Chapter 3: Autoencoding Language Models

In the previous chapter, we looked at and studied how a typical Transformer model can be used by HuggingFace's Transformers. So far, all the topics have included how to use pre-defined or pre-built models and less information has been given about specific models and their training.

In this chapter, we will gain knowledge of how we can train autoencoding language models on any given language from scratch. This training will include pre-training and task-specific training of the models. First, we will start with basic knowledge about the BERT model and how it works. Then we will train the language model using a simple and small corpus. Afterward, we will look at how the model can be used inside any Keras model.

For an overview of what will be learned in this chapter, we will discuss the following topics:

- BERT – one of the autoencoding language models
- Autoencoding language model training for any language
- Sharing models with the community
- Understanding other autoencoding models
- Working with tokenization algorithms

Technical requirements

The technical requirements for this chapter are as follows:

- Anaconda

Transformers >= 4.0.0

PyTorch >= 1.0.2

TensorFlow >= 2.4.0

Datasets >= 1.4.1

Tokenizers

Please also check the corresponding GitHub code of `chapter 03`:

https://github.com/PacktPublishing/Advanced-Natural-Language-Processing-with-Transformers/tree/main/CH03.

Check out the following link to see Code in Action Video: https://bit.ly/3i1ycdY

BERT – one of the autoencoding language models

Bidirectional Encoder Representations from Transformers, also known as **BERT**, was one of the first autoencoding language models to utilize the encoder Transformer stack with slight modifications for language modeling.

The BERT architecture is a multilayer Transformer encoder based on the Transformer original implementation. The Transformer model itself was originally for machine translation tasks, but the main improvement made by BERT is the utilization of this part of the architecture to provide better language modeling. This language model, after pretraining, is able to provide a global understanding of the language it is trained on.

BERT language model pretraining tasks

To have a clear understanding of the masked language modeling used by BERT, let's define it with more details. **Masked language modeling** is the task of training a model on input (a sentence with some masked tokens) and obtaining the output as the whole sentence with the masked tokens filled. But how and why does it help a model to obtain better results on downstream tasks such as classification? The answer is simple: if the model can do a cloze test (a linguistic test for evaluating language

understanding by filling in blanks), then it has a general understanding of the language itself. For other tasks, it has been pretrained (by language modeling) and will perform better.

Here's an example of a cloze test:

George Washington was the first President of the ___ States.

It is expected that *United* should fill in the blank. For a masked language model, the same task is applied, and it is required to fill in the masked tokens. However, masked tokens are selected randomly from a sentence.

Another task that BERT is trained on is **Next Sentence Prediction** (**NSP**). This pretraining task ensures that BERT learns not only the relations of all tokens to each other in predicting masked ones but also helps it understand the relation between two sentences. A pair of sentences is selected and given to BERT with a *[SEP]* splitter token in between. It is also known from the dataset whether the second sentence comes after the first one or not.

The following is an example of NSP:

It is required from reader to fill the blank. Bitcoin price is way over too high compared to other altcoins.

In this example, the model is required to predict it as negative (the sentences are not related to each other).

These two pretraining tasks enable BERT to have an understanding of the language itself. BERT token embeddings provide a contextual embedding for each token. **Contextual embedding** means each token has an embedding that is completely related to the surrounding tokens. Unlike Word2Vec and such models, BERT provides better information for each token embedding. NSP tasks, on the other hand, enable BERT to have better embeddings for *[CLS]* tokens. This token, as was discussed in the first chapter, provides information about the whole input. *[CLS]* is used for classification tasks and in the pretraining part learns the overall embedding of the whole input. The following figure shows an overall look at the BERT model. *Figure 3.1* shows the respective input and output of the BERT model:

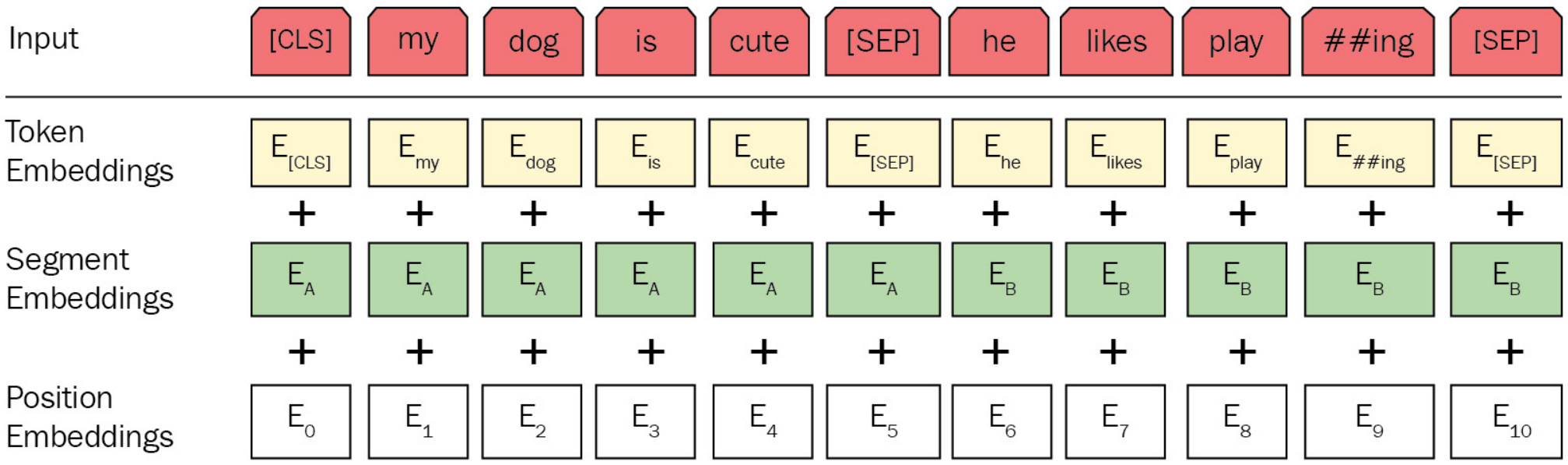

Figure 3.1 – The BERT model

Let's move on to the next section!

A deeper look into the BERT language model

Tokenizers are one of the most important parts of many NLP applications in their respective pipelines. For BERT, WordPiece tokenization is used. Generally, **WordPiece**, **SentencePiece**, and **BytePairEncoding** (**BPE**) are the three most widely known tokenizers, used by different Transformer-based architectures, which are also covered in the next sections. The main reason that BERT or any other Transformer-based architecture uses subword tokenization is the ability of such tokenizers to deal with unknown tokens.

BERT also uses positional encoding to ensure the position of the tokens is given to the model. If you recall from *Chapter 1*, *From Bag-of-Words to the Transformers*, BERT and similar models use non-sequential operations such as dense neural layers. Conventional models such as LSTM- and RNN-based models get the position by the order of the tokens in the sequence. In order to provide this extra information to BERT, positional encoding comes in handy.

Pretraining of BERT such as autoencoding models provides language-wise information for the model, but in practice, when dealing with different problems such as sequence classification, token classification, or question answering, different parts of the model output are used.

For example, in the case of a sequence classification task, such as sentiment analysis or sentence classification, it is proposed by the original BERT article that *[CLS]* embedding from the last layer must be used. However, there is other research that performs classification using different techniques using BERT (using average token embedding from all tokens, deploying an LSTM over the last layer, or even using a CNN on top of the last layer). The last *[CLS]* embedding for sequence classification can be used by any classifier, but the proposed, and the most common one, is a

dense layer with an input size equal to the final token embedding size and an output size equal to the number of classes with a softmax activation function. Using sigmoid is also another alternative when the output could be multilabel and the problem itself is a multilabel classification problem.

To give you more detailed information about how BERT actually works, the following illustration shows an example of an NSP task. Note that the tokenization is simplified here for better understanding:

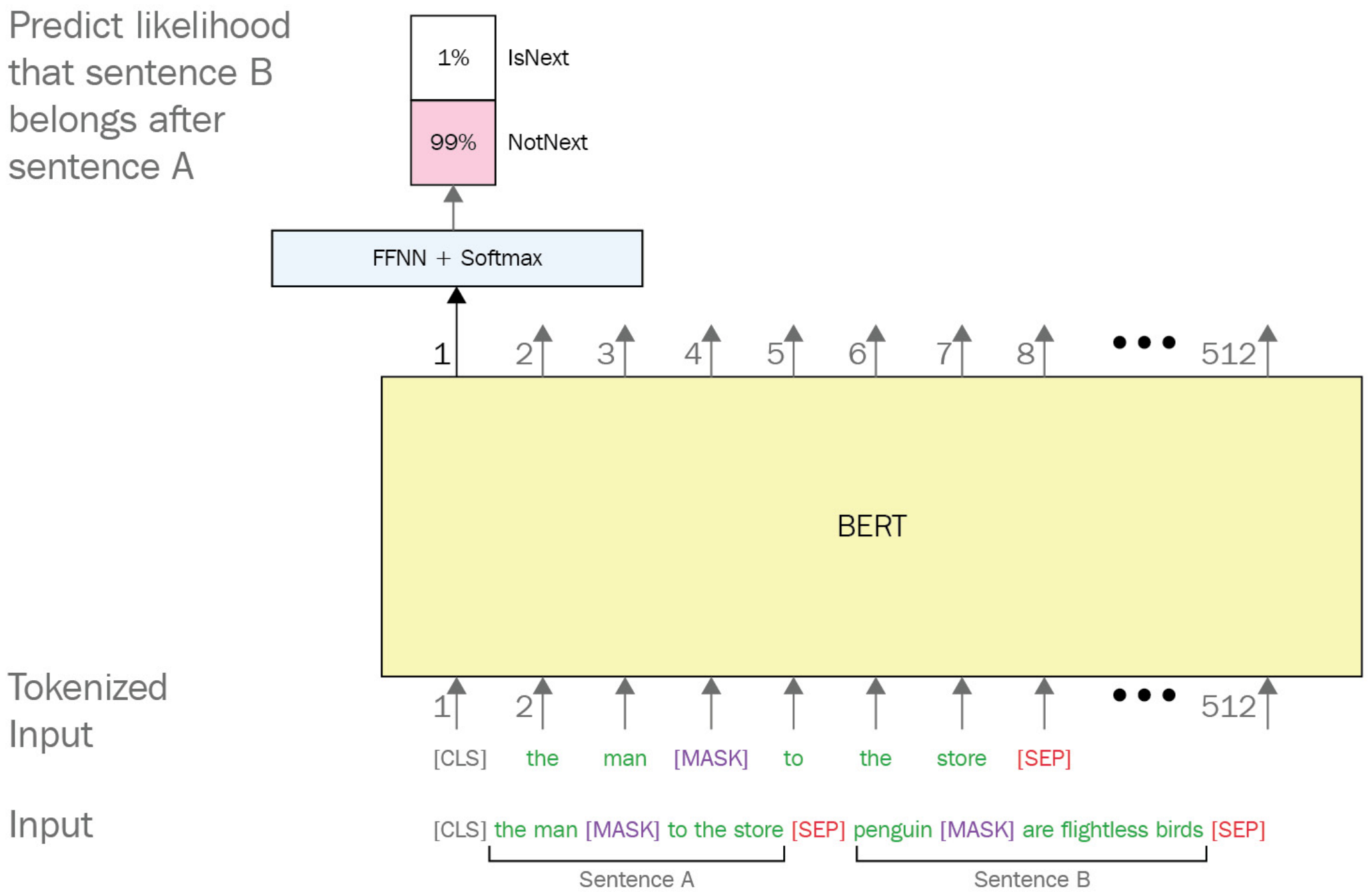

Figure 3.2 – BERT example for an NSP task

The BERT model has different variations, with different settings. For example, the size of the input is variable. In the preceding example, it is set to *512* and the maximum sequence size that model can get as input is *512*. However, this size includes special tokens, *[CLS]* and *[SEP]*, so it will be reduced to *510*. On the other hand, using WordPiece as a tokenizer yields subword tokens, and the sequence size before we can have fewer words, and after tokenization, the size will increase because the tokenizer breaks words into subwords if they are not commonly seen in the pretrained corpus.

The following figure shows an illustration of BERT for different tasks. For an NER task, the output of each token is used instead of *[CLS]*. In the case of question answering, the question and the answer are concatenated using the *[SEP]* delimiter token and

the answer is annotated using *Start/End* and the *Span* output from the last layer. In this case, the *Paragraph* is the context that the *Question* is asked about it:

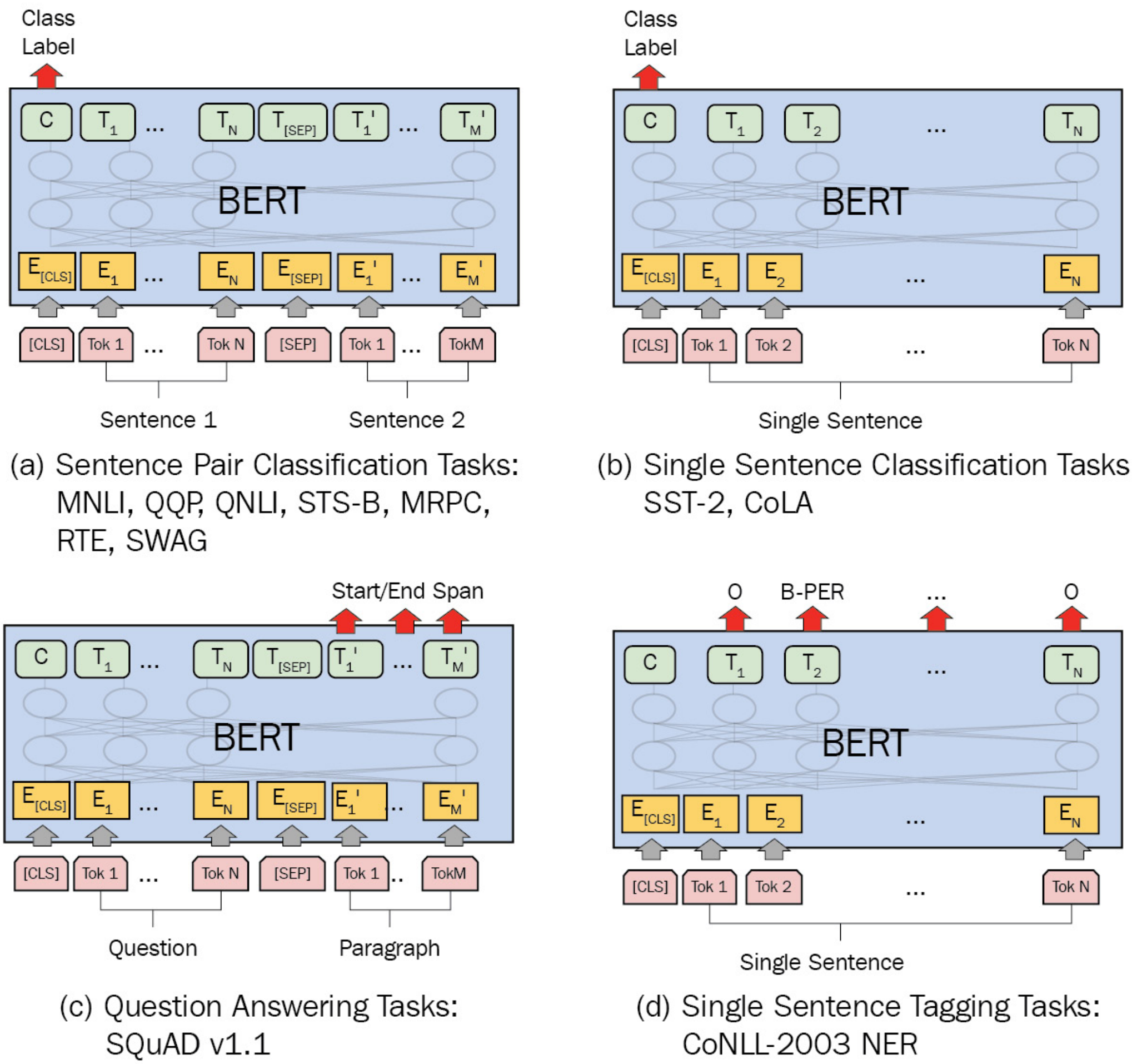

Figure 3.3 – BERT model for various NLP tasks

Regardless of all of these tasks, the most important ability of BERT is its contextual representation of text. The reason it is successful in various tasks is because of the Transformer encoder architecture that represents input in the form of dense vectors. These vectors can be easily transformed into output by very simple classifiers.

Up to this point, you have learned about BERT and how it works. You have learned detailed information on various tasks that BERT can be used for and the important points of this architecture.

In the next section, you will learn how you can pre-train BERT and use it after training.

Autoencoding language model training for any language

We have discussed how BERT works and that it is possible to use the pretrained version of it provided by the HuggingFace repository. In this section, you will learn how to use the HuggingFace library to train your own BERT.

Before we start, it is essential to have good training data, which will be used for the language modeling. This data is called the **corpus**, which is normally a huge pile of data (sometimes it is preprocessed and cleaned). This unlabeled corpus must be appropriate for the use case you wish to have your language model trained on; for example, if you are trying to have a special BERT for, let's say, the English language. Although there are tons of huge, good datasets, such as Common Crawl (https://commoncrawl.org/), we would prefer a small one for faster training.

