Chapter 15
Workshop#6 Transformers with spaCy and TensorFlow (Hour 11–12)

15.1 Introduction

In Chap. 8, the basic concept about Transfer Learning, its motivation and related background knowledge such as Recurrent Neural Networks (RNN) with Transformer Technology and BERT model are introduced.

This workshop will learn about the latest topic Transformers in NLP, how to use them with TensorFlow and spaCy. First, we will learn about Transformers and Transfer learning. Second, we will learn about a commonly used Transformer architecture—Bidirectional Encoder Representations from Transformers (BERT) as well as to how BERT Tokenizer and WordPiece algorithms work.

Further, we will learn how to start with pre-trained transformer models of HuggingFace library (HuggingFace 2022) and practice to fine-tune HuggingFace Transformers with TensorFlow and Keras (TensorFlow 2022; Keras 2022) followed by how spaCy v3.0 (spaCy 2022) integrates transformer models as pre-trained pipelines. These techniques and tools will be used in the last workshop for building a Q&A chatbot.

Hence, this workshop will cover the following topics:

- Transformers and Transfer Learning.
- Understanding BERT.
- Transformers and TensorFlow.
- Transformers and spaCy.

15.2 Technical Requirements

Transformers, TensorFlow and spaCy (TensorFlow 2022; spaCy 2022) are to be installed in own PC/notebook computer.

R. S. T. Lee, *Natural Language Processing*

All source codes for this workshop can be downloaded from GitHub archive on NLPWorkshop6 (NLPWorkshop6 2022).

Use pip install commands to install the following packages:

- pip install spacy.
- pip install TensorFlow (note: version 2.2 or above).
- pip install transformers.

15.3 Transformers and Transfer Learning in a Nutshell

Transformers in NLP is an innovative idea which aims to solve sequential modelling tasks and targets problems introduced by Long-Short-Term-Memory (LSTM) architecture (Ekman 2021; Korstanje 2021).

It is a contemporary machine learning concept and architecture introduced by Vaswani et al. (2017) in research paper *Attention Is All You Need*. It explained that "The Transformer is the first transduction model relying entirely on self-attention to compute representations of its input and output without using sequence-aligned RNNs or convolution."

Transduction in this context means transforming input words to output words by transforming input words and sentences into vectors. A transformer is trained on a large corpus such as Wiki or news. These vectors will be used to convey information regarding word semantics, sentence structures, and sentence semantics for downstream tasks.

Word vectors like Glove and FastText are already trained on Wikipedia corpus that can be used in semantic similarity calculations, hence, Transfer Learning means to import knowledge from pre-trained word vectors or pre-trained statistical models.

Transformers offer many pre-trained models to perform NLP tasks such as text classification, text summarization, question answering, machine translation, and natural language generation in over 100 languages. It aims to make state-of-the-art NLP accessible to everyone (Bansal 2021; Rothman 2022; Tunstall et al. 2022; Yıldırım and Asgari-Chenaghlu 2021).

A list of Transformer models provided by HuggingFace (2022) is shown in Fig. 15.1. Each model is named with a combination of architecture names such as BERT or DistilBert, possibly a language code, i.e. en, de, multilingual, which is located at the left side of the figure, and information regarding whether the model is cased or uncased i.e. distinguish between uppercase and lowercase characters.

Task names are also listed at the left-hand side. Each model is labeled with a task name such as text classification or machine translation for Q&A chatbot.

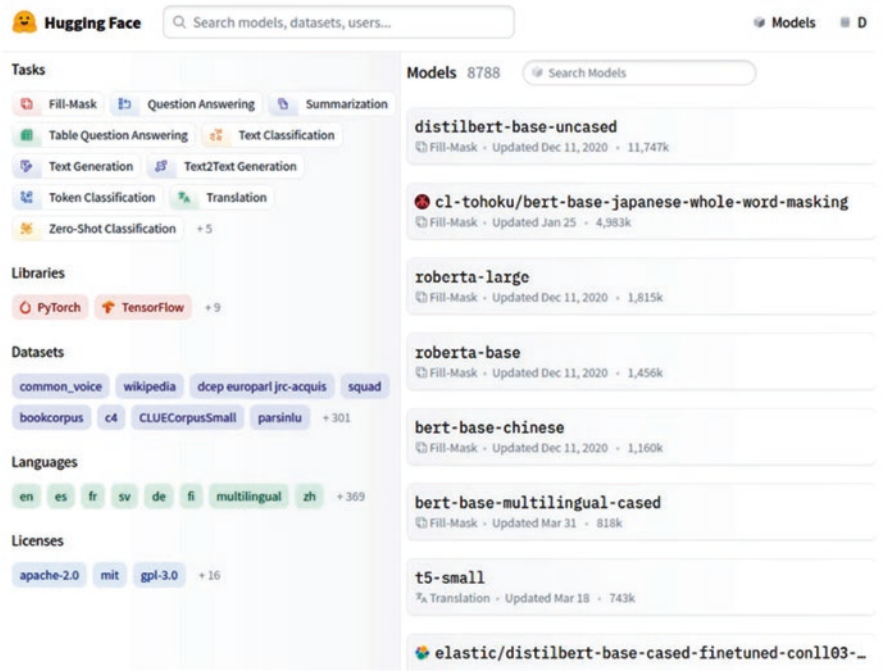

Fig. 15.1 Sample Input Texts and their corresponding Output Class Labels

15.4 Why Transformers?

Let us review text classification with spaCy in LSTM architecture.

LSTMs work for modelling text effectively, but there are shortcomings:

- LSTM architecture has difficulties in learning long texts sometimes. Statistical dependencies in a long text have problems represented by LSTM because it can fail to recall words processed earlier as time steps progress.
- LSTMs are sequential which means that a single word can process at each time step but is impossible to parallelizing learning process causing bottleneck.

Transformers address these problems by not using recurrent layers at all, their architecture is different from LSTM architecture (Bansal 2021; Rothman 2022; Tunstall et al. 2022; Yıldırım and Asgari-Chenaghlu 2021). A Transformer architecture has an input encoder block at the left, called Encoder, and an output decoder at the right called Decoder as shown in Fig. 15.2.

