold.

Keywords /
Model PF "i'm a man," said the guy, "i'm a man." "i'm a man," said the guy, "i'm a guy," said the guy. "i'm a guy." "i'm a guy."
Model F the "the "s" is the "s". "i'm a "s-a-"." "i'm a "s-uh-uh-uh-uh-uh-uh-uh-uh-uh-uh-uh-uh-uh-uh

We conclude that this is one of the side effects of the limited training both models had. What is interesting to see, however, is that the model that was only pre-trained worked decently well in these scenario's, predicting "*the rain is a beautiful thing.*" and "*the best way to get is to be a good person.*" respectively.

Several other examples are given in `Appendix A` and a more detailed examination concerning the attention heat-map of one of those examples is given in `Appendix B`. In these appendices, two interesting observations are made:

- `Model P` tends to create small sentences concerning one or a few subsequent keywords and stitches these sentences after each other as the final prediction.
- Both `Model PF` as `Model F` have a hard time paying attention to the right keywords. The heat-maps of the attention of the decoder's first head, found in `Appendix B`, show that most of the intermediate words are predicted without paying attention to the input at all. In other words, these intermediate words are predicted solely based on the previously predicted (output) words.

At last, it's important to note that the found predictions are indeed *original*. Only a very limited set of subsequent words (e.g. "*said the*") is present in the dataset. This implies that our goal of obtaining original predictions indeed holds.

Discussion

When reflecting on the performance of our models, we can conclude that our expectations were partly met. Considering that the models created were of limited size, trained from scratch on a laptop and only used a rather small dataset, they were still able to create English-like sentences. However, the main goal of this project (i.e. creating a model that transforms a text into a poem that follows the rhyming scheme `AABBA`) was not met since the predicted outputs hardly ever rhymed. The remainder of this section will discuss a critical reflection of our work, as well as some potential improvements.

The first point of improvement, which was one of the big factors influencing the model's performance, is the limerick dataset. Although the processing (discussed in `Implementation`) phase tried to improve the flaws of this dataset where possible, it seemed that still a lot of *rotten samples* passed through. Examples of such samples are ones where multiple words were concatenated without any white space in between them, or simply samples that made no sense at all. Furthermore, often the words used were syntactically incorrect (e.g. words written in medieval English), which confused the model when it was pre-trained on the quotes dataset that was of good quality. Replacing this dataset with another one of better quality would most likely improve the model's performance, definitely that of `Model PF`. An example of such a dataset would be one that consists of *Nursery Rhymes* since these are short verses meant for children and thus would most likely not include syntactically incorrect words.

Next, we hypothesised that the quotes and the limericks would have a similar syntactical structure, hence we stated that the quotes would be a good dataset to pre-train our model on. However, as shown in the `Evaluation and Results` sections, the pre-trained model often performed slightly worse compared to the model that was only fine-tuned, even though it was trained on more than seven times the data. We conclude that this was due to a *valley* in between the pre-training and fine-tuning optima in which the model would get stuck during training. At first, we thought that this worse performance was due to the early stopping mechanism that potentially stopped the learning process too early. However, after training the model without the early stopping mechanism we can conclude that this was not the case. We state that this drop in performance relates to the previous point of improvement where was said that the words used vary regularly between the two datasets (i.e. quotes use clean, syntactically correct English, where the limericks often do not).

The lack of data to train the model on was most likely another major reason for the limited performance of our model. A general rule of thumb for any neural network is to have ten times as much data as the number of parameters the model has. However, in our case, we had a little shy of 700.000 data samples but around 8.700.000 parameters to train. The lack of data was mainly visible in the *logic* of the predicted samples. Often, the following structure occurred in the predictions: "*'blabla' said 'blabla'*". In such a structure, at least one of the two sides needs to be a person, i.e. "*'blabla' said the man*" or "*the man said 'blabla'*". There are two options to resolve this problem: or the model needs to be trained on a larger dataset, or a pre-trained model should be used. Note that the latter solution comes with the cost of not being able to choose your model-setup, which directly influences the number of parameters used.

Another potential point of improvement is the number of keywords used. We've decided to feed at most 10 keywords to the Transformer model which it would then use to base its translation on. Perhaps, this number was a little high resulting in this limiting the *creativity* of our model. But on the other side, having more keywords has the advantage that it helps the model more in the *right* direction (i.e. towards the target prediction), hence, a trade-off must be made. What is often seen now (`Appendix A`) is that each of the trained models tried to stitch words together with as less intermediate words as possible. A way to give the models more *creative freedom* would be to lower this keyword-count, but this decrease would also make the task of predicting the right target output harder.

