Chapter 4:Autoregressive and Other Language Models

We looked at details of **Autoencoder** (**AE**) language models in *Chapter 3*, *Autoencoding Language Models*, and studied how an AE language model can be trained from scratch. In the current chapter, you will see theoretical details of **Autoregressive** (**AR**) language models and learn how to pre-train them on your own corpus. You will learn how to pre-train any language model such as **Generated Pre-trained Transformer 2** (**GPT-2**) on your own text and use it in various tasks such as **Natural Language Generation** (**NLG**). You will understand the basics of a **Text-to-Text Transfer Transformer** (**T5**) model and train a **Multilingual T5** (**mT5**) model on your own **Machine Translation** (**MT**) data. After finishing this chapter, you will have an overview of AR language models and their various use cases in `text2tex`t applications, such as summarization, paraphrasing, and MT.

The following topics will be covered in this chapter:

- Working with AR language models
- Working with **Sequence-to-Sequence** (**Seq2Seq**) models
- AR language model training
- NLG using AR models
- Summarization and MT fine-tuning using `simpletransformers`

Technical requirements

The following libraries/packages are required to successfully complete this chapter:

- Anaconda
- `transformers 4.0.0`

```
pytorch 1.0.2
tensorflow 2.4.0
datasets 1.4.1
tokenizers
simpletransformers 0.61
```

All notebooks with coding exercises will be available at the following GitHub link: https://github.com/PacktPublishing/Mastering-Transformers/tree/main/CH04.

Check out the following link to see the Code in Action: https://bit.ly/3yjn55X

Working with AR language models

The Transformer architecture was originally intended to be effective for Seq2Seq tasks such as MT or summarization, but it has since been used in diverse NLP problems ranging from token classification to coreference resolution. Subsequent works began to use separately and more creatively the left and right parts of the architecture. The objective, also known as **denoising objective**, is to fully recover the original input from the corrupted one in a bidirectional fashion, as shown on the left side of *Figure 4.1*, which you will see shortly. As seen in the **Bidirectional Encoder Representations from Transformers** (**BERT**) architecture, which is a notable example of AE models, they can incorporate the context of both sides of a word. However, the first issue is that the corrupting `[MASK]` symbols that are used during the pre-training phase are absent from the data during the fine-tuning phase, leading to a pre-training-fine-tuning discrepancy. Secondly, the BERT model arguably assumes that the masked tokens are independent of each other.

On the other hand, AR models keep away from such assumptions regarding independence and do not naturally suffer from the pre-train-fine-tuning discrepancy because they rely on the objective predicting the next token conditioned on the previous tokens without masking them. They merely utilize the decoder part of the transformer with masked self-attention. They prevent the model from accessing words to the right of the current word in a forward direction (or to the left of the current word in a backward direction), which is called **unidirectionality**. They are also called **Causal Language Models** (**CLMs**) due to their unidirectionality.

The difference between AE and AR models is simply depicted here:

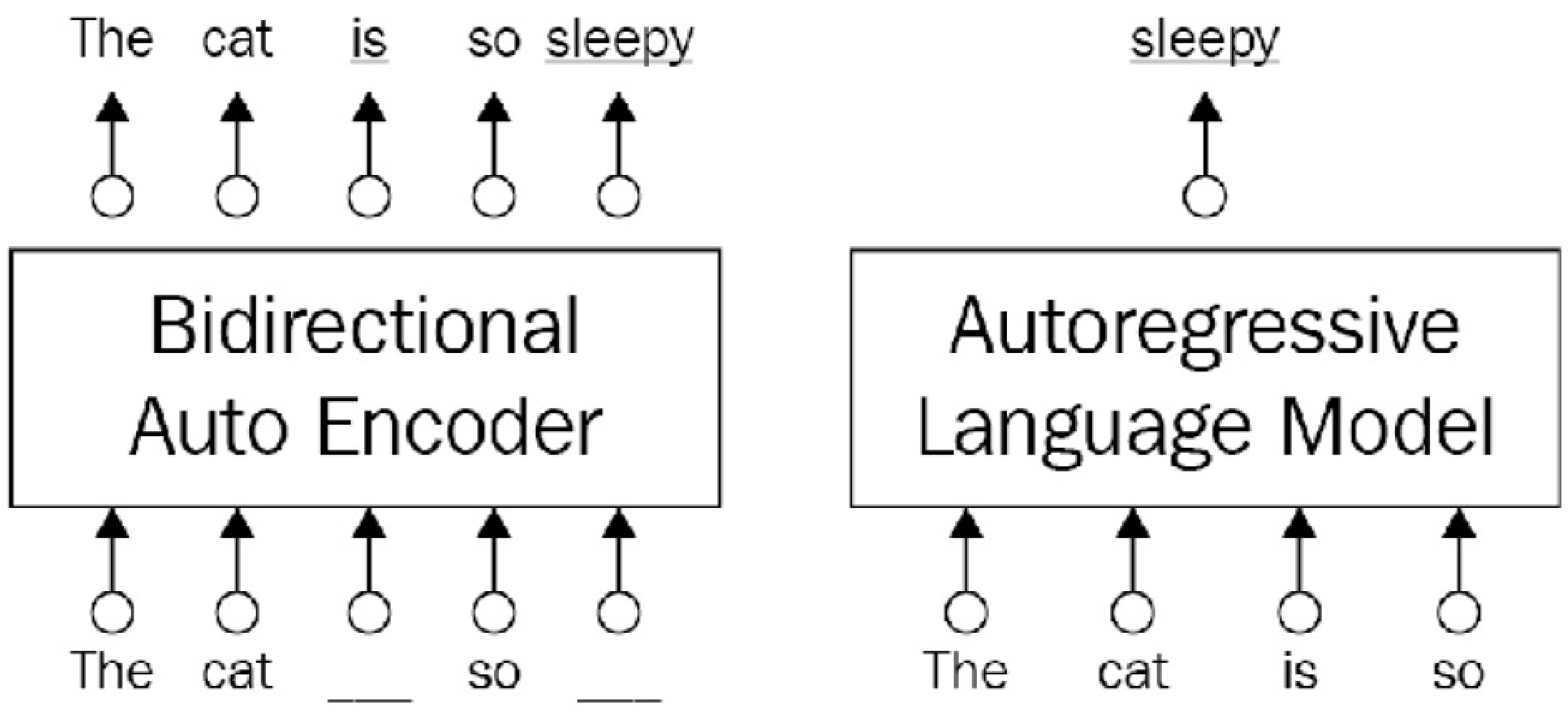


Figure 4.1 – AE versus AR language model

GPT and its two successors (GPT-2, GPT-3), **Transformer-XL**, and **XLNet** are among the popular AR models in the literature. Even though XLNet is based on autoregression, it somehow managed to make use of both contextual sides of the word in a bidirectional fashion, with the help of the permutation-based language objective. Now, we start introducing them and show how to train the models with a variety of experiments. Let's look at GPTs first.

Introduction and training models with GPT

AR models are made up of multiple transformer blocks. Each block contains a masked multi-head self-attention layer along with a pointwise feed-forward layer. The activation in the final transformer block is fed into a softmax function that produces the word-probability distributions over an entire vocabulary of words to predict the next word.

In the original GPT paper *Improving Language Understanding by Generative Pre-Training* (2018), the authors addressed several bottlenecks that traditional **Machine Learning** (**ML**)-based **Natural Language Processing** (**NLP**) pipelines are subject to. For example, these pipelines firstly require both a massive amount of task-specific data and task-specific architecture. Secondly, it is hard to apply task-aware input transformations with minimal changes to the architecture of the pre-trained model. The original GPT and its successors (GPT-2 and GPT-3), designed by the OpenAI team, have focused on solutions to alleviate these bottlenecks. The major contribution of the original GPT study is that the pre-trained model achieved satisfactory results, not only for a single task but a diversity of tasks. Having learned the generative model from unlabeled data, which is called unsupervised pre-training, the model is simply fine-tuned to a downstream task by a relatively small amount of task-specific data, which is called **supervised fine-tuning**. This two-stage scheme is widely used in other transformer models, where unsupervised pre-training is followed by supervised fine-tuning.

To keep the GPT architecture as generic as possible, only the inputs are transformed into a task-specific manner, while the entire architecture is kept almost the same. This traversal-style approach converts the textual input into an ordered sequence according to the task so that the pre-trained model can understand the task from it. The left part of *Figure 4.2* (inspired from the original paper) illustrates the transformer architecture and training objectives used in the original GPT work. The right part shows how to transform input for fine-tuning on several tasks.