The architecture is catered for a machine translation task, input is a sequence of words from source language, and output is a sequence of words in target language.

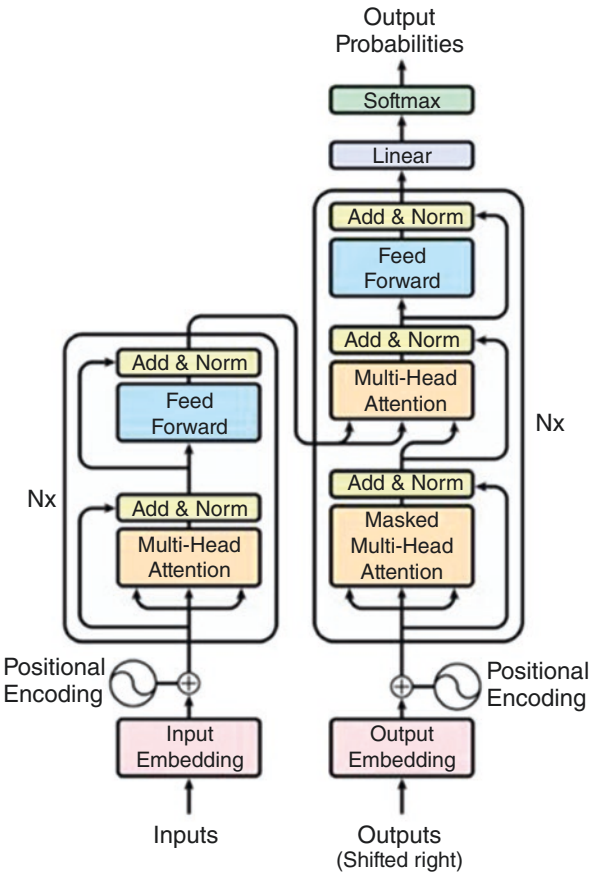

Fig. 15.2 Transformer architecture

Encoder generates a vector representation of input words and passes them to decoder where the word vector transfer is represented by an arrow from encoder block to decoder block direction. Decoder extracts input word vectors, transforms output words into word vectors, and generates the probability of each output word.

There are feedforward layers, which are dense layers in encoder and decoder blocks used for text classification with spaCy. The innovative transformers can place in a Multi-Head Attention block to create a dense representation for each word with self-attention mechanism. This mechanism relates each word in input sentence to other words in the input sentence. Word embedding is calculated by taking a weighted average of other words' embeddings, and each word significance can be calculated in input sentence to enable the architecture focus on each input word sequentially.

A self-attention mechanism of how input words at the left-hand side attend input word *it* at the right-hand side is shown in Fig. 15.3. Dark colors represented relevance, phrase *the animal* are related to *it* than other words in the sentence. This signified transformer can resolve many semantic dependencies in a sentence and

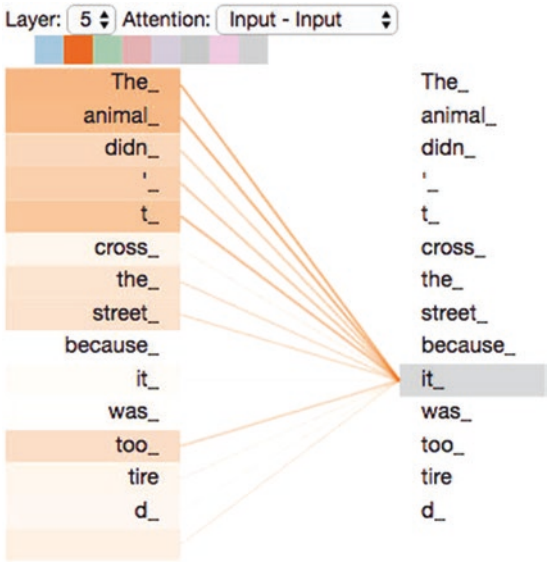

Fig. 15.3 Illustration of the self-attention mechanism

used in different tasks such as text classification and machine translation since they have several architectures depending on tasks. BERT is a popular architecture to be used.

15.5 An Overview of BERT Technology

15.5.1 What Is BERT?

BERT is introduced in a Google's original research paper published by Devlin et al. (2019), the complete Google BERT model can be downloaded from Google's GitHub archive (GoogleBert 2022).

It has the following output features (Bansal 2021; Rothman 2022; Tunstall et al. 2022; Yıldırım and Asgari-Chenaghlu 2021):

- Bidirectional: Each input sentence text data training is processed from left to right and from right to left.
- Encoder: An encoder encodes input sentence.
- Representations: A representation is a word vector.
- Transformers: A transformer-based architecture.

BERT is a trained transformer encoder stack. The input is a sentence, and the output is a sequence of word vectors. Word vectors are contextual which means that a word vector is assigned to a word based on input sentence. In short, BERT outputs contextual word representations as shown in Fig. 15.4.

Fig. 15.4 Word vector for the word "bank"

Fig. 15.5 Two distinct word vectors generated by BERT for the same word *bank* in two different contexts

It is noted that word *bank* has different meanings in these two sentences, word vectors are the same because Glove and FastText are static. Each word has only one vector and vectors are saved to a file after training. Then, these pre-trained vectors can be downloaded to our application. BERT word vectors are dynamic on the contrary. It can generate different word vectors for the same word depending on input sentence. Word vectors generated by BERT are shown in Fig. 15.5 against the counterpart shown in Fig. 15.4:

15.5.2 BERT Architecture

BERT is a transformer encoder stack, which means several encoder layers are stacked on top of each other. The first layer initializes word vectors randomly, and then each encoder layer transforms output of the previous encoder layer. Figure 15.6 illustrates two BERT model sizes: BERT Base and BERT Large.

BERT Base and BERT Large have 12 and 24 encoder layers to generate word vectors sizes of 768 and 1024 comparatively.

BERT outputs word vectors for each input word. A high-level overview of BERT inputs and outputs is illustrated in Fig. 15.7. It showed that BERT input should be in a special format to include special tokens such as CLS.

15.5.3 BERT Input Format

After learning BERT basic architecture, let us look at how to generate output vectors using BERT.

