# A Tutorial for using BERT for the First Time

Prepared by Jiashuo Wang
Tuesday, March 9, 2021

BERT, which stands for *Bidirectional Encoder Representations from Transformers*, is a language representation model designed by Google in 2018. Unlike previous language representation models, BERT is designed to pretrain deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. Therefore, we can fine-tune BERT with just one additional output layer to create state-of-art models for a wide range of tasks, such as text classification, and question answering. I really recommend that you could read the original paper. There are many packages and libraries containing BERT model, and *transformers* is one of them, which is from Hugging Face. *Transformers* provides general-purpose architectures (BERT, GPT, XLM…) for Natural Language Understanding (NLU) and Natural Language Generation (NLG). In this tutorial, we will learn how BERT works and how to use BERT with *transformers* from installation. Here is the link of the documentation.

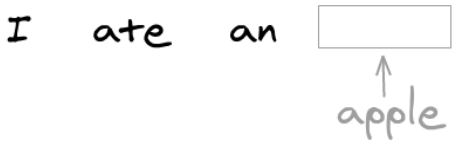
## 1. How BERT Works

BERT is trained with two unsupervised learning tasks intuitively designed to pre-train words and sentence representation, which is the core of the BERT algorithm.

### Task 1: Masked Language Model (MLM)

Most probabilistic language models can only be trained in a directional way, where the target word is predicted only by preceding words or following words. Although it is possible to combine two language models which are trained from left to right and from right to left, the new model still cannot use effectively the bidirectional context information. It lacks the ability to simultaneously "see" bidirectional words. Therefore, there is a need to modify the language models to predict the target word according to both left and right words.

Intuitively, the strategy is easy that we just need to mask a certain proportion of the words randomly in a sentence, and then use those masked words as the predicting targets instead of predicting next word. The final layer is a softmax layer to convert hidden representation to normalized probability distribution over the vocabulary.

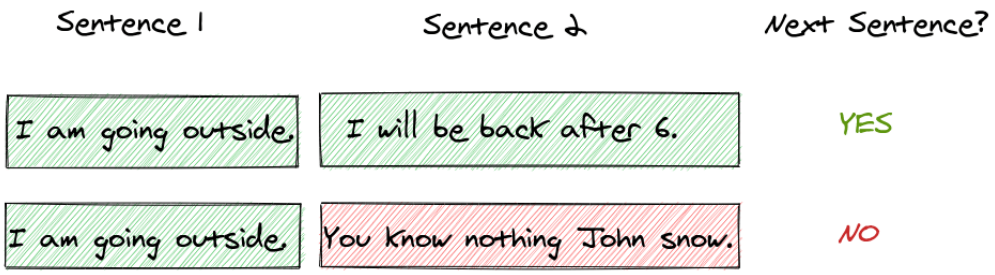



### Task 2: Next Sentence Prediction (NSP)

There are many NLP tasks based on the understanding the relationship between two sentences, but this information cannot be directly captured by previous pretraining algorithms. General pre-

training methods only get word representation without sentence representation which can be used for sentence classification tasks.

Intuitively, a new task is required to capture sentence-level information. Its objective is to generate a vector used for classifying sentences. Following the idea of language model which can obtain word-level representation by maximizing the likelihood estimation of a sequence of words, the new task may be to compute the similar likelihood estimation of the sequence of sentences. For simplicity, the BERT algorithm only considers the case that the sequence only has two sentences, i.e. to predict whether one sentence is the next sentence of the other sentence.




## 2. Installation

Let's install the transformers package from Hugging Face, which gives us a PyTorch interface for working with BERT. I'll show two ways to install *transformers*.

### 2.1. Installation with pip

When PyTorch has been installed, we can use pip to install *transformers* directly:

```
pip install transformers
```

Alternatively, for CPU-support only, you can install *transformers* and PyTorch in one line with:

```
pip install transformers[torch]
```

To check whether *transformer* is properly installed, run the following command:

```
python -c "from transformers import pipeline; print(pipeline('sentiment-analysis')('we love you'))"
```

If your operations are correct, you could get such results:

```
[{'label': 'POSITIVE', 'score': 0.9998704791069031}]
```

### 2.2. Installation from Source

Here is how to quickly install *transformers* from source:

```
pip install git+https://github.com/huggingface/transformers
```