The last point to discuss is our design decision to approach this creative task (see `Creative Task`) as a translation problem rather than a text generation problem. A counter-argument would be that this approach (translation) forces the model to use the keywords in a certain order. However, we're still convinced that this is an interesting approach and think that the model would be able to pull this off if it was given enough data to train on. Also, as stated before, the text-generation has already been proven to work[16], hence it would not be interesting to research.

[16]Gwern Branwen. *GPT-2 Neural Network Poetry*. `https://www.gwern.net/GPT-2`. 2019

Conclusion

We conclude this project with the following statement:

Transformers are indeed laptop trainable, as long as your expectation aren't too high

With this statement, we imply that if you want to use the Transformer model for a small and very specific task, then it wouldn't be needed to use any of the large state-of-the-art models found online, but that a rather small model would work out just fine. However, in contrast to what was done in this project, we would recommend using an already pre-trained model. An example of such a small pre-trained model could be the ALBERT Transformer model[17], which has a little above 11 million parameters.

Time Estimation

This section discusses the time-estimation put into this project. Since the subject at hand was fairly new, a lot of time was put into the understanding of the model's implementation and capabilities at the beginning of the project. Once we understood well how the model operates, it was time to find a creative task. Several draft ideas were made, but fairly quick we settled for the idea proposed in this work.

Next, we wanted to get our hands dirty. Before implementing the project itself, we wanted to fiddle around with the Transformer model such that we knew that we were using it right. We started reading through the HuggingFace library's documentation since originally we wanted to use one of their models. However, several days past by where we were trying to get a model trainable on our laptops without any real result. The main reason why it kept failing was due to the model-size being too big. The smallest model in their catalogue of Transformers was the ALBERT model, which had a little above 11 million parameters. We were able to use this model to generate text, however, it always crashed during training (due to memory constraints). Out of compulsion, we tried to find a *hacky* way to change the model's number of parameters. An idea we had was to reduce the number of heads used by the model. This reduction would indeed take away a large portion of the model's quality, but we thought that it'll be still better than a model with randomised weights. Unfortunately, we were not able to pull this off, hence we searched for a way to train our model from scratch.

Although the HuggingFace library looked promising at first, it started to be a real pain in the ass when we wanted to set up the model ourselves. Hence, we've decided to use another library. After searching through several GitHub repositories and tutorials we found the `pytorch-seq2seq` library that contained a decent tutorial, it was the holy grail that came in desperate needs.

Once we figured out the implementation of the Transformer model, the rest followed rather easily. The data gathering and processing took some time to implement but wasn't much of a hassle.

[17]HuggingFace, *ALBERT*

A lot of time was spent fine-tuning the project. Examples of this were additional phases in the data-processing, updating the model's configuration and figuring out why the model performed as it did. Training the model took some time as well: $\pm$30min for the data-processing part, $\pm$30min each epoch of pre-training, $\pm$5min each epoch of fine-tuning and $\pm$30min for the (automatic) evaluation of the model.

At last, more time than expected was put into writing this report and preparing the presentation. This task alone took over 20 hours in total.

Overall, we expect that the combined time put into this project was between 140 and 160 hours. Sieben helped reading through the theoretical matter, setting up the creative task and reading through this paper and the presentation. Ruben's influence was present in each aspect of the project, mainly in the implementation itself and the reporting of it. Although this division of labour was significantly imbalanced, we deliberately choose it this way. The difference in effort put into this project was compensated in the other Capita Selecta project where Sieben did the vast majority of the work. The reason why we choose to divide the work like this is since Ruben had the most experience working with Neural Network libraries (e.g. PyTorch), where Sieben was more familiar with the other Capita Selecta project.

