Transformers provides the following tasks which are most important features in NLP.

1. Sentiment analysis: is a text positive or negative?
2. Text generation (in English): provide a prompt and the model will generate what follows.
3. Name entity recognition (NER): in an input sentence, label each word with the entity it represents (person, place, etc.)
4. Question answering: provide the model with some context and a question, extract the answer from the context.
5. Filling masked text: given a text with masked words (e.g., replaced by mask) fill the blanks.
6. Summarization: generate a summary of a long text.
7. Translation: translate a text in another language.
8. Feature extraction: return a tensor representation of the text.
9. Next Sentence Prediction.

Libraries will play’s vital role in transformers. The libraries are built around three types of classes for each model.

* **Model classes** such as Bert Model, which are 30+ PyTorch models or Keras models that work with the pretrained weights provided in the library.
* **Configuration classes** such as BertConfig, which store all the parameters required to build a model. You don’t always need to instantiate these yourself. If you are using a pretrained model without any modification, creating the model will automatically take care of instantiating the configuration (which is part of the model).
* **Tokenizer classes** such as BertTokenizer, which store the vocabulary for each model and provide methods for encoding/decoding strings in a list of token embeddings indices to be fed to a model.

All these classes can be instantiated from pretrained instances and saved locally using two methods:

* **from\_pretrained ()** lets you instantiate a model/configuration/tokenizer from a pretrained version either provided by the library itself (the supported models are provided in the list here or stored locally (or on a server) by the user,
* **save\_pretrained ()** lets you save a model/configuration/tokenizer locally so that it can be reloaded using **from\_pretrained ()**.

 General Terms with respect to Transformers:

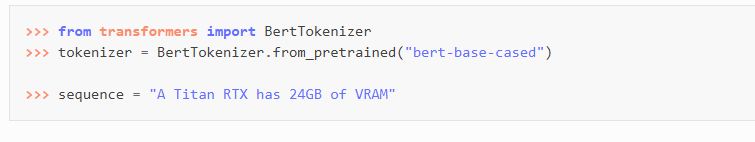
* **CLM**: causal language modeling, a pretraining task where the model reads the texts in order and has to predict the next word. It’s usually done by reading the whole sentence but using a mask inside the model to hide the future tokens at a certain timestep.
* **MLM**: masked language modeling, a pretraining task where the model sees a corrupted version of the texts, usually done by masking some tokens randomly, and has to predict the original text.
* **multimodal**: a task that combines texts with another kind of inputs (for instance images).
* **NLG**: natural language generation, all tasks related to generating text (for instance talk with transformers, translation).
* **NLP**: natural language processing, a generic way to say, “deal with texts”.
* **NLU**: natural language understanding, all tasks related to understanding what is in a text (for instance classifying the whole text, individual words)
* **pretrained model**: a model that has been pretrained on some data (for instance all of Wikipedia). Pretraining methods involve a self-supervised objective, which can be reading the text and trying to predict the next word (see CLM) or masking some words and trying to predict them (see MLM).
* **RNN**: recurrent neural network, a type of model that uses a loop over a layer to process texts.
* **seq2seq or sequence-to-sequence**: models that generate a new sequence from an input, like translation models, or summarization models.
* **token**: a part of a sentence, usually a word, but can also be a subword (non-common words are often split in subwords) or a punctuation symbol.

**Model Inputs:**

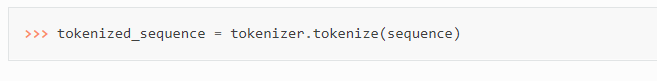
Model inputs contains input IDs, Attention mask, token IDs, Position IDs and Feed forward chunking.

**Input IDs:** The input ids are often the only required parameters to be passed to the model as input. They are token indices, numerical representations of tokens building the sequences that will be used as input by the model*.*

Each tokenizer works differently but the underlying mechanism remains the same. Here’s an example using the BERT tokenizer, which is a Word Piece tokenizer.



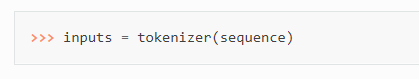
The tokenizer takes care of splitting the sequence into tokens available in the tokenizer vocabulary.



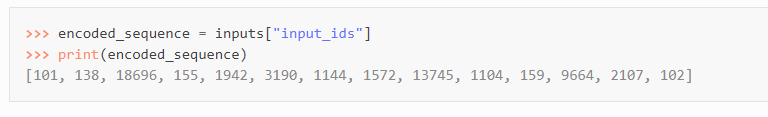
The tokens are either words or subwords. Here for instance, “VRAM” wasn’t in the model vocabulary, so it’s been split in “V”, “RA” and “M”. To indicate those tokens are not separate words but parts of the same word, a double-hash prefix is added for “RA” and “M”:



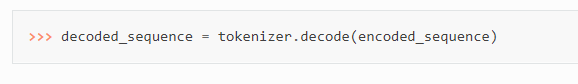
These tokens can then be converted into IDs which are understandable by the model. This can be done by directly feeding the sentence to the tokenizer.