BERT input format can represent a single sentence and a pair of sentences in a single sequence of tokens (for tasks such as question answering and semantic similarity, we input two sentences to the model).

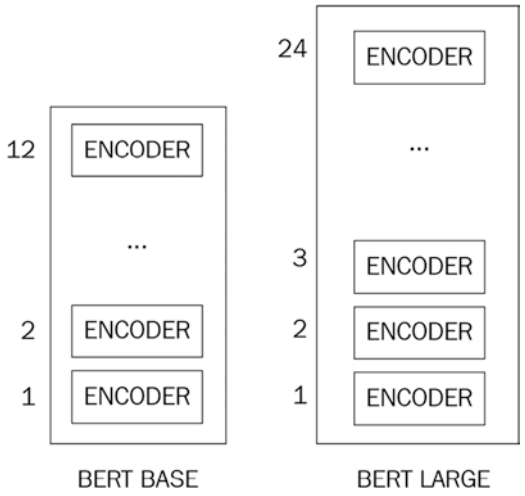

Fig. 15.6 BERT Base and Large architectures (having 12 and 24 encoder layers, respectively)

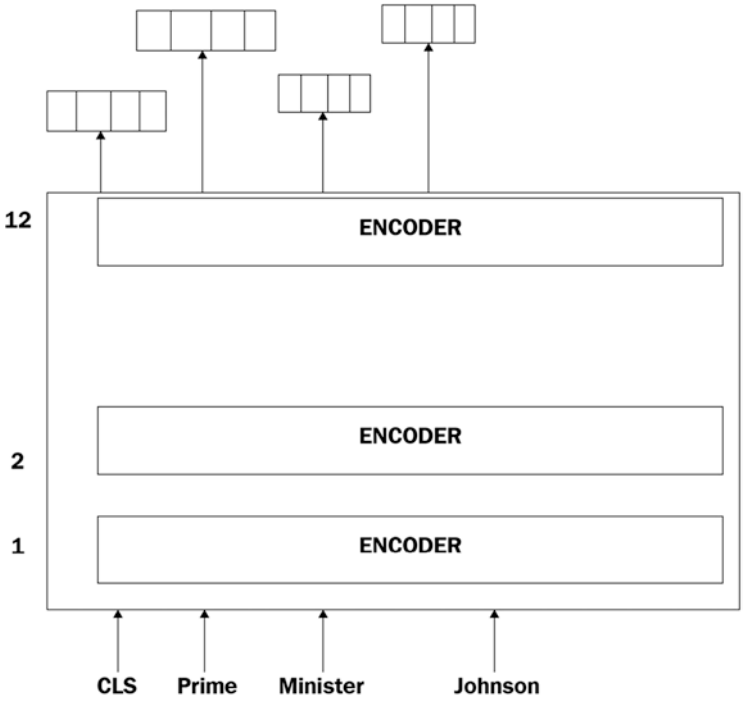

Fig. 15.7 BERT model input word and output word vectors

BERT works with a special tokens class and a special tokenization algorithm called WordPiece.

There are several types of special tokens [CLS], [SEP], and [PAD]:

- [CLS] is the first special token type for every input sequence. This token is a quantity of input sentence for classification tasks but disregard non-classification tasks.
- [SEP] is a sentence separator. If the input is a single sentence, this token will be placed at the end of sentence, i.e. [CLS] sentence [SEP], or to separate two sentences, i.e. [CLS] sentence1 [SEP] sentence2 [SEP].
- [PAD] is a special token for padding. The padding values can generate sentences from dataset with equal length. BERT receives sentences with fixed length only, hence, padding short sentences is required prior feeding to BERT. The tokens maximum length can feed to BERT is 512.

It was learnt that a sentence can feed to Keras model one word at a time, input sentences can be tokenized into words using spaCy tokenizer, but BERT works differently as it uses WordPiece tokenization. A *word piece* is literally a piece of a word.

WordPiece algorithm breaks down words into several subwords, its logic behind is to break down complex/long tokens into tokens, e.g. the word *playing* is tokenized as play + ##ing. A ## character is placed before every word piece to indicate that this token is not a word from language's vocabulary but is a word piece.

Let us look at some examples:

```
playing play, ##ing
played play, ##ed
going go, ##ing
vocabulary = [play, go, ##ing, ##ed]
```

It can concise language vocabulary as WordPiece groups common subwords.

WordPiece tokenization can divide rare/unseen words into their subwords.

After input sentence is tokenized and special tokens are added, each token is converted to its ID and feed token-ID sequences to BERT.

An input sentence transformed into BERT input format is illustrated in Fig. 15.8.

BERT Tokenizer has several methods to perform above tasks but it has an encoding method that combines these steps into a single step.

15.5.4 How to Train BERT?

BERT originators stated that "We then train a large model (12-layers to 24-layers Transformer) on a large corpus (Wikipedia + BookCorpus) for a long time (1 M update steps), and that is BERT" in Google Research's BERT GitHub repository (GoogleBert 2022).