## 3. Tokenization and Input Formatting

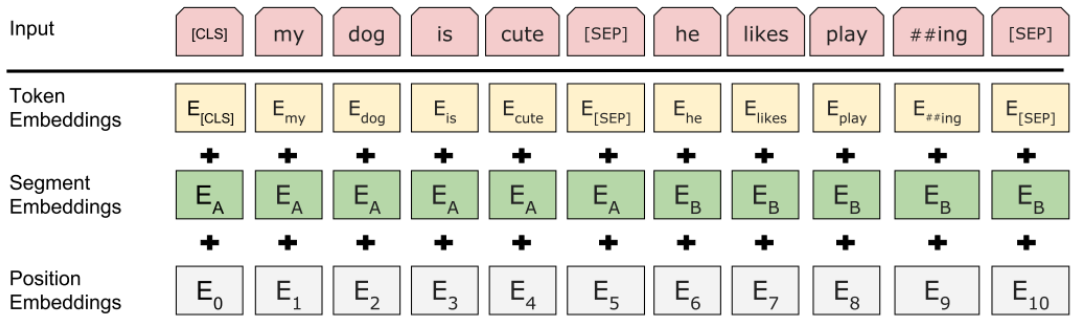
In this step, we'll transform our dataset into the format that BERT can be trained on.

### 3.1. Special Tokens

Before tokenizing a sequence with BERT, let's first have a look at some special tokens in BERT. There are 3 special tokens, which are [CLS], [SEP], and [PAD] respectively.

***[CLS] and [SEP]***

The following picture is from the original paper of BERT. It is how to train BERT with NSP. Here we can observe the position of [CLS]and [SEP] in the model. [CLS] is at the begging of the sentences, which [SEP] is in the middle of these two sentences.




For the classification task, a **single** vector representing the whole input sentence is needed to be fed to a classifier. In BERT, the decision is that the hidden state of the **first token** is taken to represent the whole sentence. To achieve this, an additional token has to be added manually to the input sentence. In the original implementation, the token [CLS] is chosen for this purpose.

In the "next sentence prediction" task, we need a way to inform the model where does the **first sentence end**, and where does the **second sentence begin**. Hence, another artificial token, [SEP], is introduced. If we are trying to train a classifier, each input sample will contain only one sentence (or a single text input). In that case, the [SEP] token will be added to the end of the input text.

In summary, to preprocess the input text data, the first thing we will have to do is to add the [CLS] token at the beginning, and the [SEP] token at the end of each input text.

***Padding Token [PAD]***

The BERT model receives a fixed length of sentence as input. Usually the maximum length of a sentence depends on the data we are working on. For sentences that are shorter than this maximum length, we will have to add paddings (empty tokens) to the sentences to make up the length. In the original implementation, the token [PAD] is used to represent paddings to the sentence.

### 3.2. BERT Tokenizer

The BERT Tokenizer is a tokenizer that works with BERT. It has many functionalities for any type of tokenization tasks. You can download the tokenizer using the line of code:

```
from transformers import BertTokenizer
```

```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
sent = “He remains characteristically confident and optimistic.”
encoding = tokenizer.encode_plus(sent, add_special_tokens = True, truncation = True, padding = "longest", return_attention_mask = True, return_tensors = "pt")
```

We can get the following results:

```
>>> encoding = tokenizer.encode_plus(sent, add_special_tokens = True, truncation = True, padding = "longest", return_att
ention_mask = True, return_tensors = "pt")
>>> encoding
{'input_ids': tensor([[  101,  2002,  3464,  8281,  3973,  9657,  1998, 21931,  1012,   102]]), 'token_type_ids': tensor
([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]), 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])}
>>>
```