To put it simply, for a single-sequence task such as text classification, the input is passed through the network as-is, and the linear layer takes the last activations to make a decision. For sentence-pair tasks such as textual entailment, the input that is made up of two sequences is marked with a delimiter, shown as the second example in *Figure 4.2*. In both scenarios, the architecture sees uniform token sequences be processed by the pre-trained model. The delimiter used in this transformation helps the pre-trained model to know which part is premise or hypothesis in the case of textual entailment. Thanks to input transformation, we do not have to make substantial changes in the architecture across the tasks.

You can see a representation of input transformation here:

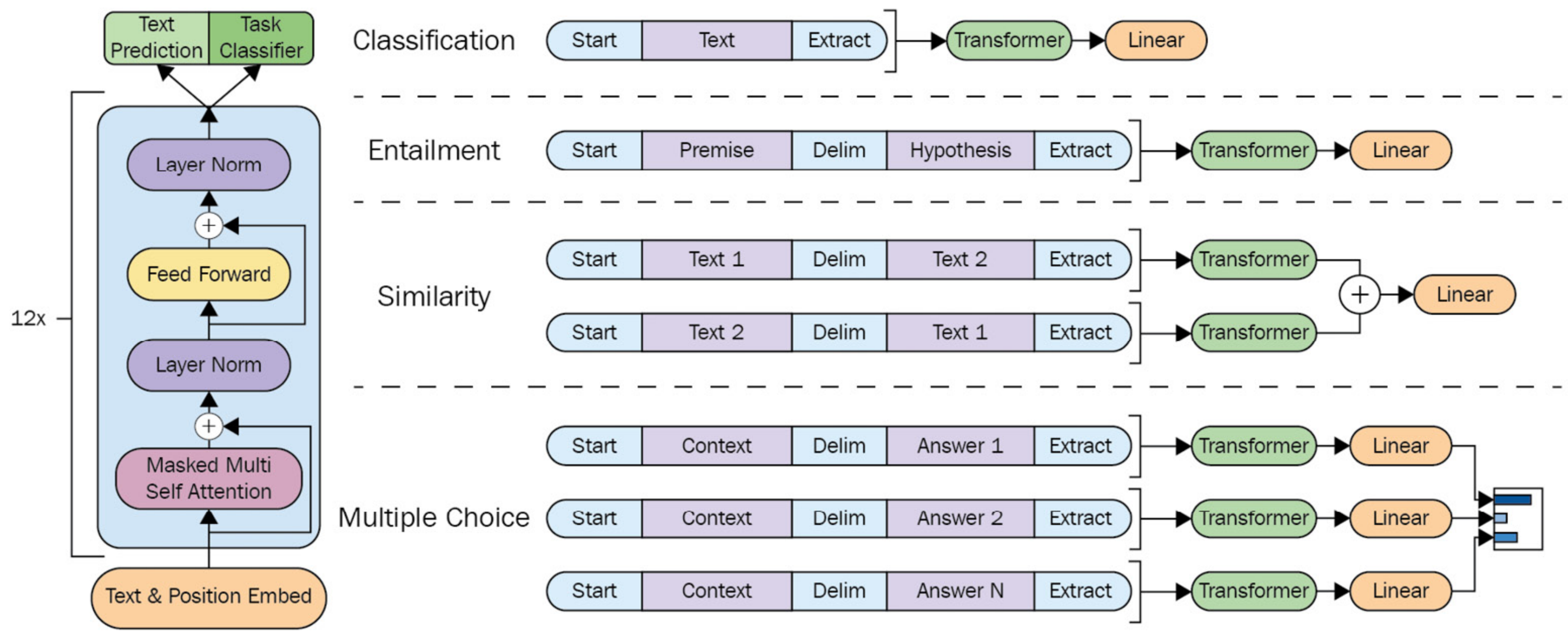


Figure 4.2 – Input transformation (inspired from the paper)

The GPT and its two successors mostly focused on seeking a particular architectural design where the fine-tuning phase was not required. It is based on the idea that a model can be very skilled in the sense that it can learn much of the information about a language during the pre-training phase, with little work left for the fine-tuning phase. Thus, the fine-tuning process can be completed within three epochs and with relatively small examples for most of the tasks. In an extreme case, zero-shot learning aims to disable the fine-tuning phase. The underlying idea is that the model can learn much information about the language during pre-training. This is especially true for all transformer-based models.