The IMDB dataset of 50K movie reviews (available at https://www.kaggle.com/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews) is a large dataset for sentiment analysis, but small if you use it as a corpus for training your language model:

1. You can easily download and save it in `.txt` format for language model and tokenizer training by using the following code:

Copy Explain

```
import pandas as pd
imdb_df = pd.read_csv("IMDB Dataset.csv")
reviews = imdb_df.review.to_string(index=None)
with open("corpus.txt", "w") as f:
      f.writelines(reviews)
```

2. After preparing the corpus, the tokenizer must be trained. The `tokenizers` library provides fast and easy training for the WordPiece tokenizer. In order to train it on your corpus, it is required to run the following code:

```
>>> from tokenizers import BertWordPieceTokenizer
>>> bert_wordpiece_tokenizer =BertWordPieceTokenizer()
>>> bert_wordpiece_tokenizer.train("corpus.txt")
```

3. This will train the tokenizer. You can access the trained vocabulary by using the `get_vocab()` function of the trained `tokenizer` object. You can get the vocabulary by using the following code:

```
>>> bert_wordpiece_tokenizer.get_vocab()
```

The following is the output:

```
{'almod': 9111, 'events': 3710, 'bogart': 7647, 'slapstick': 9541, 'terrorist':
16811, 'patter': 9269, '183': 16482, '##cul': 14292, 'sophie': 13109, 'thinki':
10265, 'tarnish': 16310, '##outh': 14729, 'peckinpah': 17156, 'gw': 6157,
'##cat': 14290, '##eing': 14256, 'successfully': 12747, 'roomm': 7363,
'stalwart': 13347,...}
```

4. It is essential to save the tokenizer to be used afterwards. Using the `save_model()` function of the object and providing the directory will save the tokenizer vocabulary for further usage:

```
>>> bert_wordpiece_tokenizer.save_model("tokenizer")
```

5. And you can reload it by using the `from_file()` function:

```
>>> tokenizer = \ BertWordPieceTokenizer.from_file("tokenizer/vocab.txt")
```

6. You can use the tokenizer by following this example:

```
>>> tokenized_sentence = \
tokenizer.encode("Oh it works just fine")
>>> tokenized_sentence.tokens
['[CLS]', 'oh', 'it', 'works', 'just', 'fine','[SEP]']
```

The special tokens `[CLS]` and `[SEP]` will be automatically added to the list of tokens because BERT needs them for processing input.

7. Let's try another sentence using our tokenizer:

```
>>> tokenized_sentence = \
tokenizer.encode("ohoh i thougt it might be workingg well")
['[CLS]', 'oh', '##o', '##h', 'i', 'thoug', '##t', 'it', 'might', 'be',
'working', '##g', 'well', '[SEP]']
```

8. Seems like a good tokenizer for noisy and misspelled text. Now that you have your tokenizer ready and saved, you can train your own BERT. The first step is to use `BertTokenizerFast` from the `Transformers` library. You are required to load the trained tokenizer from the previous step by using the following command:

```
>>> from Transformers import BertTokenizerFast
>>> tokenizer = \ BertTokenizerFast.from_pretrained("tokenizer")
```

We have used `BertTokenizerFast` because it is suggested by the HuggingFace documentation. There is also `BertTokenizer`, which, according to the definition from the library documentation, is not implemented as fast as the fast version. In most of the pretrained models' documentations and cards, it is highly recommended to use the `BertTokenizerFast` version.

9. The next step is preparing the corpus for faster training by using the following command:

```
>>> from Transformers import LineByLineTextDataset
>>> dataset = \
LineByLineTextDataset(tokenizer=tokenizer,
                      file_path="corpus.txt",
                      block_size=128)
```

10. And it is required to provide a data collator for masked language modeling:

```
>>> from Transformers import DataCollatorForLanguageModeling
>>> data_collator = DataCollatorForLanguageModeling(
                      tokenizer=tokenizer,
                      mlm=True,
                      mlm_probability=0.15)
```

The data collator gets the data and prepares it for the training. For example, the data collator above takes data and prepares it for masked language modeling with a probability of `0.15`. The purpose of using such a mechanism is to do the preprocessing on the fly, which makes it possible to use fewer resources. On the other hand, it slows down the training process because each sample has to be preprocessed on the fly at training time.

11. The training arguments also provide information for the trainer in the training phase, which can be set by using the following command:

```
>>> from Transformers import TrainingArguments
>>> training_args = TrainingArguments(
                    output_dir="BERT",
                    overwrite_output_dir=True,
                    num_train_epochs=1,
                    per_device_train_batch_size=128)
```

12. We'll now make the BERT model itself, which we are going to use with the default configuration (the number of attention heads, Transformer encoder layers, and so on):

```
>>> from Transformers import BertConfig, BertForMaskedLM
>>> bert = BertForMaskedLM(BertConfig())
```

13. And the final step is to make a trainer object:

```
>>> from Transformers import Trainer
>>> trainer = Trainer(model=bert,
                      args=training_args,
                      data_collator=data_collator,
                      train_dataset=dataset)
```

14. Finally, you can train your language model using the following command:

```
>>> trainer.train()
```

It will show you a progress bar indicating the progress made in training:

Figure 3.4 – BERT model training progress

During the model training, a log directory called `runs` will be used to store the checkpoint in steps:

Figure 3.5 – BERT model checkpoints

15. After the training is finished, you can easily save the model using the following command:

```
>>> trainer.save_model("MyBERT")
```

Up to this point, you have learned how you can train BERT from scratch for any specific language that you desire. You've learned how to train the tokenizer and BERT model using the corpus you have prepared.

16. The default configuration that you provided for BERT is the most essential part of this training process, which defines the architecture of BERT and its hyperparameters. You can take a peek at these parameters by using the following code:

```
>>> from Transformers import BertConfig
>>> BertConfig()
```

The following is the output:


```
BertConfig {
  "attention_probs_dropout_prob": 0.1,
  "gradient_checkpointing": false,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "pad_token_id": 0,
  "position_embedding_type": "absolute",
  "transformers_version": "4.4.2",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 30522
}
```


Figure 3.6 – BERT model configuration

If you wish to replicate **Tiny**, **Mini**, **Small**, **Base**, and relative models from the original BERT configurations (https://github.com/google-research/bert), you can change these settings:

	H=128	H=256	H=512	H=768
L=2	2/128 (BERT-Tiny)	2/256	2/512	2/768
L=4	4/128	4/256 (BERT-Mini)	4/512 (BERT-Small)	4/768
L=6	6/128	6/256	6/512	6/768
L=8	8/128	8/256	8/512 (BERT-Medium)	8/768
L=10	10/128	10/256	10/512	10/768
L=12	12/128	12/256	12/512	12/768 (BERT-Base)

Figure 3.7 – BERT model configurations (https://github.com/google-research/bert)

Note that changing these parameters, especially `max_position_embedding`, `num_attention_heads`, `num_hidden_layers`, `intermediate_size`, and `hidden_size`, directly affects the training time. Increasing them dramatically increases the training time for a large corpus.

17. For example, you can easily make a new configuration for a tiny version of BERT for faster training using the following code:

```
>>> tiny_bert_config = \ BertConfig(max_position_embeddings=512, hidden_size=128,
            num_attention_heads=2,
            num_hidden_layers=2,
            intermediate_size=512)
>>> tiny_bert_config
```

The following is the result of the code:


```
BertConfig {
  "attention_probs_dropout_prob": 0.1,
  "gradient_checkpointing": false,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 128,
  "initializer_range": 0.02,
  "intermediate_size": 512,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 2,
  "num_hidden_layers": 2,
  "pad_token_id": 0,
  "position_embedding_type": "absolute",
  "transformers_version": "4.4.2",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 30522
}
```


Figure 3.8 – Tiny BERT model configuration

18. By using the same method, we can make a tiny BERT model using this configuration:

```
>>> tiny_bert = BertForMaskedLM(tiny_bert_config)
```

19. And using the same parameters for training, you can train this tiny new BERT:

```
>>> trainer = Trainer(model=tiny_bert, args=training_args,
                      data_collator=data_collator,
                      train_dataset=dataset)
>>> trainer.train()
```

The following is the output:

Figure 3.9 – Tiny BERT model configuration

It is clear that the training time is dramatically decreased, but you should be aware that this is a tiny version of BERT with fewer layers and parameters, which is not as good as BERT Base.

Up to this point, you have learned how to train your own model from scratch, but it is essential to note that using the `datasets` library is a better choice when dealing with datasets for training language models or leveraging it to perform task-specific training.

20. The BERT language model can also be used as an embedding layer combined with any deep learning model. For example, you can load any pretrained BERT model or your own version that has been trained in the previous step. The following code shows how you must load it to be used in a Keras model:

```
>>>  from Transformers import\
TFBertModel, BertTokenizerFast
>>>  bert = TFBertModel.from_pretrained(
"bert-base-uncased")
>>> tokenizer = BertTokenizerFast.from_pretrained(
"bert-base-uncased")
```

21. But you do not need the whole model; instead, you can access the layers by using the following code:

```
>>> bert.layers
[<Transformers.models.bert.modeling_tf_bert.TFBertMainLayer at 0x7f72459b1110>]
```

22. As you can see, there is just a single layer from `TFBertMainLayer`, which you can access within your Keras model. But before using it, it is nice to test it and see what kind of output it provides:

```
>>> tokenized_text = tokenizer.batch_encode_plus(
                    ["hello how is it going with you",
                    "lets test it"],
                     return_tensors="tf",
                     max_length=256,
                     truncation=True,
                     pad_to_max_length=True)
>>> bert(tokenized_text)
```

The output is as follows:

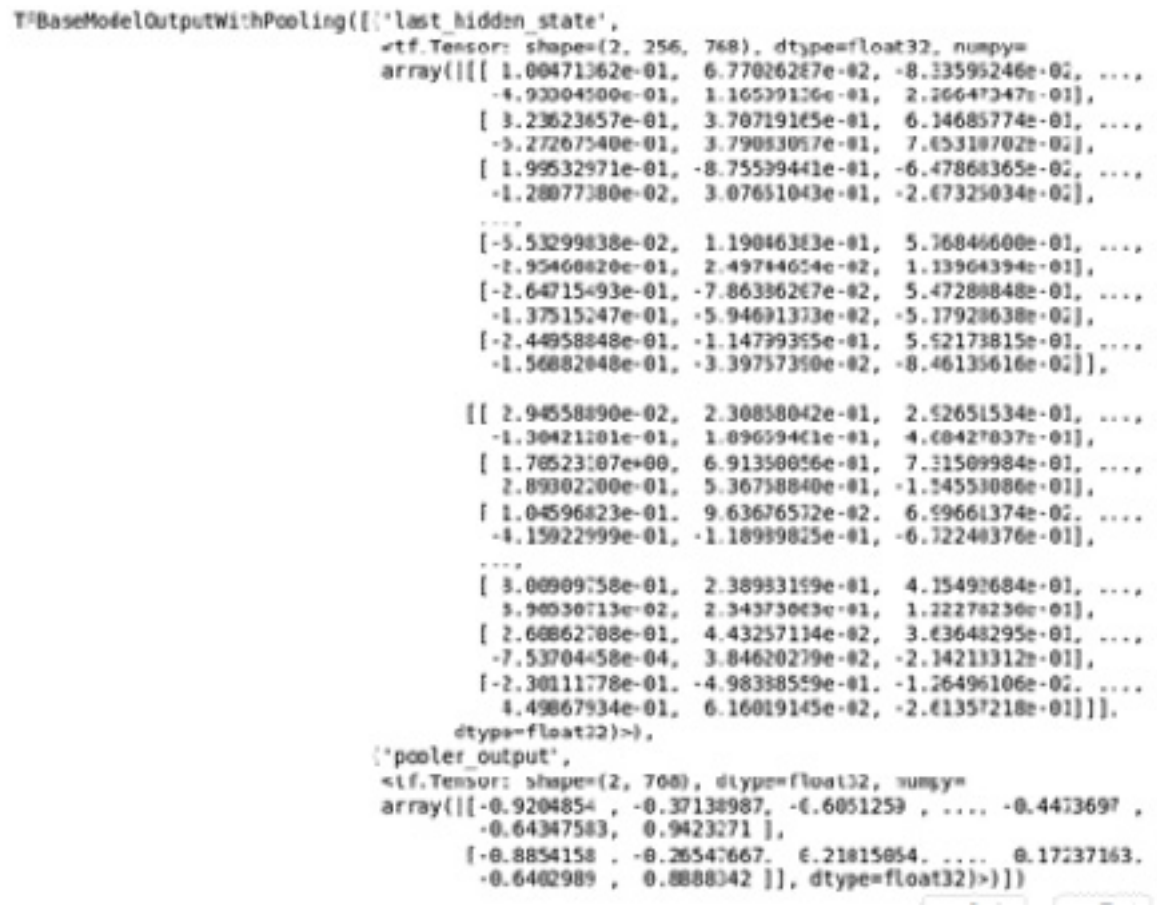

Figure 3.10 – BERT model output

As can be seen from the result, there are two outputs: one for the last hidden state and one for the pooler output. The last hidden state provides all token embeddings from BERT with additional *[CLS]* and *[SEP]* tokens at the start and end, respectively.

23. Now that you have learned more about the TensorFlow version of BERT, you can make a Keras model using this new embedding:

```
from tensorflow import keras
import tensorflow as tf
max_length = 256
tokens = keras.layers.Input(shape=(max_length,),
                            dtype=tf.dtypes.int32)
masks = keras.layers.Input(shape=(max_length,),
                           dtype=tf.dtypes.int32)
embedding_layer = bert.layers[0]([tokens,masks])[0][:,0,:]
dense = tf.keras.layers.Dense(units=2,
        activation="softmax")(embedding_layer)
model = keras.Model([tokens,masks],dense)
```

24. The model object, which is a Keras model, has two inputs: one for tokens and one for masks. Tokens has `token_ids` from the tokenizer output and the masks will have `attention_mask`. Let's try it and see what happens:

```
>>> tokenized = tokenizer.batch_encode_plus(
["hello how is it going with you",
"hello how is it going with you"],
return_tensors="tf",
max_length= max_length,
truncation=True,
pad_to_max_length=True)
```

25. It is important to use `max_length`, `truncation`, and `pad_to_max_length` when using `tokenizer`. These parameters make sure you have the output in a usable shape by padding it to the maximum length of 256 that was defined before. Now you can run the model using this sample:

```
>>>model([tokenized["input_ids"],tokenized["attention_mask"]])
```

The following is the output:

```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[0.45928752, 0.5407125 ],
       [0.45928752, 0.5407125 ]], dtype=float32)>
```

Figure 3.11 – BERT model classification output

26. When training the model, you need to compile it using the `compile` function:

```
>>> model.compile(optimizer="Adam",
loss="categorical_crossentropy",
metrics=["accuracy"])
>>> model.summary()
```

The output is as follows:

```
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
input_tokens (InputLayer)    [(None, 256)]             0
_________________________________________________________________________________________________
input_masks (InputLayer)     [(None, 256)]             0
_________________________________________________________________________________________________
bert (TFBertMainLayer)       multiple                  109482240   input_tokens[0][0]
                                                                   input_masks[0][0]
_________________________________________________________________________________________________
tf.__operators__.getitem_3 (Sli (None, 768)            0           bert[3][0]
_________________________________________________________________________________________________
output_layer (Dense)         (None, 2)                 1538        tf.__operators__.getitem_3[0][0]
=================================================================================================
Total params: 109,483,778
Trainable params: 109,483,778
Non-trainable params: 0
_________________________________________________________________________________________________
```

Figure 3.12 – BERT model summary

27. From the model summary, you can see that the model has 109,483,778 trainable parameters including BERT. But if you have your BERT model pretrained and you want to freeze it in a task-specific training, you can do so with the following command:

```
>>> model.layers[2].trainable = False
```

As far as we know, the layer index of the embedding layer is 2, so we can simply freeze it. If you rerun the summary function, you will see the trainable parameters are reduced to 1,538, which is the number of parameters of the last layer:

```
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
input_tokens (InputLayer)    [(None, 256)]             0
_________________________________________________________________________________________________
input_masks (InputLayer)     [(None, 256)]             0
_________________________________________________________________________________________________
bert (TFBertMainLayer)       multiple                  109482240   input_tokens[0][0]
                                                                   input_masks[0][0]
_________________________________________________________________________________________________
tf.__operators__.getitem_3 (Sli (None, 768)            0           bert[3][0]
_________________________________________________________________________________________________
output_layer (Dense)         (None, 2)                 1538        tf.__operators__.getitem_3[0][0]
=================================================================================================
Total params: 109,483,778
Trainable params: 1,538
Non-trainable params: 109,482,240
_________________________________________________________________________________________________
```

Figure 3.13 – BERT model summary with fewer trainable parameters

28. As you recall, we used the IMDB sentiment analysis dataset for training the language model. Now you can use it for training the Keras-based model for

sentiment analysis. But first, you need to prepare the input and output:

```
import pandas as pd
imdb_df = pd.read_csv("IMDB Dataset.csv")
reviews = list(imdb_df.review)
tokenized_reviews = \
tokenizer.batch_encode_plus(reviews, return_tensors="tf",
                            max_length=max_length,
                            truncation=True,
                            pad_to_max_length=True)
import numpy as np
train_split = int(0.8 * \ len(tokenized_reviews["attention_mask"]))
train_tokens = tokenized_reviews["input_ids"]\
[:train_split]
test_tokens = tokenized_reviews["input_ids"][train_split:]
train_masks = tokenized_reviews["attention_mask"]\
[:train_split]
test_masks = tokenized_reviews["attention_mask"]\
[train_split:]
sentiments = list(imdb_df.sentiment)
labels = np.array([[0,1] if sentiment == "positive" else\
[1,0] for sentiment in sentiments])
train_labels = labels[:train_split]
test_labels = labels[train_split:]
```

29. And finally, your data is ready, and you can fit your model:

```
>>> model.fit([train_tokens,train_masks],train_labels,
              epochs=5)
```

And after fitting the model, your model is ready to be used. Up to this point, you have learned how to perform model training for a classification task. You have learned how to save it, and in the next section, you will learn how it is possible to share the trained model with the community.

Sharing models with the community

HuggingFace provides a very easy-to-use model-sharing mechanism:

1. You can simply use the following `cli` tool to log in:

```
Transformers-cli login
```

2. After you've logged in using your own credentials, you can create a repository:

```
Transformers-cli repo create a-fancy-model-name
```

3. You can put any model name for the `a-fancy-model-name` parameter and then it is essential to make sure you have git-lfs:

```
git lfs install
```

   Git LFS is a Git extension used for handling large files. HuggingFace pretrained models are usually large files that require extra libraries such as LFS to be handled by Git.

4. Then you can clone the repository you have just created:

```
git clone https://huggingface.co/username/a-fancy-model-name
```

5. Afterward, you can add and remove from the repository as you like, and then, just like Git usage, you have to run the following command:

```
git add . && git commit -m "Update from $USER"
git push
```

Autoencoding models rely on the left encoder side of the original Transformer and are highly efficient at solving classification problems. Even though BERT is a typical example of autoencoding models, there are many alternatives discussed in the literature. Let's take a look at these important alternatives.