For the meaning of each parameter, we can use help(transformers) in python to check it:

```
Help on BertTokenizer in module transformers.models.bert.tokenization_bert object:

class BertTokenizer(transformers.tokenization_utils.PreTrainedTokenizer)
 |  BertTokenizer(vocab_file, do_lower_case=True, do_basic_tokenize=True, never_split=None, unk_token='[UNK]', sep_token
='[SEP]', pad_token='[PAD]', cls_token='[CLS]', mask_token='[MASK]', tokenize_chinese_chars=True, strip_accents=None, **
kwargs)
 |
 |  Construct a BERT tokenizer. Based on WordPiece.
 |
 |  This tokenizer inherits from :class:`~transformers.PreTrainedTokenizer` which contains most of the main methods.
 |  Users should refer to this superclass for more information regarding those methods.
 |
 |  Args:
 |      vocab_file (:obj:`str`):
 |          File containing the vocabulary.
 |      do_lower_case (:obj:`bool`, `optional`, defaults to :obj:`True`):
 |          Whether or not to lowercase the input when tokenizing.
 |      do_basic_tokenize (:obj:`bool`, `optional`, defaults to :obj:`True`):
 |          Whether or not to do basic tokenization before WordPiece.
 |      never_split (:obj:`Iterable`, `optional`):
 |          Collection of tokens which will never be split during tokenization. Only has an effect when
 |          :obj:`do_basic_tokenize=True`
 |      unk_token (:obj:`str`, `optional`, defaults to :obj:`"[UNK]"`):
 |          The unknown token. A token that is not in the vocabulary cannot be converted to an ID and is set to be this
 |          token instead.
 |      sep_token (:obj:`str`, `optional`, defaults to :obj:`"[SEP]"`):
 |          The separator token, which is used when building a sequence from multiple sequences, e.g. two sequences for
 |          sequence classification or for a text and a question for question answering. It is also used as the last
 |          token of a sequence built with special tokens.
 |      pad_token (:obj:`str`, `optional`, defaults to :obj:`"[PAD]"`):
 |          The token used for padding, for example when batching sequences of different lengths.
 |      cls_token (:obj:`str`, `optional`, defaults to :obj:`"[CLS]"`):
 |          The classifier token which is used when doing sequence classification (classification of the whole sequence
 |          instead of per-token classification). It is the first token of the sequence when built with special tokens.
 |      mask_token (:obj:`str`, `optional`, defaults to :obj:`"[MASK]"`):
 |          The token used for masking values. This is the token used when training this model with masked language
 |
-- More  --
```

## 4. Tasks

In this section, let’s have a look at how to use BERT in some specific tasks. BERT can be applied to different tasks easily with fine-tuning. Here, I will show the detailed steps and explanation of text classification and give steps directly for other tasks. The code can be download from google drive.

### 4.1. Text Classification

Text classification can be applied in a wide range of scenarios, such as sentimental classification or decide whether an possible answer is the correct one of the given question. In this task, we use Yelp reviews-polarity. This task aims at predict the sentimental polarity of the input review.

The first step to do a task is preparing data. The data should first be processed in the format as *label text*. I recommend *pandas* to read and process the data.

```
from transformers import BertTokenizer
import pandas as pd
train_df = pd.read_csv('train.csv', header=None)
```

We can have a look at first several lines of the data by train_df.head():

```
>> train_df = pd.read_csv('train.csv', header=None)
>> train_df.head()
  0                                                  1
1  Unfortunately, the frustration of being Dr. Go...
2  Been going to Dr. Goldberg for over 10 years. ...
1  I don't know what Dr. Goldberg was like before...
1  I'm writing this review to give you a heads up...
2  All the food is great here. But the best thing...
```

The first column indicates the labels, where 1 means bad review and 2 means good review, and the second column indicates the text. To make the labels more familiar 0/1 labelling, I change the good reviews labeled as 1 and bad reviews labeled as 0.

```
train_df[0]=(train_df[0]==2).astype(int)
```

To make things a little BERT-friendly, we use *tsv* to store a new dataframe:

```
train_df_bert = pd.DataFrame({
        'id':range(len(train_df)),
        'label': train_df[0],
        'alpha': ['a']*train_df.shape[0],
        'text': train_df[1].replace(r'\n', ' ', regex=True)
})
```

```
>>> train_df_bert.head()
   id  label alpha                                               text
0   0      1     a  Unfortunately, the frustration of being Dr. Go...
1   1      2     a  Been going to Dr. Goldberg for over 10 years. ...
2   2      1     a  I don't know what Dr. Goldberg was like before...
3   3      1     a  I'm writing this review to give you a heads up...
4   4      2     a  All the food is great here. But the best thing...
>>>
```

```
train_df_bert.to_csv('train.tsv', sep='\t', index=False, header=False)
```

We do the same operation to *dev.csv* and get *dev.tsv*.