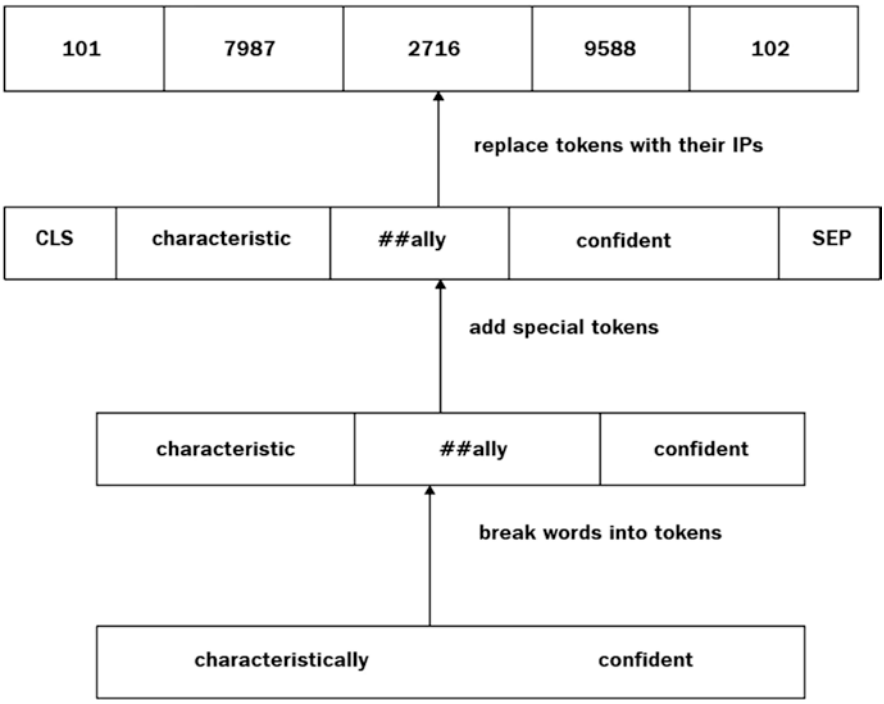

Fig. 15.8 Transforming an input sentence into BERT input format

BERT is trained by masked language model (MLM) and next sentence prediction (NSP).

Language modelling is the task of predicting the next token given the sequence of previous tokens. For example, given the sequence of words *Yesterday I visited*, a language model can predict the next token as one of the tokens *church*, *hospital*, *school*, and so on.

MLM is different. A percentage of tokens are masked randomly to replace a [MASK] token and presume MLM predicts the masked words.

BERT's masked language model is implemented as follows:

1. select 15 input tokens randomly.
2. 80% of selected tokens are replaced by [MASK].
3. 10% of selected tokens are replaced by another token from vocabulary.
4. 10% remain unchanged.

A training sentence to LMM example is as follows:

```
[CLS] Yesterday I [MASK] my friend at [MASK] house [SEP]
```

NSP is the task of predicting the next sentence given by an input sentence. There are two sentences fed to BERT and presume BERT predicts sentences order if second sentence is followed by first sentence.

An input of two sentences separated by a [SEP] token to NSP example is as follows:

```
[CLS] A man robbed a [MASK] yesterday [MASK] 8 o'clock [SEP] He [MASK] the bank with 6
million dollars [SEP]

Label = IsNext
```

It showed that second sentence can follow first sentence, hence, the predicted label is IsNext.

Here is another example:

```
[CLS] Rabbits like to [MASK] carrots and [MASK] leaves [SEP] [MASK] Schwarzenegger is
elected as the governor of [MASK] [SEP]

Label= NotNext
```

This example showed that the pair of sentences generate a *NotNext* label without contextual or semantical relevance.

15.6 Transformers with TensorFlow

Pre-trained transformer models are provided to program developers in open sources by many organizations including Google (GoogleBert 2022), Facebook (Facebook-transformer 2022), and HuggingFace (HuggingFace_transformer 2022).

HuggingFace is an AI company focuses on NLP apportioned to open source.

These pre-trained models and agreeable interfaces can integrate transformers into Python code, as interfaces are compatible with either PyTorch or TensorFlow or both. HuggingFace‘s pre-trained transformers and their TensorFlow interface to transformer models will be used in this workshop.

15.6.1 HuggingFace Transformers

This section will explore HuggingFace‘s pre-trained models, TensorFlow interface and its conventions. HuggingFace offers several models as in Fig. 15.1. Each model is dedicated to a task such as text classification, question answering, and sequence-to-sequence modelling.

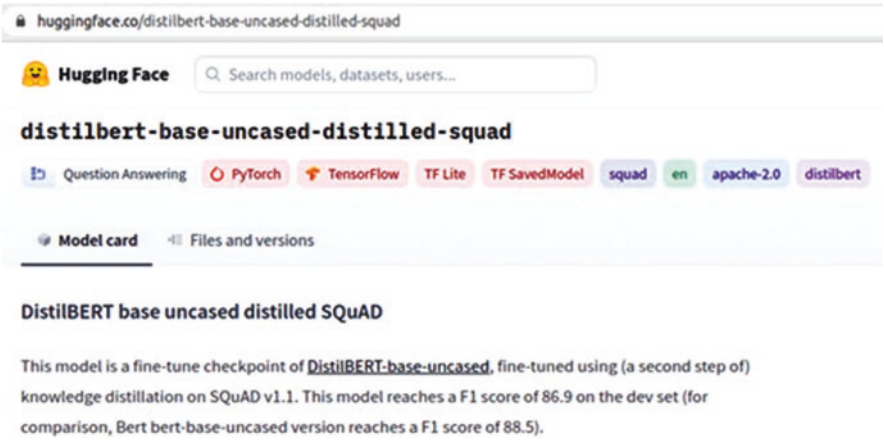

Fig. 15.9 Documentation of the distilbert-base-uncased-distilled-squad model

A HuggingFace documentation of a distilbert-base-uncased-distilled-squad model is shown in Fig. 15.9. A Question-Answering task tag is assigned to the upper-left corner in the documentation followed by supporting deep learning libraries PyTorch, TensorFlow, TFLite, TFSavedModel, training dataset, e.g. squad, model language, e.g. en for English; the license and base model's name, e.g. DistilBERT.

Some models are trained with similar algorithms that belong to identical model family. For example, the DistilBERT family has many models such as distilbert-base-uncased and distilbert-multilingual-cased. Each model name includes information such as casing to distinguish uppercase/lowercase or model language such as en, de, or multilingual.

HuggingFace documentation provides information about each model family with individual model's API in detail. Lists of available models and BERT model architecture variations are shown in Fig. 15.10.

BERT model has many tasks variations such as text classification, question answering, and next sentence prediction.

Each of these models is obtained by placing extra layers atop of BERT output as these outputs are a sequence of word vectors for each word of input sentences.

For example, a BERTForSequenceClassification model is obtained by placing a dense layer atop of BERT word vectors.

15.6.2 Using the BERT Tokenizer

BERT uses WordPiece algorithm for tokenization to ensure that each input word is divided into subwords.