Understanding other autoencoding models

In this part, we will review autoencoding model alternatives that slightly modify the original BERT. These alternative re-implementations have led to better downstream tasks by exploiting many sources: optimizing the pre-training process and the number of layers or heads, improving data quality, designing better objective functions, and so forth. The source of improvements roughly falls into two parts: *better architectural design choice* and *pre-training control*.

Many effective alternatives have been shared lately, so it is impossible to understand and explain them all here. We can take a look at some of the most cited models in the literature and the most used ones on NLP benchmarks. Let's start with **Albert** as a re-implementation of BERT that focuses especially on architectural design choice.

Introducing ALBERT

The performance of language models is considered to improve as their size gets bigger. However, training such models is getting more challenging due to both memory limitations and longer training times. To address these issues, the Google team proposed the **Albert model** (**A Lite BERT** for Self-Supervised Learning of Language Representations), which is indeed a reimplementation of the BERT architecture by utilizing several new techniques that reduce memory consumption and increase the training speed. The new design led to the language models scaling much better than the original BERT. Along with 18 times fewer parameters, Albert trains 1.7 times faster than the original BERT-large model.

The Albert model mainly consists of three modifications of the original BERT:

Factorized embedding parameterization

Cross-layer parameter sharing

Inter-sentence coherence loss

The first two modifications are parameter-reduction methods that are related to the issue of model size and memory consumption in the original BERT. The third corresponds to a new objective function: **Sentence-Order Prediction** (**SOP**), replacing the **Next Sentence Prediction** (**NSP**) task of the original BERT, which led to a much thinner model and improved performance.

Factorized embedding parameterization is used to decompose the large vocabulary-embedding matrix into two small matrices, which separate the size of the hidden layers from the size of the vocabulary. This decomposition reduces the embedding

parameters from *O(V × H) to O(V × E + E × H)* where *V* is *Vocabulary*, *H* is *Hidden Layer Size*, *E* is *Embedings*, which leads to more efficient usage of the total model parameters *if H >> E* is satisfied.

Cross-layer parameter sharing prevents the total number of parameters from increasing as the network gets deeper. The technique is considered another way to improve parameter efficiency since we can keep the parameter size smaller by sharing or copying. In the original paper, they experimented with many ways to share parameters, such as either sharing FF-only parameters across layers or sharing attention-only parameters or entire parameters.

The other modification of Albert is inter-sentence coherence loss. As we already discussed, the BERT architecture takes advantage of two loss calculations, the **Masked Language Modeling** (**MLM**) loss and NSP. NSP comes with binary cross-entropy loss for predicting whether or not two segments appear in a row in the original text. The negative examples are obtained by selecting two segments from different documents. However, the Albert team criticized NSP for being a topic detection problem, which is considered a relatively easy problem. Therefore, the team proposed a loss based primarily on coherence rather than topic prediction. They utilized SOP loss, which focuses on modeling inter-sentence coherence instead of topic prediction. SOP loss uses the same positive examples technique as BERT, (which is two consecutive segments from the same document), and as negative examples, the same two consecutive segments but with their order swapped. The model is then forced to learn finer-grained distinctions between coherence properties at the discourse level.

1. Let's compare the original BERT and Albert configuration with the `Transformers` library. The following piece of code shows how to configure a BERT-Base initial model. As you see in the output, the number of parameters is around 110 M:

```
#BERT-BASE (L=12, H=768, A=12, Total Parameters=110M)
>> from Transformers import BertConfig, BertModel
>> bert_base= BertConfig()
>> model = BertModel(bert_base)
>> print(f"{model.num_parameters() /(10**6)}\
 million parameters")
109.48224 million parameters
```

2. And the following piece of code shows how to define the Albert model with two classes, `AlbertConfig` and `AlbertModel`, from the `Transformers` library:

```
# Albert-base Configuration
>>> from Transformers import AlbertConfig, AlbertModel
>>> albert_base = AlbertConfig(hidden_size=768,
                               num_attention_heads=12,
                               intermediate_size=3072,)
>>> model = AlbertModel(albert_base)
>>> print(f"{model.num_parameters() /(10**6)}\
million parameters")
11.683584 million parameters
```

Due to that, the default Albert configuration points to Albert-xxlarge. We need to set the hidden size, the number of attention heads, and the intermediate size to fit Albert-base. And the code shows the Albert-base mode as 11M, 10 times smaller than the BERT-base model. The original paper on ALBERT reported benchmarking as in the following table:

Model		Parameters	SQuAD1.1	SQuAD2.0	MNLI	SST-2	RACE	Avg	Speedup
BERT	base	108M	90.4/83.2	80.4/77.6	84.5	92.8	68.2	82.3	4.7x
	large	334M	92.2/85.5	85.0/82.2	86.6	93.0	73.9	85.2	1.0
ALBERT	base	12M	89.3/82.3	80.0/77.1	81.6	90.3	64.0	80.1	5.6x
	large	18M	90.6/83.9	82.3/79.4	83.5	91.7	68.5	82.4	1.7x
	xlarge	60M	92.5/86.1	86.1/83.1	86.4	92.4	74.8	85.5	0.6x
	xxlarge	235M	**94.1/88.3**	**88.1/85.1**	**88.0**	**95.2**	**82.3**	**88.7**	0.3x

Figure 3.14 – Albert model benchmarking

3. From this point on, in order to train an Albert language model from scratch, we need to go through similar phases to those we already illustrated in BERT training in the previous sections by using the uniform Transformers API. There's no need to explain the same steps here! Instead, let's load an already trained Albert language model as follows:

```
from Transformers import AlbertTokenizer, AlbertModel
tokenizer = \
AlbertTokenizer.from_pretrained("albert-base-v2")
model = AlbertModel.from_pretrained("albert-base-v2")
text = "The cat is so sad ."
encoded_input = tokenizer(text, return_tensors='pt')
output = model(**encoded_input)
```

4. The preceding pieces of code download the Albert model weights and its configuration from the HuggingFace hub or from our local cache directory if already cached, which means you've already called the `AlbertTokenizer.from_pretrained()` function before. Since that the model object is a pre-trained language model, the things we can do with this model are limited for now. We need to train it on a downstream task to able to use it for inference, which will be the main subject of further chapters. Instead, we can take advantage of its masked language model objective as follows:

```
from Transformers import pipeline
fillmask= pipeline('fill-mask', model='albert-base-v2')
pd.DataFrame(fillmask("The cat is so [MASK] ."))
```

The following is the output:

sequence	score	token	token_str
[CLS] the cat is so cute.[SEP]	0.281025	10901	_cute
[CLS] the cat is so adorable.[SEP]	0.094893	26354	_adorable
[CLS] the cat is so happy.[SEP]	0.042963	1700	_happy
[CLS] the cat is so funny.[SEP]	0.040976	5066	_funny
[CLS] the cat is so affectionate.[SEP]	0.024233	28803	_affectionate

Figure 3.15 – The fill-mask output results for albert-base-v2

The `fill-mask` pipeline computes the scores for each vocabulary token with the `SoftMax()` function and sorts the most probable tokens where `cute` is the winner with a probability score of 0.281. You may notice that entries in the *token_str* column start with the _ character, which is due to the metaspace component of the tokenizer of Albert.

Let's take a look at the next alternative, *RoBERTa*, which mostly focuses on the pre-training phase.

RoBERTa

Robustly Optimized BERT pre-training Approach (RoBERTa) is another popular BERT reimplementation. It has provided many more improvements in training strategy than architectural design. It outperformed BERT in almost all individual tasks on GLUE. Dynamic masking is one of its original design choices. Although static masking is better for some tasks, the RoBERTa team showed that dynamic masking can perform well for overall performances. Let's compare the changes from BERT and summarize all the features as follows:

The changes in architecture are as follows:

Removing the next sentence prediction training objective

Dynamically changing the masking patterns instead of static masking, which is done by generating masking patterns whenever they feed a sequence to the model

BPE sub-word tokenizer

The changes in training are as follows:

Controlling the training data: More data is used, such as 160 GB instead of the 16 GB originally used in BERT. Not only the size of the data but the quality and diversity were taken into consideration in the study.

Longer iterations of up to 500K pretraining steps.

A longer batch size.

Longer sequences, which leads to less padding.

A large 50K BPE vocabulary instead of a 30K BPE vocabulary.

