Lexical ressources: Multilingual BERT, Exercise 1

Multilingual BERT - Exercise 1

The aim of this exercise is to familiarize you with handling textual data for BERT, as well as to introduce you with the popular technique of visualizing attention weights.

Attention weights generally give a clear explanation of how each representation from the previous layer contributes to the current vector representation. To understand where this idea comes from, recall the definition for attention:

$$A = softmax\,(\frac{Q \cdot K^T}{\sqrt{d_K}}) \cdot V$$

Note that if the Query matrix is of size $q \times d$, and the Key and Value matrices of size $n \times d$, the final output will be of size $Qq \times d$; ie. each query will be mapped to a d-dimensional vector. More importantly, remark that $softmax\,(\frac{Q \cdot K^T}{\sqrt{d_K}})$ is of size $q \times n$; and each row corresponds to a n-dimensional vector that sums to 1. This softmax is referred to as "attention weights"; and each row can be seen as a learned probability distribution over the n values.

Therefore, the output vector $\vec{a}_i$ of an attention mechanism for a given query q_i can be seen as a weighted sum of the values vectors V, which are a simple linear transformation of the output of the previous vector $\vec{o}^1_{L-1}, \ldots, \vec{o}^n_{L-1}$. The attention weights for q_i, namely $softmax\,(\frac{Q_i \cdot K^T}{\sqrt{d_K}})$ therefore directly indicate how much each representation of the previous layer weighs in the final output $\vec{a}_i$.

Hence, we can visualize the attention weights $softmax\,(\frac{Q \cdot K^T}{\sqrt{d_K}})$, and directly interpret how much each input representation $\vec{o}^j_{L-1}$ contributes to each output representation $\vec{a}_i$. This allows us to glean some insight on "what the model is actually doing".

In this exercise, you will learn to extract attention weights and visualize them. As this exercise is meant to be a bit more challenging, this time, there is no code canvas provided. Below is a step-by-step breakdown of how to do this, but not everything is clearly delineated. Keep in mind that you can always get some information on a specific object or function `some_object` from the python interpreter by typing `help(some_object)`.

1.1 Write a function to load a BERT model

There are multiple libraries that can load a BERT model. The easiest one to handle is probably the `transformers` library from Hugging Face. To install it, first make sure you have pytorch installed (run `pip install torch` otherwise), afterwards, you can install it using `pip install transformers`.

To load a model, you can use the function `BertModel.from_pretrained()`. This function accepts string names as BERT model identifiers: hence you can simply pass it the string *`"bert-base-multilingual-cased"`* to download and load the relevant BERT model. A list of valid model names is available here: https://huggingface.co/transformers/pretrained_models.html.

As we are going to work with attention weights, you will also have to use a specific configuration: you will need to use the function `BertConfig.from_pretrained(model_name)`, and specify that you want to retrieve attention weights using the keyword `output_attentions=True`. As an example (though slightly complicated), have a look the script `bertology.py` from Hugging Face's github.

1.2 Write a function to prepare an input string

As mentionned during the lecture, BERT uses byte-pair-encodings, as well as segment encodings. Converting strings into usable indices can be done using a `BertTokenizer` object. To use the BPE vocabulary from the original multilingual BERT model, you can use the `BertTokenizer.from_pretrained(model_name)` function.

Applying the learned BPE tokenization on a raw python string can be done with the method `tokenizer.tokenize(string)`, where `string` is the string you want to split in BPE tokens, and `tokenizer` is your `BertTokenizer` object. You can then convert these BPE tokens into indices (integer values which the model can use to retrieve input embeddings) with the function `tokenizer.convert_tokens_to_ids(tokens)`. You can append special tokens `[CLS]` and `[SEP]`, as well as obtain the correct segment encodings, with the function `tokenizer.prepare_for_model(ids)`. This function accepts an *optional* argument `pair_ids`, which you can fill in with a second sentence. The last remaining step is to convert lists into `torch.Tensor` objects. Note that you will need to convert flat lists of size n into tensors of dimension $1 \times n$.

1.3 Write a function to run the model on an input and return the attention weights

To run the model on a valid prepared input (cf. ex 1.2), you can simply use the syntax `model(prepared_input)`. The output of such a call should be tuple. If you've correctly configured the model (cf. exercise 1.1), the last element of this tuple should correspond to the attention weights for that sentence.

You might find it useful to study the quickstart guide from Hugging Face, that has a specific section on BERT: if you're encountering difficulties, you can check whether your input has the same shape as the quickstart example.

1.4 Write a function to visualize the attention weights

An intuitive visualization method is to use a heatmap plot: a $n \times n$ matrix table representation of the attention weights, where weights are translated into colors. The `matplotlib` python library can prove useful here, in particular, have a look at this tutorial. However, you don't *have* to use `matplotlib`: any other python visualisation library (eg. `seaborn`) should do the trick.

NB: Keep in mind that the `BertModel` outputs attention weights for *all* attention mechanisms. Each of the L layers contains a multihead attention sublayer, and each of the H heads of each multihead attention sublayer corresponds to a distinct attention mechanism.

1.5 Put everything together

Write a function that takes as input a raw string and produces a visualization of the attention weights. You should use the functions from exercises 1.2, 1.3 and 1.4.

Try not to load the BERT model and BERT tokenizer everytime you call this function, but only once!

1.6 Done early?

There are multiple things you can do to improve your script, and there are many things you can experiment. Here are a few possible directions:

- **Try to make your script user-friendly:** add a main entry point and command-line arguments.
- **Compare the attention weights per layer:** an observation that has been made recently is that not all layers do the same thing. Some attention mechanisms, in particular, seemingly keep track of syntax.
- **Compare the attention weights per BERT model:** do you get the same sort of results with multilingual BERT and other monolingual BERT models, such as the french CamemBERT?
- **Find which attention mechanisms truly matter:** some researchers have suggested that attention heads are not equally useful. For instance, have a look at this paper.