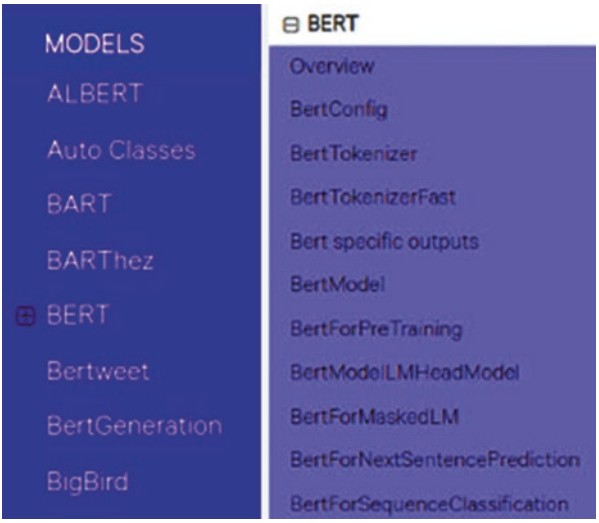

Fig. 15.10 Lists of the available models (left-hand side) and BERT model variations (right-hand side)

Let's look at how to prepare input data with HuggingFace library.

In[1]
```
# Import transformer package
from transformers import BertTokenizer

# Create bert_tokenizer and sample utterance (utt1) and tokens (tok1)
btokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
utt1 = "He lived characteristically idle and romantic."
utt1 = "[CLS] " + utt1 + " [SEP]"
tok1 = btokenizer.tokenize(utt1)
```

In[2]
```
# Display bert tokens
tok1
```

Out[2]
```
['[CLS]', 'he', 'lived', 'characteristic', '##ally', 'idle', 'and', 'romantic', '.', '[SEP]']
```

In[3]
```
# Convert bert tokens to ids (id1)
id1 = btokenizer.convert_tokens_to_ids(tok1)
id1
```

Out[3]
```
[101, 2002, 2973, 8281, 3973, 18373, 1998, 6298, 1012, 102]
```

1. Import BertTokenizer. Note that different models have different tokenizers, e.g. XLNet model's tokenizer is called XLNetTokenizer.
2. Call from_pretrained method on tokenizer object and provide model's name. Needless to download pre-trained bert-base-uncased (or model) as this method downloads model by itself.
3. Call tokenize method. It tokenizes sentence by dividing all words into subwords.
4. Print tokens to examine subwords. The words *he*, *lived*, *idle*, that exist in Tokenizer's vocabulary are to be remained. *Characteristically* is a rare word does not exist in Tokenizer's vocabulary. Thus, tokenizer splits this word into subwords *characteristic* and *##ally*. Notice that *##ally* starts with characters *##* to emphasize that this is a piece of word.
5. Call convert_tokens_to_ids.

Since [CLS] and [SEP] tokens must add to the beginning and end of input sentence, it required to add them manually for the preceding code, but these preprocessing steps can perform in a single step.

BERT provides a method called encode that can:

- add CLS and SEP tokens to input sentence
- tokenize sentence by dividing tokens into subwords
- converts tokens to their token-IDs

Call encode method on input sentence directly as follows:

In[4]

```
from transformers import BertTokenizer

btokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
utt2 = "He lived characteristically idle and romantic."
id2 = btokenizer.encode(utt2)
print(id2)
```

Out[4]

```
[101, 2002, 2973, 8281, 3973, 18373, 1998, 6298, 1012, 102]
```

This code segment outputs token-IDs in a single step instead of step-by-step. The result is a Python list.

Since all input sentences in a dataset must have equal length because BERT cannot process variable-length sentences, padding the longest sentence from dataset into short sentences is required using the parameter "padding='longest'".

Writeup conversion codes are also required if a TensorFlow tensor is used instead of a plain list. HuggingFace library provides *encode_plus* to combine all these steps into single method as follows:

In[5]
```
from transformers import BertTokenizer

btokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
utt3 = "He lived characteristically idle and romantic."

encoded = btokenizer.encode_plus(
        text=utt3,
        add_special_tokens=True,
        padding='longest',
        return_tensors="tf"
)

id3 = encoded["input_ids"]

print(id3)
```

Out[5]
```
tf.Tensor([[ 101 2002 2973 8281 3973 18373 1998 6298 1012  102]],
shape=(1, 10), dtype=int32)
```

Call *encode_plus* to input sentence directly. It is padded to a length of 10 including special tokens [CLS] and [SEP]. The output is a direct TensorFlow tensor with token IDs.

Verify parameter list of *encode_plus()* by:

In[6]
```
btokenizer.encode_plus?
```

BERT Tokenizer provides several methods on input sentences. Data preparation is not straightforward, but practice makes perfect. Try out code examples with own text.

It is ready to process transformed input sentences to BERT model and obtain BERT word vectors.

15.6.3 Word Vectors in BERT

This section will examine BERT model output as they are a sequence of word vectors assigned by one vector per input word. BERT has a special output format. Let' us look at the code first.

In[7]

```
from transformers import BertTokenizer, TFBertModel

btokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
bmodel = TFBertModel.from_pretrained("bert-base-uncased")

utt4 = "He was idle."

encoded = btokenizer.encode_plus(
        text=utt4,
        add_special_tokens=True,
        padding='longest',
        max_length=10,
        return_attention_mask=True,
        return_tensors="tf"
)

id4 = encoded["input_ids"]

outputs = bmodel(id4)
```

- Import TFBertModel
- Initialize out BERT model with a bert-base-uncased pre-trained model
- Transform input sentence to BERT input format with encode_plus, and capture result tf.tensor in the input variable
- Feed sentence to BERT model and capture output with the output's variables

BERT model output is a tuple of two elements. Let us print the shapes of output pair:

In[8]

```
print(outputs[0].shape)
```

Out[8]

```
(1, 6, 768)
```

In[9]

```
print(outputs[1].shape)
```

Out[9]

```
(1, 768)
```

1. Shape, i.e. batch size, sequence length, hidden size is the first element of output. A batch size is the numbers of sentences that can feed to model instantly. When one sentence is fed, the batch size is 1. Sequence length is 10 because sentence is fed max_length = 10 to the tokenizer and padded to length of 10. Hidden_size is a BERT parameter. BERT architecture has 768 hidden layers size to produce word vectors with 768 dimensions. Hence, the first output element contains 768-dimensional vectors per word means it contains 10 words × 768-dimensional vectors
2. The second output is only one vector of 768-dimension. This vector is the word embedding of [CLS] token. Since [CLS] token is an aggregate of the whole sentence, this token embedding is regarded as embeddings pooled version of all words in the sentence. The shape of output tuple is always the batch size, hidden_size. It is to collect [CLS] token's embedding per input sentence basically

When BERT embeddings are extracted, they can be used to train text classification model with TensorFlow and tf.keras.