Used Resources

The following online datasets, repositories and libraries were used throughout this project (in order of occurrence throughout the text):

- `PoetRNN` GitHub repository for our limerick dataset.[18]
- `Quotes-500k` Kaggle dataset for our quotes dataset.[19]
- `langdetect` library used during data processing.[20]
- `AlbertTokenizer` from the `transformers` library published by HuggingFace.[21]
- `pytorch-seq2seq` GitHub repository which significantly helped with the understanding and implementation of the Transformer model.[22]
- `PyTorch` library, which was used to implement the Transformer model.[23]

[18]sballas8, *PoetRNN*
[19]Manan, *Quotes-500k*
[20]Danilak, *langdetect 1.0.8*
[21]HuggingFace, *ALBERT*
[22]bentrevett, *pytorch-seq2seq*
[23]PyTorch, *PyTorch*

Appendices

Appendix A - Example Predictions

A collection of predictions made by each of the three models. `Model PF` is the model that was pre-trained and fine-tuned, `Model F` is the model that was only fine-tuned and `Model P` is the model that had only undergone pre-training. It can be seen that in general, `Model F` outperforms `Model PF`, but only incrementally. Both create gibberish English that does not follow that target rhyming scheme (`AABBA`). Furthermore, `Model P` is often seen to form tiny sentences surrounding one or a few keywords. The first example demonstrates this statement.

Keywords lost depressing deny look dressing worry hell hurry shotgun girl
Model PF i've lost my new girlfriend, and i'm depressing. i deny it, look at my dressing. i worry, i'm hell, hurry, and my shotgun is a girl.
Model F i've lost my new girlfriend, and depressing, i deny that my look at the dressing. i worry, hell, i'm a hurry, and i'm gonna get my shotgun and girl.
Model P i lost my hair and depressing. i don't deny you. i look at you. i'm dressing up. i worry about hell. i'm in a hurry. i'm a shotgun. i'm a girl.

Keywords crown hope spread young girls comments scalding rather dead
Model PF tonsured the crown, i hope, spread the young, and the girls are scalding. i'm rather a dead-hearted guy. i'm a "i'm a man of a man."
F tonsured the crown of the hope that spread is young, and the girls are quite vile. but the girls are a scalding, and they're rather a scalding, and not dead.
Model P tonsured in the crown of hope, spread the young, and girls, scalding, rather than dead.

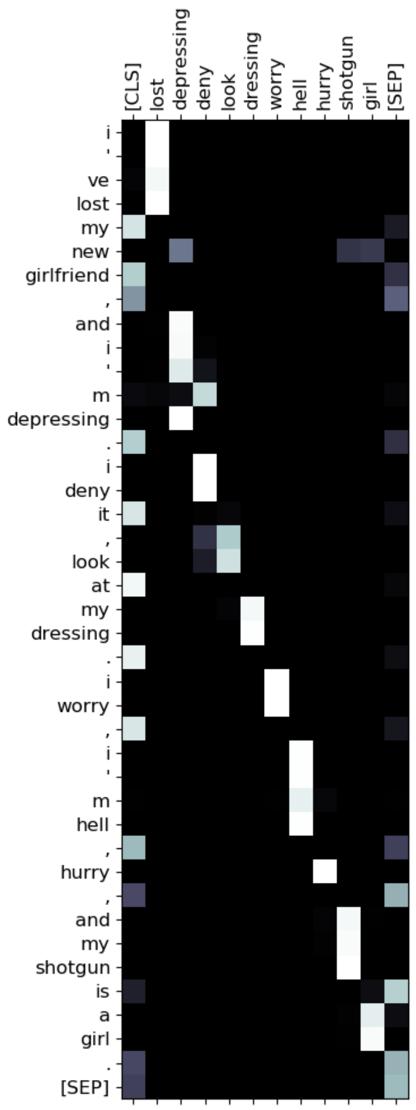
Keywords know prince head handle breathe word soul dead
Model PF i know that the prince's head is a handle, and i breathe. it's a word for my soul, but i'm dead. i'm a "i'm a man," i'm a man.
Model F i know that the prince was a head, and i handle the word that i'd breathe. it's a soul that i'll see, but i'm dead, and i'm not a big deal.
Model P i know the prince of my head is the handle of the sea. i breathe the word of my soul. i am dead.

Keywords complacency outlook frankly mess base decision data vision trust
Model PF complacency's outlook, frankly, a mess. it's a base. it's a decision to fix your data, and your vision will trust.
Model F complacency's outlook is frankly a mess. it's a base of decision. it's my data, my vision, and i trust my dear wife, and i'm not gonna get my vision.
Model P complacency is a outlook, frankly, a mess is base. it is a decision to be data, a vision, and trust.