Then we fine-tune BERT on our task. To fine-tune BERT, we first need to understand the original code, and we can clone google/bert through

```
git clone https://github.com/google-research/bert.git
```

To fine-tune BERT, we need to modify run_classifier.py, which is shown below:

```
class DataProcessor(object):
  """Base class for data converters for sequence classification data sets."""

  def get_train_examples(self, data_dir):
    """Gets a collection of `InputExample`s for the train set."""
    raise NotImplementedError()

  def get_dev_examples(self, data_dir):
    """Gets a collection of `InputExample`s for the dev set."""
    raise NotImplementedError()

  def get_test_examples(self, data_dir):
    """Gets a collection of `InputExample`s for prediction."""
    raise NotImplementedError()

  def get_labels(self):
    """Gets the list of labels for this data set."""
    raise NotImplementedError()

  @classmethod
  def _read_tsv(cls, input_file, quotechar=None):
    """Reads a tab separated value file."""
    with tf.gfile.Open(input_file, "r") as f:
      reader = csv.reader(f, delimiter="\t", quotechar=quotechar)
      lines = []
      for line in reader:
        lines.append(line)
      return lines
```

Besides, we should also define a *BinaryClassificationProcessor* similar with *XXXProcessor*, such as *MnliProcess*:

```
class BinaryClassificationProcessor(DataProcessor):
    """Processor for binary classification dataset."""

    def get_train_examples(self, data_dir):
        """See base class."""
        return self._create_examples(
            self._read_tsv(os.path.join(data_dir, "train.tsv")), "train")

    def get_dev_examples(self, data_dir):
        """See base class."""
        return self._create_examples(
            self._read_tsv(os.path.join(data_dir, "dev.tsv")), "dev")

    def get_labels(self):
        """See base class."""
        return ["0", "1"]

    def _create_examples(self, lines, set_type):
        """Creates examples for the training and dev sets."""
        examples = []
        for (i, line) in enumerate(lines):
            guid = "%s-%s" % (set_type, i)
            text_a = line[3]
            label = line[1]
            examples.append(
                InputExample(guid=guid, text_a=text_a, text_b=None, label=label))
        return examples
```

With the modified code, we can train the model after setting hyper-parameters:

```
if __name__ == '__main__':
    print(f'Preparing to convert {train_examples_len} examples..')
    print(f'Spawning {process_count} processes..')
    with Pool(process_count) as p:
        train_features = list(tqdm_notebook(p.imap(convert_examples_to_features.convert_example_to_feature, train_examples_for_processing), total=train_examples_len))
    model = BertForSequenceClassification.from_pretrained(BERT_MODEL, cache_dir=CACHE_DIR, num_labels=num_labels)
    model.to(device)
    param_optimizer = list(model.named_parameters())
    no_decay = ['bias', 'LayerNorm.bias', 'LayerNorm.weight']
    optimizer_grouped_parameters = [
        {'params': [p for n, p in param_optimizer if not any(nd in n for nd in no_decay)], 'weight_decay': 0.01},
        {'params': [p for n, p in param_optimizer if any(nd in n for nd in no_decay)], 'weight_decay': 0.0}
    ]
    optimizer = BertAdam(optimizer_grouped_parameters,
                         lr=LEARNING_RATE,
                         warmup=WARMUP_PROPORTION,
                         t_total=num_train_optimization_steps)

    global_step = 0
    nb_tr_steps = 0
    tr_loss = 0
```

The completed code can be download from google drive.

### 4.2. Question Answering

We use SQuAD 2.0 (train, validation) for this task. This task aims at extract target answer from the given article.

> **Article:** Endangered Species Act
> **Paragraph:** " *. . . Other legislation followed, including the Migratory Bird Conservation Act of 1929, a 1937 treaty prohibiting the hunting of right and gray whales, and the Bald Eagle Protection Act of 1940. These later laws had a low cost to society—the species were relatively rare—and little opposition was raised.*"
>
> **Question 1:** "*Which laws faced significant opposition?*"
> **Plausible Answer:** *later laws*
>
> **Question 2:** "*What was the name of the 1937 treaty?*"
> **Plausible Answer:** *Bald Eagle Protection Act*