15.7 Revisit Text Classification Using BERT

Some of the codes will be used from previous workshop, but this time the code is shorter because the embedding and LSTM layers will be replaced by BERT to train a binary text classifier and tf.keras.

This section will use an email log dataset *emails.csv* for spam mail classification found in NLP Workshop6 GitHub repository (NLPWorkshop6 2022).

15.7.1 Data Preparation

Before Text Classification model using BERT is created, let us prepare the data first just like being learnt in the previous workshop:

15.7.1.1 Import Related Modules

In[10]

```
import pandas as pd
import numpy as np
import tensorflow
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.models import Model
```

15.7.1.2 Read emails.csv Datafile

In[11]
```
emails=pd.read_csv("emails.csv",encoding='ISO-8859-1')
emails.head()
```

Out[11]

	text	spam
0	Subject: naturally irresistible your corporate...	1
1	Subject: the stock trading gunslinger fanny i...	1
2	Subject: unbelievable new homes made easy im ...	1
3	Subject: 4 color printing special request add...	1
4	Subject: do not have money , get software cds ...	1

15.7.1.3 Use dropna() to Remove Record with Missing Contents

In[12]
```
emails=emails.dropna()
emails=emails.reset_index(drop=True)
emails.columns = ['text','label']
emails.head()
```

Out[12]

	text	label
0	Subject: naturally irresistible your corporate...	1
1	Subject: the stock trading gunslinger fanny i...	1
2	Subject: unbelievable new homes made easy im ...	1
3	Subject: 4 color printing special request add...	1
4	Subject: do not have money , get software cds ...	1

15.7.2 Start the BERT Model Construction

15.7.2.1 Import BERT Models and Tokenizer

In[13]
```
from transformers import BertTokenizer, TFBertModel, BertConfig, TF-
BertForSequenceClassification
bert_tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
bmodel = TFBertModel.from_pretrained("bert-base-uncased")
```

1. Import BertTokenizer and BERT model, TFBertModel
2. Initialize both tokenizer and BERT model with a pre-trained bert-base-uncased model. Note that model's name starts with TF as names of all HuggingFace pre-trained models for TensorFlow start with TF. Please pay attention to this when using other transformer models

15.7.2.2 Process Input Data with BertTokenizer

In[14]
```
emails.head()
```

Out[14]

	text	label
0	Subject: naturally irresistible your corporate...	1
1	Subject: the stock trading gunslinger fanny i...	1
2	Subject: unbelievable new homes made easy im ...	1
3	Subject: 4 color printing special request add...	1
4	Subject: do not have money , get software cds ...	1

15.7.2.3 Double Check Databank to See Whether Data Has

In[15]
```
messages=emails['text']
labels=emails['label']
len(messages),len(labels)
```

Out[15]
```
(5728, 5728)
```

15.7.2.4 Use BERT Tokenizer

In[16]
```
input_ids=[]
attention_masks=[]

for msg in messages:
    bert_inp=bert_tokenizer.encode_plus(msg,add_special_tokens = True,
max_length =64,pad_to_max_length = True,
return_attention_mask = True)
    input_ids.append(bert_inp['input_ids'])
    attention_masks.append(bert_inp['attention_mask'])

input_ids=np.asarray(input_ids)
attention_masks=np.array(attention_masks)
labels=np.array(labels)
```

This code segment will generate token-IDs for each input sentence of the dataset and append to a list. They are list of class labels consist of 0 and 1 s. convert python lists, input_ids, label to numpy arrays and feed them to Keras model

15.7.2.5 Define Keras Model Using the Following Lines

In[17]

```
inputs = Input(shape=(64,), dtype="int32")
bert = bmodel(inputs)
bert = bert[1]
outputs = Dense(units=1, activation="sigmoid")(bert)
model = Model(inputs, outputs)

adam = tensorflow.keras.optimizers.Adam (learning_rate=2e-5,
epsilon=1e-08)
model.compile(loss="binary_crossentropy", metrics=["accuracy"],
optimizer=adam)
```

15.7.2.6 Perform Model Fitting and Use 1 Epoch to Save Time

In[18]

```
model.fit?
```

In[19]

```
history=model.fit(input_ids,labels,batch_size=1,epochs=1)
```

Out[19]

```
5728/5728 [==============================] - 8675s
2s/step - loss: 0.0950 - accuracy: 0.9663
```

15.7.2.7 Review Model Summary

In[20]

```
bmodel.summary()
```

Out[20]

```
Model: "tf_bert_model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
bert (TFBertMainLayer)       multiple                  109482240

=================================================================
Total params: 109,482,240
Trainable params: 109,482,240
Non-trainable params: 0
_________________________________________________________________
```

A BERT-based text classifier using less than 10 lines of code is to:

1. Define input layer to inputs sentences to model. The shape is 64 because each input sentence has 64 tokens in length. Pad each sentence to 64 tokens when encode_plus method is called
2. Feed input sentences to BERT model
3. Extract second output of BERT output at the third line. Since BERT model's output is a tuple, the first element of output tuple is a sequence of word vectors, and the second element is a single vector that represents the whole sentence called pooled output vector. bert[1] extracts pooled output vector which is a vector of shape (1, 768)
4. Squash pooled output vector to a vector of shape 1 by a sigmoid function which is the class label
5. Define Keras model with inputs and outputs
6. Compile model
7. Fit Keras model

BERT model accepts one line only but can transfer enormous knowledge of Wiki corpus to model. This model obtains an accuracy of 0.96 at the end of the training. A single epoch is usually fitted to the model due to BERT overfits a moderate size corpus.

The rest of the code handles compiling and fitting Keras model as BERT has a huge memory requirement as can be seen by RAM requirements of Google Research's GitHub archive (GoogleBert-Memory 2022).

The training code operates for about an hour in local machine, where bigger datasets require more time even for one epoch.