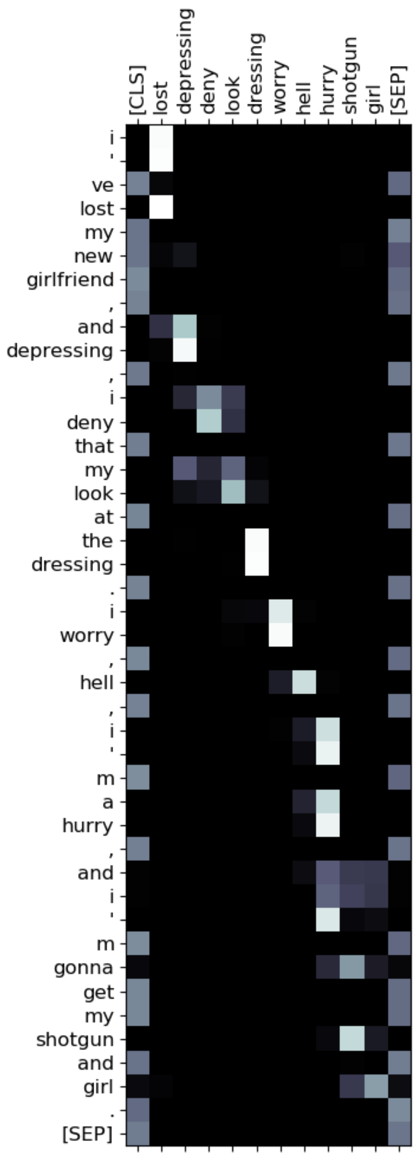
Keywords ferment today catch prefer suffused okay bent
Model PF ferment's today, i'll catch you, i prefer you to see. i'm suffused with a "okay," i'm bent. "i'm a "a"."
Model F fermentment's today, and i catch it suffused with the sun. i prefer it suffused with okay, but i'm bent on the end of the end of my bent.
Model P the ferment of today is to catch up and prefer to be suffused with the right thing. it's okay to be bent.

Appendix B - Heatmap of the Attention Layer

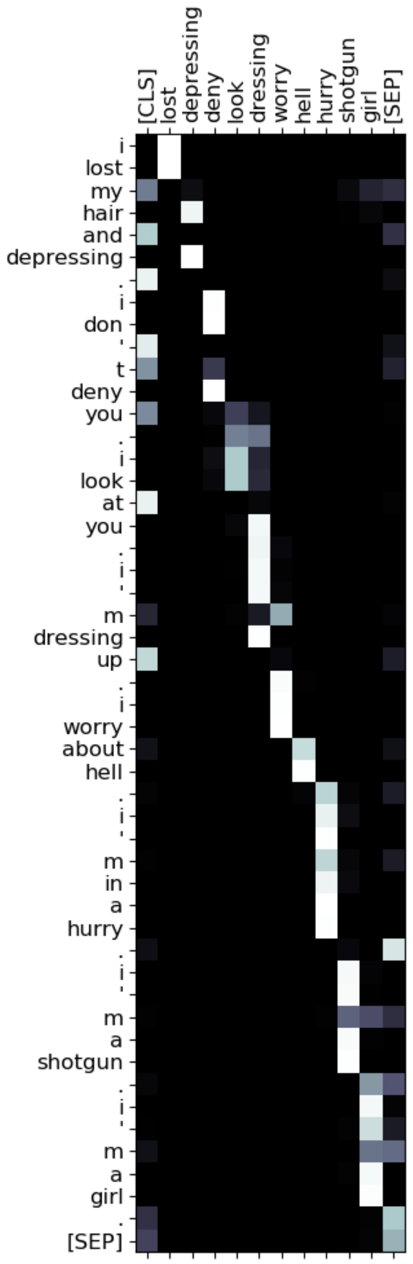
The figures below visualize the attention heatmap of the decoder's first head for each of the three models. The horizontal axes displays the keyword-input, where the vertical axis displays the model's prediction. As stated in `Appendix A`, `Model P` mainly relates each sentence to only one or two keywords, which is visualized by the vertical bars in the attention heatmap. Furthermore, a lot of noise is present in the `Model PF` but especially `Model F`, since it pays a lot of attention to the starting and closing token, which do hardly contain any meaning. These *noisy* predictions are often intermediate words and symbols, hence we can conclude that this noise in the attention layer is because the model doesn't need to relate to the inputs to base these predictions on.



(a) Model PF



(b) Model F



(c) Model P

Figure 3: Attention heatmap of the decoder's first head