Thanks to the Transformers uniform API, as in the Albert model pipeline above, we initialize the RoBERTa model as follows:

Copy Explain

```
>>> from Transformers import RobertaConfig, RobertaModel
>>> conf= RobertaConfig()
>>> model = RobertaModel(conf)
>>> print(f"{model.num_parameters() /(10**6)}\
million parameters")
109.48224 million parameters
```

In order to load the pre-trained model, we execute the following pieces of code:

Copy Explain

```
from Transformers import RobertaTokenizer, RobertaModel
tokenizer = \
RobertaTokenizer.from_pretrained('roberta-base')
model = RobertaModel.from_pretrained('roberta-base')
text = "The cat is so sad ."
encoded_input = tokenizer(text, return_tensors='pt')
output = model(**encoded_input)
```

These lines illustrate how the model processes a given text. The output representation at the last layer is not useful at the moment. As we've mentioned several times, we need to fine-tune the main language models. The following execution applies the `fill-mask` function using the `roberta-base` model:

```
>>> from Transformers import pipeline
>>> fillmask= pipeline("fill-mask ",model="roberta-base",
                       tokenizer=tokenizer)
>>> pd.DataFrame(fillmask("The cat is so <mask> ."))
```

The following is the output:

sequence	score	token	token_str
<s>The cat is so cute.</s>	0.191843	11962	Ġcute
<s>The cat is so sweet.</s>	0.051524	4045	Ġsweet
<s>The cat is so funny.</s>	0.033595	6269	Ġfunny
<s>The cat is so handsome.</s>	0.032893	19222	Ġhandsome
<s>The cat is so beautiful.</s>	0.032314	2721	Ġbeautiful

Figure 3.16 – The fill-mask task results for roberta-base

Like the previous ALBERT `fill-mask` model, this pipeline ranks the suitable candidate words. Please ignore the prefix `Ġ` in the tokens – that is an encoded space character produced by the byte-level BPE tokenizer, which we will discuss later. You should have noticed that we used the `[MASK]` and `<mask>` tokens in ALBERT and RoBERTa pipeline in order to hold place for masked token. This is because of the configuration of `tokenizer`. To learn which token expression will be used, you can check `tokenizer.mask_token`. Please see the following execution:

```
>>> tokenizer = \
 AlbertTokenizer.from_pretrained('albert-base-v2')
>>> print(tokenizer.mask_token)
[MASK]
>>> tokenizer = \
RobertaTokenizer.from_pretrained('roberta-base')
>>> print(tokenizer.mask_token)
<mask>
```

To ensure proper mask token use, we can add the `fillmask.tokenizer.mask_token` expression in the pipeline as follows:

```
fillmask(f"The cat is very\
{fillmask.tokenizer.mask_token}.")
```

ELECTRA

The **ELECTRA** model (*proposed by Kevin Clark et al. in 2020*) focuses on a new masked language model utilizing the replaced token detection training objective. During pre-training, the model is forced to learn to distinguish real input tokens from synthetically generated replacements where the synthetic negative example is sampled from plausible tokens rather than randomly sampled tokens. The Albert model criticized the NSP objective of BERT for being a topic detection problem and using low-quality negative examples. ELECTRA trains two neural networks, a generator and a discriminator, so that the former produces high-quality negative examples, whereas the latter distinguishes the original token from the replaced token. We know GAN networks from the field of computer vision, in which the generator *G* produces fake images and tries to fool the discriminator *D*, and the discriminator network tries to avoid being fooled. The ELECTRA model applies almost the same generator-discriminator approach to replace original tokens with high-quality negative examples that are plausible replacements but synthetically generated.

In order not to repeat the same code with other examples, we only provide a simple `fill-mask` example for the Electra generator as follows:

Copy Explain

```
fillmask = \
pipeline("fill-mask", model="google/electra-small-generator")
fillmask(f"The cat is very \{fillmask.tokenizer.mask_token} .")
```

You can see the entire list of models at the following link: https://huggingface.co/Transformers/model_summary.html.

The model checkpoints can be found at https://huggingface.co/models.

Well done! We've finally completed the autoencoding model part. Now we'll move on to tokenization algorithms, which have an important effect on the success of Transformers.

Working with tokenization algorithms

In the opening part of the chapter, we trained the BERT model using a specific tokenizer, namely `BertWordPieceTokenizer`. Now it is worth discussing the tokenization process in detail here. Tokenization is a way of splitting textual input into tokens and assigning an identifier to each token before feeding the neural network architecture. The most intuitive way is to split the sequence into smaller chunks in terms of space. However, such approaches do not meet the requirement of some languages, such as Japanese, and also may lead to huge vocabulary problems. Almost all Transformer models leverage subword tokenization not only for reducing dimensionality but also for encoding rare (or unknown) words not seen in training. The tokenization relies on the idea that every word, including rare words or unknown words, can be decomposed into meaningful smaller chunks that are widely seen symbols in the training corpus.

Some traditional tokenizers developed within Moses and the `nltk` library apply advanced rule-based techniques. But the tokenization algorithms that are used with Transformers are based on self-supervised learning and extract the rules from the corpus. Simple intuitive solutions for rule-based tokenization are based on using characters, punctuation, or whitespace. Character-based tokenization causes language models to lose the input meaning. Even though it can reduce the vocabulary size, which is good, it makes it hard for the model to capture the meaning of `cat` by means of the encodings of the characters `c`, `a`, and `t`. Moreover, the dimension of the input sequence becomes very large. Likewise, punctuation-based models cannot treat some expressions, such as *haven't* or *ain't*, properly.

Recently, several advanced subword tokenization algorithms, such as BPE, have become an integral part of Transformer architectures. These modern tokenization procedures consist of two phases: The pre-tokenization phase simply splits the input into tokens either using space as or language-dependent rules. Second, the tokenization training phase is to train the tokenizer and build a base vocabulary of a reasonable size based on tokens. Before training our own tokenizers, let's load a pre-trained tokenizer. The following code loads a Turkish tokenizer, which is of type `BertTokenizerFast`, from the `Transformers` library with a vocabulary size of 32K:

Copy Explain

```
>>> from Transformers import AutoModel, AutoTokenizer
>>> tokenizerTUR = AutoTokenizer.from_pretrained(
                   "dbmdz/bert-base-turkish-uncased")
>>> print(f"VOC size is: {tokenizerTUR.vocab_size}")
>>> print(f"The model is: {type(tokenizerTUR)}")
VOC size is: 32000
The model is: Transformers.models.bert.tokenization_bert_fast.BertTokenizerFast
```

The following code loads an English BERT tokenizer for the `bert-base-uncased` model:

Copy Explain

```
>>> from Transformers import AutoModel, AutoTokenizer
>>> tokenizerEN = \
 AutoTokenizer.from_pretrained("bert-base-uncased")
>>> print(f"VOC size is: {tokenizerEN.vocab_size}")
>>> print(f"The model is {type(tokenizerEN)}")
VOC size is: 30522
The model is ... BertTokenizerFast
```

Let's see how they work! We tokenize the word `telecommunication` with these two tokenizers:

Copy Explain

```
>>> word_en="telecommunication"
>>> print(f"is in Turkish Model ? \
{word_en in tokenizerTUR.vocab}")
>>> print(f"is in English Model ? \
{word_en in tokenizerEN.vocab}")
is in Turkish Model ? False
is in English Model ? True
```

The `word_en` token is already in the vocabulary of the English tokenizer but not in that of the Turkish one. So let's see what happens with the Turkish tokenizer:

Copy Explain

```
>>> tokens=tokenizerTUR.tokenize(word_en)
>>> tokens
['tel', '##eco', '##mm', '##un', '##ica', '##tion']
```

Since the Turkish tokenizer model has no such a word in its vocabulary, it needs to break the word into parts that make sense to it. All these split tokens are already stored in the model vocabulary. Please notice the output of the following execution:

Copy Explain

```
>>> [t in tokenizerTUR.vocab for t in tokens]
[True, True, True, True, True, True]
```

Let's tokenize the same word with the English tokenizer that we already loaded:

```
>>> tokenizerEN.tokenize(word_en)
['telecommunication']
```

Since the English model has the word `telecommunication` in the base vocabulary, it does not need to break it into parts but rather takes it as a whole. By learning from the corpus, the tokenizers are capable of transforming a word into mostly grammatically logical subcomponents. Let's take a difficult example from Turkish. As an agglutinative language, Turkish allows us to add many suffixes to a word stem to construct very long words. Here is one of the longest words in the Turkish language used in a text (https://en.wikipedia.org/wiki/Longest_word_in_Turkish):

Muvaffakiyetsizleştiricileştiriveremeyebileceklerimizdenmişsinizcesine

It means that *As though you happen to have been from among those whom we will not be able to easily/quickly make a maker of unsuccessful ones*. The Turkish BERT tokenizer may not have seen this word in training, but it has seen its pieces; *muvaffak (succesful) as the stem, ##iyet(successfulness), ##siz (unsuccessfulness), ##leş (become unsuccessful)*, and so forth. The Turkish tokenizer extracts components that seem to be grammatically logical for the Turkish language when comparing the results with a Wikipedia article:

```
>>> print(tokenizerTUR.tokenize(long_word_tur))
['muvaffak', '##iyet', '##siz', '##les', '##tir', '##ici', '##les', '##tir', '##iver',
'##emeye', '##bilecekleri', '##mi', '##z', '##den', '##mis', '##siniz', '##cesine']
```

The Turkish tokenizer is an example of the WordPiece algorithm since it works with a BERT model. Almost all language models including BERT, DistilBERT, and ELECTRA require a WordPiece tokenizer.

Now we are ready to take a look at the tokenization approaches used with Transformers. First, we'll discuss the widely used tokenizations of BPE, WordPiece, and SentencePiece a bit and then train them with HuggingFace's fast `tokenizers` library.

Byte pair encoding

BPE is a data compression technique. It scans the data sequence and iteratively replaces the most frequent pair of bytes with a single symbol. It was first adapted and proposed in *Neural Machine Translation of Rare Words with Subword Units,*

Sennrich et al. 2015, to solve the problem of unknown words and rare words for machine translation. Currently, it is successfully being used within GPT-2 and many other state-of-the-art models. Many modern tokenization algorithms are based on such compression techniques.