This section will learn how to train a Keras model with BERT from scratch.

15.8 Transformer Pipeline Technology

HuggingFace Transformers library provide pipelines to assist program developers and benefit from transformer code immediately without custom training. A pipeline is a combination of a tokenizer and a pre-trained model.

HuggingFace provides models for various NLP tasks, its HuggingFace pipelines offer:

- Sentiment analysis (Agarwal 2020; Siahaan and Sianipar 2022)
- Question answering (Rothman 2022; Tunstall et al. 2022)
- Text summarization (Albrecht et al. 2020; Kedia and Rasu 2020)
- Translation (Arumugam and Shanmugamani 2018; Géron 2019)

This section will explore pipelines for sentiment analysis and question answering.

15.8.1 Transformer Pipeline for Sentiment Analysis

Let us start examples on sentiment analysis:

In[21]
```
from transformers import pipeline

nlp = pipeline("sentiment-analysis")

utt5 = "I hate I am being a worker in the desert."
utt6 = "I like you who are beautiful and kind."

result1 = nlp(utt5)
result2 = nlp(utt6)
```

The following steps are taken in the preceding code snippet:

1. Import pipeline function from transformers' library. This function creates pipeline objects with task name given as a parameter. Hence, a sentiment analysis pipeline object nlp is created by calling this function on the second line
2. Define two example sentences with negative and positive sentiments. Then feed these sentences to the pipeline object nlp.

Check outputs:

In[22]
```
result1
```
Out[22]
```
[{'label': 'NEGATIVE', 'score': 0.9276903867721558}]
```

In[23]
```
result2
```
Out[23]
```
[{'label': 'POSITIVE', 'score': 0.9998767375946045}]
```

15.8.2 Transformer Pipeline for QA System

Next, we will perform on question answering. Let us see the code:

In[24]

```
from transformers import pipeline

nlp = pipeline("question-answering")

res = nlp({
   'question': 'What is the name of this book ?',
   'context': "I'll publish my new book Natural Language Processing soon."
})

print(res)
```

Out[24]

```
{'score': 0.9857430458068848, 'start': 25, 'end': 52, 'answer': 'Natural Language Processing'}
```

Again, import pipeline function to create a pipeline object nlp. A context which has identical background information for the model is required for question-answering tasks to the model

- Request the model about this book's name after giving information of *this new publication will be available soon*
- The answer is *natural language processing*, as expected
- Try your own examples as simple exercise

HuggingFace transformers studies are completed. Let us move on to final section to see what spaCy offers on transformers.

Workshop 6.1 revisit sentiment analysis using Transformer technology

1. Use either previous workshop databank or another to import databank for sentiment analysis
2. Try to implement sentiment analysis using previous and Transformer technology learnt in this workshop
3. Compare performances and analysis (bonus)

15.9 Transformer and spaCy

SpaCy v3.0 had released new features and components. It has integrated transformers into spaCy NLP pipeline to introduce one more pipeline component called Transformer. This component allows users to use all HuggingFace models with spaCy pipelines. A spaCy NLP pipeline without transformers is illustrated in Fig. 15.11.

With the release of v3.0, v2 style spaCy models are still supported and transformer-based models introduced. A transformer-based pipeline component looks like the following as illustrated in Fig. 15.12:

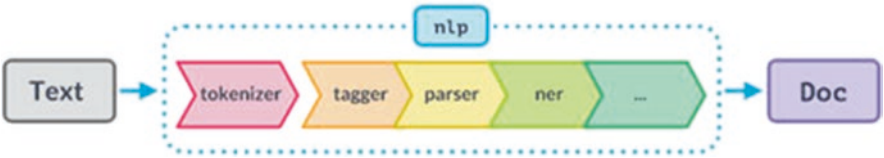

Fig. 15.11 Vector-based spaCy pipeline components

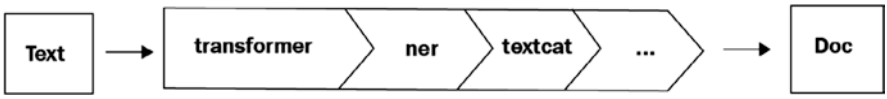

Fig. 15.12 Transformed-based spaCy pipeline components

en_core_web_trf

English transformer pipeline (roberta-base). Components: transformer, tagger, parser, ner, attribute_ruler, lemmatizer.

LANGUAGE	EN English
TYPE	CORE Vocabulary, syntax, entities, vectors
GENRE	WEB written text (blogs, news, comments)
SIZE	TRF 438 MB
COMPONENTS	transformer, tagger, parser, ner, attribute_ruler, lemmatizer
PIPELINE	transformer, tagger, parser, ner, attribute_ruler, lemmatizer
VECTORS	0 keys, 0 unique vectors (0 dimensions)
SOURCES	OntoNotes 5
AUTHOR	Explosion
LICENSE	MIT

Fig. 15.13 spaCy English transformer-based language models

Transformer-based models and v2 style models are listed under Models page of the documentation (spaCy-model 2022) in English model for each supported language. Transformer-based models have various sizes and pipeline components like v2 style models. Also, each model has corpus and genre information like v2 style models. An example of an English transformer-based language model from Models page is shown in Fig. 15.13.

It showed that the first pipeline component is a transformer that generates word representations and deals with WordPiece algorithm to tokenize words into subwords. Word vectors are fed to the rest of the pipeline.

Downloading, loading, and using transformer-based models are identical to v2 style models.

English has two pre-trained transformer-based models, en_core_web_trf and en_core_web_lg currently. Let us start by downloading the en_core_web_trf model:

```
python3 -m spacy download en_core_web_trf
```

Import spaCy module and transformer-based model:

In[25]
```
import spacy
import torch
import spacy_transformers
nlp = spacy.load("en_core_web_trf")
```

After loading model and initializing pipeline, use this model the same way as in v2 style models:

In[26]
```
utt7 = nlp("I visited my friend Betty at her house.")
utt7.ents
```

Out[26]
```
(Betty,)
```

In[27]
```
for word in utt7:
    print(word.pos_, word.lemma_)
```

Out[27]
```
PRON I
VERB visit
PRON my
NOUN friend
PROPN Betty
ADP at
PRON her
NOUN house
PUNCT .
```

These features related to the transformer component can be accessed by *._.trf_data* which contain word pieces, input ids, and vectors generated by the transformer.

