Université Paris Cité

How Multilingual is Multilingual BERT?

Guillaume Wisniewski
guillaume.wisniewski@u-paris.fr

October 2022

1 Introduction

The goal of this lab, in addition to answering a very interesting question (at least to me), is to teach you how to use the `HuggingFace` API to fine-tune a pretrained language models. `HuggingFace` is a start-up that allows us to :

— easily download datasets and pre-trained models ;
— consistent API to fine-tune a model on a specific task.

`HuggingFace` provides an additional layer to encapsulate model implementations (in pytorch or tensorflow) and make their use easier. It is today the *de facto* standard to disseminate a corpus or a model and to fine-tune pre-trained models.

Rather than preparing the corpora for you in a suitable format, I have decided to have you transform the original data so that you have an idea of the complexity of certain fundamental steps that are often overlooked (typically tokenization), particularly in a multilingual context. (This is an understatement to warn you that some technical questions may not be trivial and that you will have to write "interesting" code.)

For this lab, you need to install (at least) :

— the `transformers` and `datasets` libraries of `HuggingFace` : `https://huggingface.co/docs/transformers/index`
— the `spacy` library : `https://spacy.io/`

2 Fine-Tuning `mBERT`

The goal of the first part of this lab is to train (and evaluate) a multilingual PoS tagger by fine-tuning `mBERT`. To assess the multilingual capacity of `mBERT` we (you !) will :

— evaluate the performance of a PoS tagger fine-tuned on a given language and compare it to the performance of a PoS tagger trained only on this language ;
— evaluate on a language B the performance of a PoS tagger fine-tuned on a language A.

```
1  import conllu
2
3  def load_conllu(filename):
4      for sentence in conllu.parse(open(filename, "rt", encoding="utf-8").read()):
5          tokenized_words = [token["form"] for token in sentence]
6          gold_tags = [token["upos"] for token in sentence]
7          yield tokenized_words, gold_tags
8
9  corpus = list(load_conllu("fr_sequoia-ud-test.conllu"))
```

FIGURE 1 – Extracting PoS information from a `conllu` file. The code assumes you have donwloaded the `fr_sequoia-ud-test.conllu` file and store it the same directory as your code (see line 4).

2.1 The Universal Dependencies Project

The universal dependencies project is a collection of treebanks with consistent annotations of grammar (parts of speech, morphological features, and syntactic dependencies) across different human language.

1. Why do we need "consistent annotation" for our experiment ?

Treebanks of the UD project are stored in the `conllu` format that is notoriously difficult to parse, even if the work is partly facilitated by the `conllu` [1] module which allows to read a file in this format. Figure 1 shows how to extract (tokenized) sentence ans PoS information from `conllu` files.

2. What do we use the **`yield`** keyword rather than a simple **`return`** (line 7) ? Why do we have to call the `list` constructor (line 9) ?
3. What is the distribution of labels in the test set of the `Sequoia` treebank ?

You can see [2] that there is a "special" label _ that results from the tokenization of *multiword token* : words like *au* in French or *dámelo* in Spanish that correspond to multiple syntactic words are tokenized into several token (e.g. *au* is tokenized in *à le*, *dámelo* in *da me lo*).

4. What are the multiword tokens in the test set of `Sequoia` ? How are they annotated ?
5. Why did the UD project made the decision of splitting multiword tokens into several (grammatical) words ? What is your opinion on this decision ? [3]
6. According to UD tokenization guidelines, a token can contain spaces. Are there any token in the `Sequoia` corpora that contain spaces ? If so these spaces should be removed.

1. `https://pypi.org/project/conllu/`
2. This is why you should **always** start a new project by looking at the distribution of labels and compute other descriptive statistics on your corpus.
3. It is advisable to answer this question only after completing the code for this section.

2.2 mBERT tokenization

mBERT (as well a the original BERT model) uses a tokenization in subword units. Frequent words are tokenized "normally", but infrequent words are split in smaller factors which sometimes correspond to affixes. In addition, punctuation is split at the character level. Note that, for mBERT subword tokens are prefixed with ##.

7. Why does mBERT use a subword tokenization ?

Figure 2 shows how a sentence can be tokenized by mBERT tokenizer. This code uses the HuggingFace API which allows easy access to already trained models (here mBERT) and to the different components needed to use them (here the corresponding tokenizer) : you simply have to provide a *model checkpoint* and HuggingFace will download all the required models and encapsulate their use in a high-level API. In the following, we will use the bert-base-multilingual-cased checkpoint. Other models can be easily found (with their documentation) on the *model hub* of HuggingFace.

As shown in Figure 2, the tokenizer can be used to output either the tokenized sentence in a "readable" format or to compute the information needed by the model (the attention mask and the representation of the sentence as a list of integers).