It represents text as a sequence of character n-grams, which are also called character-level subwords. The training starts initially with a vocabulary of all Unicode characters (or symbols) seen in the corpus. This can be small for English but can be large for character-rich languages such as Japanese. Then, it iteratively computes character bigrams and replaces the most frequent ones with special new symbols. For example, *t* and *h* are frequently occurring symbols. We replace consecutive symbols with the *th* symbol. This process is kept iteratively running until the vocabulary has attained the desired vocabulary size. The most common vocabulary size is around 30K.

BPE is particularly effective at representing unknown words. However, it may not guarantee the handling of rare words and/or words including rare subwords. In such cases, it associates rare characters with a special symbol, *<UNK>*, which may lead to losing meaning in words a bit. As a potential solution, **Byte-Level BPE (BBPE)** has been proposed, which uses a 256-byte set of vocabulary instead of Unicode characters to ensure that every base character is included in the vocabulary.

WordPiece tokenization

WordPiece is another popular word segmentation algorithm widely used with BERT, DistilBERT, and Electra. It was proposed by Schuster and Nakajima to solve the Japanese and Korean voice problem in 2012. The motivation behind this work was that, although not a big issue for the English language, word segmentation is important preprocessing for many Asian languages, because in these languages spaces are rarely used. Therefore, we come across word segmentation approaches in NLP studies in Asian languages more often. Similar to BPE, WordPiece uses a large corpus to learn vocabulary and merging rules. While BPE and BBPE learn the merging rules based on co-occurrence statistics, the WordPiece algorithm uses maximum likelihood estimation to extract the merging rules from a corpus. It first initializes the vocabulary with Unicode characters, which are also called vocabulary symbols. It treats each word in the training corpus as a list of symbols (initially Unicode characters), and then it iteratively produces a new symbol merging two symbols out of all the possible candidate symbol pairs based on the likelihood maximization rather than frequency. This production pipeline continues until the desired vocabulary size is reached.

Sentence piece tokenization

Previous tokenization algorithms treat text as a space-separated word list. This space-based splitting does not work in some languages. In the German language, compound nouns are written without spaces, for example, menschenrechte (human rights). The solution is to use language-specific pre-tokenizers. In German, an NLP pipeline leverages a compound-splitter module to check whether a word can be subdivided into smaller words. However, East Asian languages (for example, Chinese, Japanese, Korean, and Thai) do not use spaces between words. The **SentencePiece** algorithm is designed to overcome this space limitation, which is a simple and language-independent tokenizer proposed by Kudo et al. in 2018. It treats the input as a raw input stream where space is part of the character set. The tokenizer using SentencePiece produces the _ character, which is also why we saw _ in the output of the Albert model example earlier. Other popular language models that use SentencePiece are XLNet, Marian, and T5.

So far, we have discussed subword tokenization approaches. It is time to start conducting experiments for training with the `tokenizers` library.

The tokenizers library

You may have noticed that the already-trained tokenizers for Turkish and English are part of the `Transformers` library in the previous code examples. On the other hand, the HuggingFace team provided the `tokenizers` library independently from the `Transformers` library to be fast and give us more freedom. The library was originally written in Rust, which makes multi-core parallel computations possible and is wrapped with Python (https://github.com/huggingface/tokenizers).

To install the `tokenizers` library, we use this:

Copy Explain

```
$ pip install tokenizers
```

The `tokenizers` library provides several components so that we can build an end-to-end tokenizer from preprocessing the raw text to decoding tokenized unit IDs:

Normalizer→ PreTokenizer → Modeling → Post-Processor → Decoding

The following diagram depicts the tokenization pipeline:

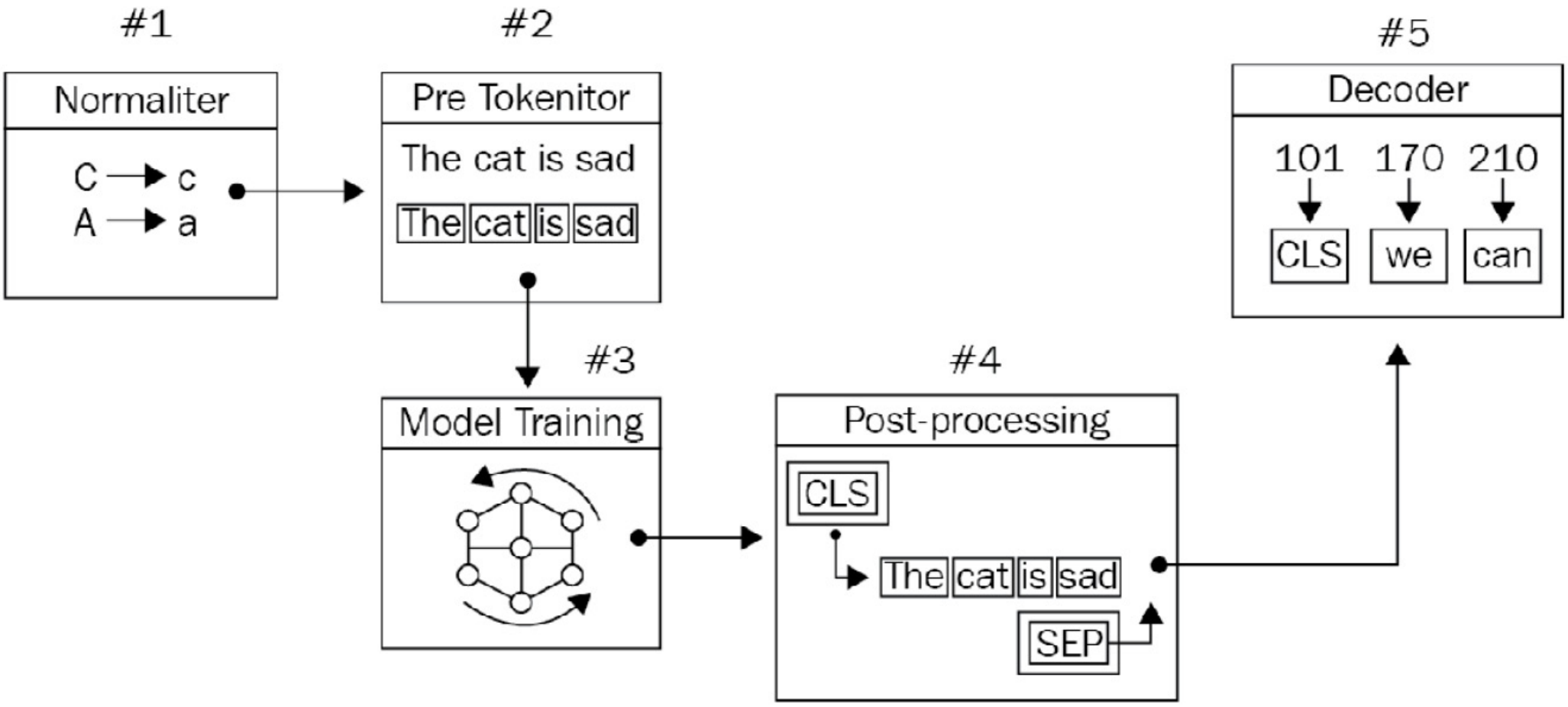

Figure 3.17 – Tokenization pipeline

Normalizer allows us to apply primitive text processing such as lowercasing, stripping, Unicode normalization, and removing accents.

PreTokenizer prepares the corpus for the next training phase. It splits the input into tokens depending on the rules, such as whitespace.

Model Training is a subword tokenization algorithm such as *BPE*, *BBPE*, and *WordPiece*, which we've discussed already. It discovers subwords/vocabulary and learns generation rules.

Post-processing provides advanced class construction that is compatible with Transformers models such as BertProcessors. We mostly add special tokens such as *[CLS]* and *[SEP]* to the tokenized input just before feeding the architecture.

Decoder is in charge of converting token IDs back to the original string. It is just for inspecting what is going on.

Training BPE

Let's train a BPE tokenizer using Shakespeare's plays:

1. It is loaded as follows:

```
import nltk
from nltk.corpus import gutenberg
nltk.download('gutenberg')
nltk.download('punkt')
plays=['shakespeare-macbeth.txt','shakespeare-hamlet.txt',
      'shakespeare-caesar.txt']
shakespeare=[" ".join(s) for ply in plays \
for s in gutenberg.sents(ply)]
```

We will need a post-processor (`TemplateProcessing`) for all the tokenization algorithms ahead. We need to customize the post-processor to make the input convenient for a particular language model. For example, the following template will be suitable for the BERT model since it needs the *[CLS]* token at the beginning of the input and *[SEP]* tokens both at the end and in the middle.