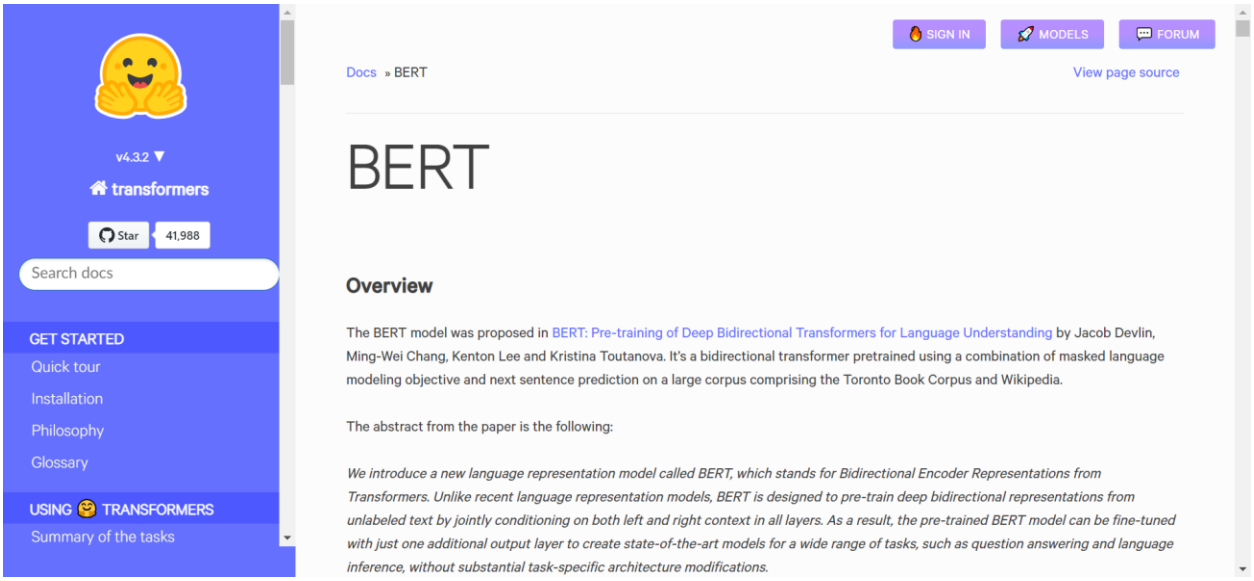
We can also do fine-tuning on SWAG, which is a multiple choice problem.

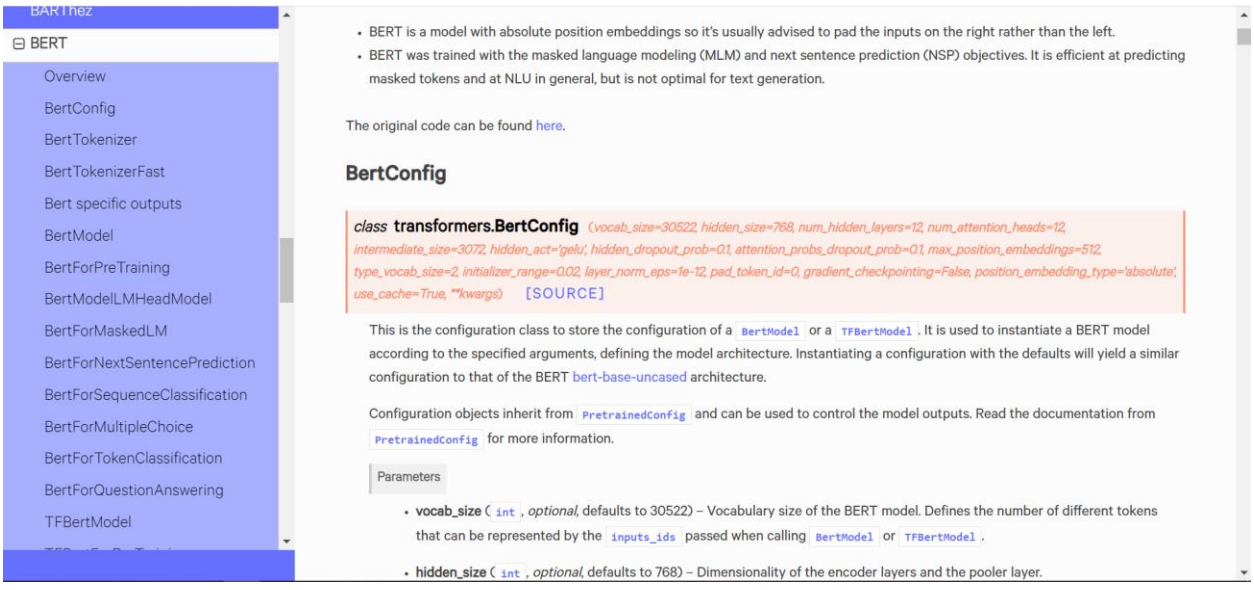
```
python ./examples/run_multiple_choice.py \
--model_type roberta \
--task_name swag \
--model_name_or_path roberta-base \
--do_train \
--do_eval \
--do_lower_case \
--data_dir SWAG_DICT \
--learning_rate 5e-5 \
--num_train_epochs 3 \
--max_seq_length 80 \
--output_dir models_bert/swag_base \
--per_gpu_eval_batch_size=16 \
```

```
--per_gpu_train_batch_size=16 \
--gradient_accumulation_steps 2 \
--overwrite_output
```