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-multilingual-cased")
# show the resulting tokenization
tokenizer.tokenize("What a wonderful world!")
# compute information required by the model
tokenizer("What a wonderful world!")
```

FIGURE 2 – Tokenizing a single sentence with mBERT tokenizer.

8. How is the sentence "Pouvez-vous donner les mêmes garanties au sein de l'Union Européene" [4] tokenized according to the UD convention ? How is it tokenized by mBERT tokenizer ?
9. Why is this difference in tokenization a problem for training a mBERT-based PoS tagger ?

2.3 Reconciling the Two Tokenizations

To train a PoS tagger, we have to provide a PoS label to each token of a sentence tokenized with mBERT tokenizer. For this we propose to implement the following two principles :

4. This sentence is made of *n*-grams from the test set of the Sequoia corpus.

— multiword tokens that have to be split according to UD annotation guidelines are kept unchanged and labelled by a new label made of the concatenation of all the label of its token. For instance the French word *au* is not decomposed into *à* (an ADP) and *le* (a DET) but labelled by ADP+DET ;
— For UD tokens that are subtokenized into several units by `mBERT`, we choose to align the tag with the first subtoken of the word and convert the rest to the prediction of a special <pad> tag. The <pad> tag will have an other role when batching sentences together and we will ignore it when evaluating the accuracy of the predicted tags.

Figure 3 shows the tokenized sentence and corresponding labels that results from the application of these two principles.

raw sentence `Revenons-en aux choses essentielles.`

UD tokenization `Revenons -en à les choses essentielles .`

UD original labels `VERB PRON ADP DET NOUN ADJ PUNCT`

UD "updated" tokenziation `Revenons -en aux choses essentielles .`

UD "updated" labels `VERB PRON ADP+DET NOUN ADJ PUNCT`

mBERT tokenization `Rev ##eno ##ns - en aux choses essentiel ##les .`

labels for mBERT tokenization `VERB <pad> <pad> PRON <pad> ADP+DET ADJ <pad> PUNCT`

FIGURE 3 – Example of tokenization alignment.

10. Write a function that takes as parameter a sentence tokenized according to UD rules (i.e. a variable of type `List[str]` and its gold labels (also a variable of type `List[str]`) and implements the first principle explained above. [5]

To implement the second principle, we can set the `return_offsets_mapping` parameter of the tokenizer to `True`. For each sub-token returned by the tokenizer, the offset mapping gives us a tuple indicating the sub-token start position and end position relative to the original token it was split from. That means that if the first position in the tuple is anything other than 0, the corresponding token corresponds to the "continuation" of word and we have to set its corresponding label to <pad>. If both the first position and the second position in the tuple are 0, the token is a special token like <pad> and <cls> and the corresponding label must also be <pad>.

It is also necessary to pad the sentence to make batching more efficient. At the end the tokenizer must be called with :

```
tokenizer(texts, is_split_into_words=True,
                 return_offsets_mapping=True,
```

5. You might want to look at the information returned by the `conllu.read` method.

```
                    padding=True,
                    truncation=True)
```

Note that, in this code, `texts` is a either a *list* of list of strings (i.e. each sentence is split into a list of words) corresponding to the sentences of the corpus.

11. Write a function that takes as input the sentences and labels created by "normalizing" UD sentences, apply the `mBERT` tokenizer and compute the corresponding label. You must ensure each subtoken (including padding symbols) has a label.

In `HuggingFace`, labels must be represented by integer, and the special <pad> label must be mapped to -100.

12. Write a function that encodes the labels into integers.

2.4 Creating a `Dataset`

Now that we have a labeled dataset adapted to `mBERT` tokenization, we can convert the data into a format suitable for training. First, we need to encapsulate our dataset into an instance of the `Dataset` class of the `HuggingFace` library. A `Dataset` is a `DataFrame` and can be seen as a list of dictionaries, each dictionary describing an example. Note that the keys of these dictionaries we be used to identify arguments when calling the model (and can not be chosen freely). In the case of `mBert`, you need to pass the following keys :

— `input_ids`
— `attention_mask`
— `labels`

13. Using the `Dataset.from_list` method, write a method that creates a `Dataset` that encapsulates a corpus in the `conllu`.
14. Create three instances of `Dataset` : one for the train set, one for the dev set and the last one for the test set.

2.5 Fine-Tuning `mBERT`

Figure 4 shows how a model can be fine-tuned very simply using `HuggingFace` API. All arguments are encapsulated in a `TrainingArguments` instance (these parameters are supposed to be optimized !) and, once these parameters are chosen fine-tuning is as simple as calling a single method (`train`).

15. How can you modify the code of Figure 4 to report the PoS tagging accuracy during optimization. Why is this information important ?

```
batch_size = 16

from transformers import AutoModelForTokenClassification, TrainingArguments, Trainer

model_checkpoint = "bert-base-multilingual-cased"
model = AutoModelForTokenClassification.from_pretrained(model_checkpoint, num_labels=len(label_list))

training_args = TrainingArguments(
    output_dir="to_be_chosen",
    evaluation_strategy = "epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size,
    num_train_epochs=3,
    weight_decay=0.01,
    logging_dir='./logs',
    logging_steps=10,
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=ds,       # the train set
    eval_dataset=ds         # the dev set
    )

trainer.train()
```

FIGURE 4 – Code to fine-tune a model with `HuggingFace`.

3 Evaluating the multilingual capacity of mBERT

16. Choose 5 languages from the UD project. For each language train a PoS tagger using the code of the previous section and test it on the 5 languages you have chosen.
17. Can you use the same hyperparameters for the different languages ?
18. What can you conclude ?