Let's examine the features one by one:

In[28]
```
utt8 = nlp("It went there unwillingly.")
```

In[29]

```
utt8._.trf_data.wordpieces
```

Out[29]

```
WordpieceBatch(strings=[['<s>', 'It', 'Ġwent', 'Ġthere', 'Ġunw', 'ill',
'ingly', '.', '</s>']], input_ids=array([[    0,   243,   439,    89, 10
963,  1873,  7790,     4,     2]]), attention_mask=array([[1., 1., 1.,
1., 1., 1., 1., 1., 1.]], dtype=float32), lengths=[9], token_type_ids=No
ne)
```

There are five elements: word pieces, input IDs, attention masks, lengths, and token type IDs in the preceding output.

Word pieces are subwords generated by WordPiece algorithm. The word pieces of this sentence are as follows:

```
<s>
It
Gwent
Gthere
Gunw
Ill
ingly
.
</s>
```

The first and last tokens are special tokens used at the beginning and end of the sentence. The word *unwillingly* is divided into three subwords—*unw*, *ill*, and *ingly*. A G character is used to mark word boundaries. Tokens without G are subwords, such as *ill* and *ingly* in the preceding word piece list, except first word in the sentence marked by <'s'>.

Input IDs have identical meanings which are subword IDs assigned by the transformer's tokenizer.

The attention mask is a list of 0 s and 1 s for pointing the transformer to tokens it should notice. 0 corresponds to PAD tokens, while all other tokens should have a corresponding 1.

Lengths refer to the length of sentence after dividing into subwords. Here is 9 but notice that len(doc) outputs is 5, while spaCy always operates on linguistic words.

token_type_ids are used by transformer tokenizers to mark sentence boundaries of two sentences input tasks such as question and answering. Since there is only one text provided, this feature is inapplicable.

Token vectors are generated by transformer, doc._.trf_data.tensors which contain transformer output, a sequence of word vectors per word, and the pooled output vector. Please refer to Obtaining BERT word vectors section if necessary.

In[30]	utt8._.trf_data.tensors[0].shape
Out[30]	(1, 9, 768)

In[31]	utt8._.trf_data.tensors[1].shape
Out[31]	(1, 768)

The first element of tuple is the vectors for tokens. Each vector is 768-dimensional, hence 9 words produce 9 x 768-dimensional vectors. The second element of tuple is the pooled output vector which is an aggregate representation for input sentence, and the shape is 1 x 768

spaCy provides user-friendly API and packaging for complicated models such as transformers. Transformer integration is a validation of using spaCy for NLP.

References

Agarwal, B. (2020) Deep Learning-Based Approaches for Sentiment Analysis (Algorithms for Intelligent Systems). Springer.

Albrecht, J., Ramachandran, S. and Winkler, C. (2020) Blueprints for Text Analytics Using Python: Machine Learning-Based Solutions for Common Real World (NLP) Applications. O'Reilly Media.

Arumugam, R., & Shanmugamani, R. (2018). Hands-on natural language processing with python. Packt Publishing.

Bansal, A. (2021) Advanced Natural Language Processing with TensorFlow 2: Build effective real-world NLP applications using NER, RNNs, seq2seq models, Transformers, and more. Packt Publishing.

Devlin, J., Chang, M. W., Lee, K. and Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Archive: https://arxiv.org/pdf/1810.04805.pdf.

Ekman, M. (2021) Learning Deep Learning: Theory and Practice of Neural Networks, Computer Vision, Natural Language Processing, and Transformers Using TensorFlow. Addison-Wesley Professional.

Facebook-transformer (2022) Facebook Transformer Model archive. https://github.com/pytorch/fairseq/blob/master/examples/language_model/README.md. Accessed 24 June 2022.

Géron, A. (2019) Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media.

GoogleBert (2022) Google Bert Model Github archive. https://github.com/google-research/bert. Accessed 24 June 2022. Accessed 24 June 2022.

GoogleBert-Memory (2022) GoogleBert Memory Requirement. https://github.com/google-research/bert#out-of-memory-issues.

HuggingFace (2022) Hugging Face official site. https://huggingface.co/. Accessed 24 June 2022.

HuggingFace_transformer (2022) HuggingFace Transformer Model archive. https://github.com/huggingface/transformers. Accessed 24 June 2022.

Kedia, A. and Rasu, M. (2020) Hands-On Python Natural Language Processing: Explore tools and techniques to analyze and process text with a view to building real-world NLP applications. Packt Publishing.

Keras (2022) Keras official site. https://keras.io/. Accessed 24 June 2022.

Korstanje, J. (2021) Advanced Forecasting with Python: With State-of-the-Art-Models Including LSTMs, Facebook's Prophet, and Amazon's DeepAR. Apress.

NLPWorkshop6 (2022) NLP Workshop 6 GitHub archive. https://github.com/raymondshtlee/NLP/tree/main/NLPWorkshop6. Accessed 24 June 2022.

Rothman, D. (2022) Transformers for Natural Language Processing: Build, train, and fine-tune deep neural network architectures for NLP with Python, PyTorch, TensorFlow, BERT, and GPT-3. Packt Publishing.

SpaCy (2022) spaCy official site. https://spacy.io/. Accessed 24 June 2022.

SpaCy-model (2022) spaCy English Pipeline Model. https://spacy.io/models/en. Accessed 24 June 2022.

Siahaan, V. and Sianipar, R. H. (2022) Text Processing and Sentiment Analysis using Machine Learning and Deep Learning with Python GUI. Balige Publishing.

TensorFlow (2022) TensorFlow official site> https://tensorflow.org /. Accessed 24 June 2022.

Tunstall, L, Werra, L. and Wolf, T. (2022) Natural Language Processing with Transformers: Building Language Applications with Hugging Face. O'Reilly Media.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30. https://arxiv.org/abs/1706.03762.

Yıldırım, S, Asgari-Chenaghlu, M. (2021) Mastering Transformers: Build state-of-the-art models from scratch with advanced natural language processing techniques. Packt Publishing.