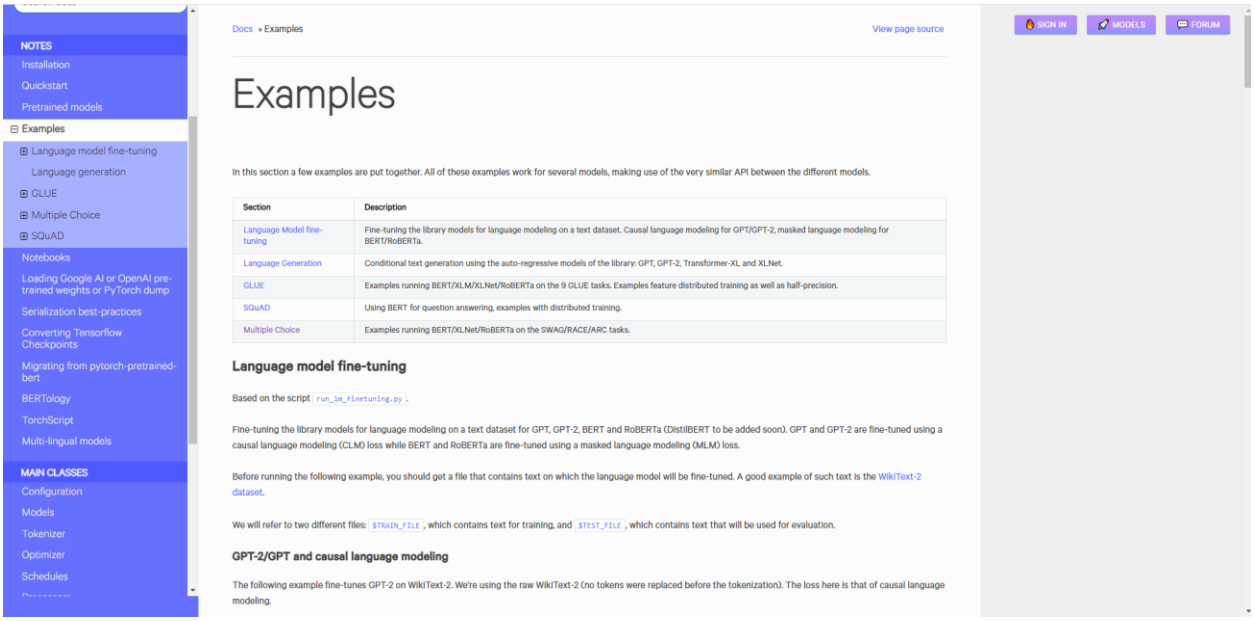
### 5. How to search a function in the documentation of *transformers*.

At last, I want to share how to search a function in the documentation of *transformers*. Sometimes, we may want to more functions to deal with various problem. In this case, we can scroll down the bar at the left of the documentation and click into the function that we are interested in and have a look at the details.




Besides, there are many other applications of BERT and other pretrained models, such as *roberta*. You can get detailed instruction from the *transformers* documentation.




One thing worth mentioning, *transformers* is just one package for BERT, and you can also explore other package such as *simple transformers*.