2. We define the template as follows:

```
from tokenizers.processors import TemplateProcessing
special_tokens=["[UNK]","[CLS]","[SEP]","[PAD]","[MASK]"]
temp_proc= TemplateProcessing(
    single="[CLS] $A [SEP]",
    pair="[CLS] $A [SEP] $B:1 [SEP]:1",
    special_tokens=[
        ("[CLS]", special_tokens.index("[CLS]")),
        ("[SEP]", special_tokens.index("[SEP]")),
    ],
)
```

3. We import the necessary components to build an end-to-end tokenization pipeline:

```
from tokenizers import Tokenizer
from tokenizers.normalizers import \
(Sequence,Lowercase, NFD, StripAccents)
from tokenizers.pre_tokenizers import Whitespace
from tokenizers.models import BPE
from tokenizers.decoders import BPEDecoder
```

4. We start by instantiating **BPE** as follows:

```
tokenizer = Tokenizer(BPE())
```

5. The preprocessing part has two components: *normalizer* and *pre-tokenizer*. We may have more than one normalizer. So, we compose a `Sequence` of normalizer components that includes multiple normalizers where `NFD()` is a Unicode normalizer and `StripAccents()` removes accents. For pre-tokenization, `Whitespace()` gently breaks the text based on space. Since the decoder component must be compatible with the model, `BPEDecoder` is selected for the `BPE` model:

```
tokenizer.normalizer = Sequence(
[NFD(),Lowercase(),StripAccents()])
tokenizer.pre_tokenizer = Whitespace()
tokenizer.decoder = BPEDecoder()
tokenizer.post_processor=temp_proc
```

6. Well! We are ready to train the tokenizer on the data. The following execution instantiates `BpeTrainer()`, which helps us to organize the entire training process by setting hyperparameters. We set the vocabulary size parameter to 5K since our Shakespeare corpus is relatively small. For a large-scale project, we use a bigger corpus and normally set the vocabulary size to around 30K:

```
>>> from tokenizers.trainers import BpeTrainer
>>> trainer = BpeTrainer(vocab_size=5000,
                         special_tokens= special_tokens)
>>> tokenizer.train_from_iterator(shakespeare,
                                  trainer=trainer)
>>> print(f"Trained vocab size:\
{tokenizer.get_vocab_size()}" )
Trained vocab size: 5000
```

   We have completed the training!

   Important note

   Training from the filesystem: To start the training process, we passed an in-memory Shakespeare object as a list of strings to `tokenizer.train_from_iterator()`. For a large-scale project with a large corpus, we need to design a Python generator that yields string lines mostly by consuming the files from the filesystem rather than in-memory storage. You should also check `tokenizer.train()` to train from the filesystem storage as applied in the BERT training section above.

7. Let's grab a random sentence from the play Macbeth, name it `sen`, and tokenize it with our fresh tokenizer:

```
>>> sen= "Is this a dagger which I see before me,\
 the handle toward my hand?"
>>> sen_enc=tokenizer.encode(sen)
>>> print(f"Output: {format(sen_enc.tokens)}")
Output: ['[CLS]', 'is', 'this', 'a', 'dagger', 'which', 'i', 'see', 'before',
'me', ',', 'the', 'hand', 'le', 'toward', 'my', 'hand', '?', '[SEP]']
```

8. Thanks to the post-processor function above, we see additional *[CLS]* and *[SEP]* tokens in the proper position. There is only one split word, *handle* (*hand*, *le*), since we passed to the model a sentence from the play Macbeth that the model already knew. Besides, we used a small corpus, and the tokenizer is not forced to use compression. Let's pass a challenging phrase, `Hugging Face`, that the tokenizer might not know:

```
>>> sen_enc2=tokenizer.encode("Macbeth and Hugging Face")
>>> print(f"Output: {format(sen_enc2.tokens)}")
Output: ['[CLS]', 'macbeth', 'and', 'hu', 'gg', 'ing', 'face', '[SEP]']
```

9. The term `Hugging` is lowercased and split into three pieces `hu`, `gg`, `ing`, since the model vocabulary contains all other tokens but `Hugging`. Let's pass two sentences now:

```
>>> two_enc=tokenizer.encode("I like Hugging Face!",
"He likes Macbeth!")
>>> print(f"Output: {format(two_enc.tokens)}")
Output: ['[CLS]', 'i', 'like', 'hu', 'gg', 'ing', 'face', '!', '[SEP]', 'he',
'li', 'kes', 'macbeth', '!', '[SEP]']
```

Notice that the post-processor injected the `[SEP]` token as an indicator.

10. It is time to save the model. We can either save the sub-word tokenization model or the entire tokenization pipeline. First, let's save the BPE model only:

```
>>> tokenizer.model.save('.')
['./vocab.json', './merges.txt']
```

11. The model saved two files regarding vocabulary and merging rules. The `merge.txt` file is composed of 4,948 merging rules:

```
$ wc -l ./merges.txt
4948 ./merges.txt
```

12. The top five rules ranked are as shown in the following where we see that (`t`, `h`) is the first ranked rule due to that being the most frequent pair. For testing, the model scans the textual input and tries to merge these two symbols first if applicable:

```
$ head -3 ./merges.txt
t h
o u
a n
th e
r e
```

The BPE algorithm ranks the rules based on frequency. When you manually calculate character bigrams in the Shakespeare corpus, you will find (`t`, `h`) the most frequent pair.

13. Let's now save and load the entire tokenization pipeline:

```
>>> tokenizer.save("MyBPETokenizer.json")
>>> tokenizerFromFile = \
Tokenizer.from_file("MyBPETokenizer.json")
>>> sen_enc3 = \
tokenizerFromFile.encode("I like Hugging Face and Macbeth")
>>> print(f"Output: {format(sen_enc3.tokens)}")
Output: ['[CLS]', 'i', 'like', 'hu', 'gg', 'ing', 'face', 'and', 'macbeth',
'[SEP]']
```

We successfully reloaded the tokenizer!

Training the WordPiece model

In this section, we will train the WordPiece model:

1. We start by importing the necessary modules:

```
from tokenizers.models import WordPiece
from tokenizers.decoders import WordPiece \
as WordPieceDecoder
from tokenizers.normalizers import BertNormalizer
```

2. The following lines instantiate an empty WordPiece tokenizer and prepare it for training. `BertNormalizer` is a pre-defined normalizer sequence that includes the processes of cleaning the text, transforming accents, handling Chinese characters, and lowercasing:

```
tokenizer = Tokenizer(WordPiece())
tokenizer.normalizer=BertNormalizer()
tokenizer.pre_tokenizer = Whitespace()
tokenizer.decoder= WordPieceDecoder()
```

3. Now, we instantiate a proper trainer, `WordPieceTrainer()` for `WordPiece()`, to organize the training process:

```
>>> from tokenizers.trainers import WordPieceTrainer
>>> trainer = WordPieceTrainer(vocab_size=5000,\
            special_tokens=["[UNK]", "[CLS]", "[SEP]",\
            "[PAD]", "[MASK]"])
>>> tokenizer.train_from_iterator(shakespeare,
trainer=trainer)
>>> output = tokenizer.encode(sen)
>>> print(output.tokens)
['is', 'this', 'a', 'dagger', 'which', 'i', 'see', 'before', 'me', ',', 'the',
'hand', '##le', 'toward', 'my', 'hand', '?']
```

4. Let's use `WordPieceDecoder()` to treat the sentences properly:

```
>>> tokenizer.decode(output.ids)
'is this a dagger which i see before me, the handle toward my hand?'
```

5. We have not come across any `[UNK]` tokens in the output since the tokenizer somehow knows or splits the input for encoding. Let's force the model to produce `[UNK]` tokens as in the following code. Let's pass a Turkish sentence to our tokenizer:

```
>>> tokenizer.encode("Kralsın aslansın Macbeth!").tokens
'[UNK]', '[UNK]', 'macbeth', '!']
```

Well done! We have a couple of unknown tokens since the tokenizer does not find a way to decompose the given word from the merging rules and the base vocabulary.

So far, we have designed our tokenization pipeline all the way from the normalizer component to the decoder component. On the other hand, the `tokenizers` library provides us with an already made (not trained) empty tokenization pipeline with proper components to build quick prototypes for production. Here are some pre-made tokenizers:

- `CharBPETokenizer`: The original BPE
- `ByteLevelBPETokenizer`: The byte-level version of the BPE
- `SentencePieceBPETokenizer`: A BPE implementation compatible with the one used by *SentencePiece*
- `BertWordPieceTokenizer`: The famous BERT tokenizer, using WordPiece

The following code imports these pipelines:

```
>>> from tokenizers import (ByteLevelBPETokenizer,
                            CharBPETokenizer,
                            SentencePieceBPETokenizer,
                            BertWordPieceTokenizer)
```

All these pipelines are already designed for us. The rest of the process (such as training, saving the model, and using the tokenizer) is the same as our previous BPE and WordPiece training procedure.

Well done! We have made great progress and trained our first Transformer model as well as its tokenizer.

Summary

In this chapter, we have experienced autoencoding models both theoretically and practically. Starting with basic knowledge about BERT, we trained it as well as a corresponding tokenizer from scratch. We also discussed how to work inside other frameworks, such as Keras. Besides BERT, we also reviewed other autoencoding models. To avoid excessive code repetition, we did not provide the full implementation

for training other models. During the BERT training, we trained the WordPiece tokenization algorithm. In the last part, we examined other tokenization algorithms since it is worth discussing and understanding all of them.

Autoencoding models use the left decoder side of the original Transformer and are mostly fine-tuned for classification problems. In the next chapter, we will discuss and learn about the right decoder part of Transformers to implement language generation models.

Previous Chapter

Next Chapter