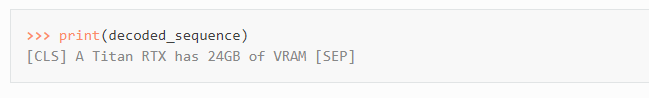


The tokenizer returns a dictionary with all the arguments necessary for its corresponding model to work properly. The token indices are under the key “input\_ids”:



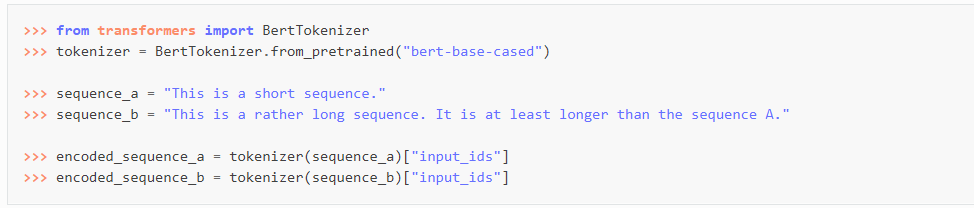
If we decode the previous sequence of ids,



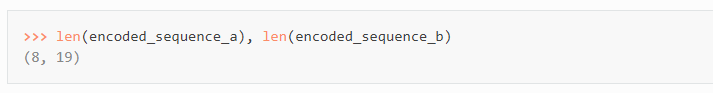


This is how Bert Model is going to expect its inputs.

**Attention Mask**: The attention mask is an optional argument used when batching sequences together. This argument indicates to the model which tokens should be attended to, and which should not. Let’s take example below:

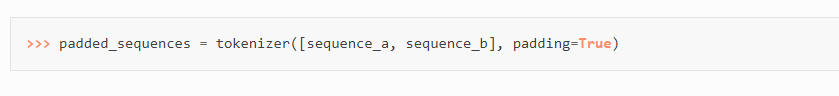


Here encoded versions have different length.

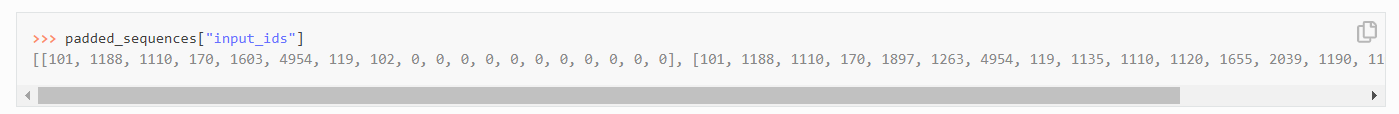


Therefore, we can’t have put them together in a same tensor as-is. The first sequence needs to be padded up to the length of the second one, or the second one needs to be truncated down to the length of the first one.

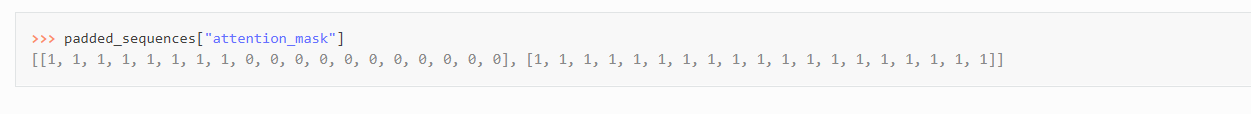
In the first case, the list of IDs will be extended by the padding indices. We can pass a list to the tokenizer and ask it to pad like this:



We can see that 0s have been added on the right of the first sentence to make it the same length as the second one:



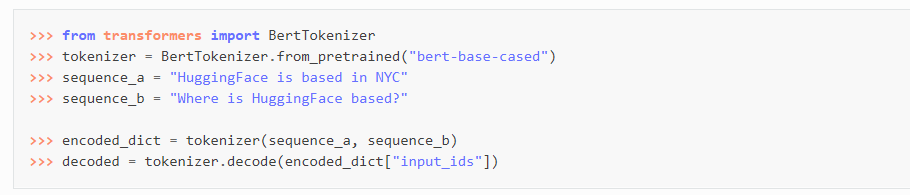
This can then be converted into a tensor in PyTorch or TensorFlow. The attention mask is a binary tensor indicating the position of the padded indices so that the model does not attend to them. For Bert Tokenizer, 1 indicates a value that should be attended to, while 0 indicates a padded value. This attention mask is in the dictionary returned by the tokenizer under the key “attention mask”:



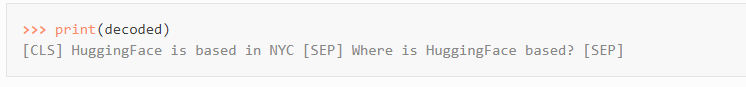
**Token Type IDs:** Some models’ purpose is to do sequence classification or question answering. These require two different sequences to be joined in a single “input\_ids” entry, which usually is performed with the help of special tokens, such as the classifier (CLS) and separator (SEP) tokens.



We can use our tokenizer to automatically generate such a sentence by passing the two sequences to tokenizer as two arguments.



Finally, it will return



This is enough for some models to understand where one sequence ends and where another begins. However, other models, such as BERT, also deploy token type IDs (also called segment IDs). They are represented as a binary mask identifying the two types of sequence in the model. The tokenizer returns this mask as the “token\_type\_ids” entry:



The first sequence, the “context” used for the question, has all its tokens represented by a 0, whereas the second sequence, corresponding to the “question”, has all its tokens represented by a 1.

**Position IDs:** The position IDs are used by the model to identify each token’s position in the list of tokens. They are an optional parameter. If no positional\_ids are passed to the model, the IDs are automatically created as absolute positional embeddings.

**Feed forward Chunking:** In each residual attention block in transformers the self-attention layer is usually followed by 2 feed forward layers. The intermediate embedding size of the feed forward layers is often bigger than the hidden size of the model.