Successors of the original GPTs

GPT-2 (see the paper *Language Models are Unsupervised Multitask Learners* (2019)), a successor to the original GPT-1, is a larger model trained on much more training data, called WebText, than the original one. It achieved state-of-the-art results on seven out of the eight tasks in a zero-shot setting in which there is no fine-tuning applied but had limited success in some tasks. It achieved comparable results on smaller datasets for measuring long-range dependency. The GPT-2 authors argued that language models do not necessarily need explicit supervision to learn a task. Instead, they can learn these tasks when trained on a huge and diverse dataset of web pages. It is considered a general system replacing the learning objective *P(output|input)* in the original GPT with *P(output|input, task-i)*, where the model produces the different output for the same input, conditioned on a specific task—that is, GPT-2 learns multiple tasks by training the same unsupervised model. One single pre-trained model learns different abilities just through the learning objective. We see similar formulations in multi-task and meta-task settings in other studies as well. Such a shift to **Multi-Task Learning (MTL**) makes it possible to perform many different tasks for the same input. But how do the models determine which task to perform? They do this through zero-shot task transfer.

Compared to the original GPT, GPT-2 has no task-specific fine-tuning and is able to work in a zero-shot-task-transfer setting, where all the downstream tasks are part of predicting conditional probabilities. The task is somehow formulated within the input, and the model is expected to understand the nature of downstream tasks and provide answers accordingly. For example, for an English-to-Turkish MT task, it is conditioned not only on the input but also on the task. The input is arranged so that an English sentence is followed by a Turkish sentence, with a delimiter from which the model understands that the task is an English-to-Turkish translation.

The OpenAI team trained the GPT-3 model (see the paper *Language models are few-shot learners* (2020)) with 175 billion parameters, which is 100 times bigger than GPT-2. The architecture of GPT-2 and GPT-3 is similar, with the main differences usually being in the model size and the dataset quantity/quality. Due to the massive amount of data in the dataset and the large number of parameters it is trained on, it achieved better results on many downstream tasks in zero-shot, one-shot, and few-shot (*K=32*) settings without any gradient-based fine-tuning. The team showed that the model performance increased as the parameter size and the number of examples increased for many tasks, including translation, **Question Answering** (**QA**), and masked-token tasks.

Transformer-XL

Transformer models suffer from the fixed-length context due to a lack of recurrence in the initial design and context fragmentation, although they are capable of learning long-term dependency. Most of the transformers break the documents into a list of fixed-length (mostly 512) segments, where any information flow across segments is not possible. Consequently, the language models are not able to capture long-term dependencies beyond this fixed-length limit. Moreover, the segmentation procedure builds the segments without paying attention to sentence boundaries. A segment can be absurdly made up of the second half of a sentence and the first half of its successor, hence the language models can miss the necessary contextual information when predicting the next token. This problem is referred to as *context fragmentation* problem by the studies.

To address and overcome these issues, the Transformer-XL authors (see the paper *Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context* (2019)) proposed a new transformer architecture, including a segment-level recurrence mechanism and a new positional encoding scheme. This approach inspired many subsequent models. It is not limited to two consecutive segments since the effective context can extend beyond the two segments. The recurrence mechanism works between every two consecutive segments, leading to spanning the several segments to a certain degree. The largest possible dependency length that the model can attend is limited by the number of layers and segment lengths.

XLNet

Masked Language Modeling (**MLM**) dominated the pre-training phase of transformer-based architectures. However, it has faced criticism in the past since the masked tokens are present in the pre-training phase but are absent during the fine-tuning phase, which leads to a discrepancy between pre-training and fine-tuning. Because of this absence, the model may not be able to use all of the information learned during the pre-training phase. XLNet (see the paper *XLNet: Generalized Autoregressive Pretraining for Language Understanding* (2019)) replaces MLM with **Permuted Language Modeling (PLM)**, which is a random permutation of the input tokens to overcome this bottleneck. The permutation language modeling makes each token position utilize contextual information from all positions, leading to capturing bidirectional context. The objective function only permutes the factorization order and defines the order of token predictions, but doesn't change the natural positions of sequences. Briefly, the model chooses some tokens as a target after permutation, and it further tries to predict them conditioned on the remaining tokens and the natural positions of the target. It makes it possible to use an AR model in a bidirectional fashion.

XLNet takes advantage of both AE and AR models. It is, indeed, a generalized AR model; however, it can attend the tokens from both left and right contexts, thanks to permutation-based language modeling. Besides its objective function, XLNet is made up of two important mechanisms: it integrates the segment-level recurrence mechanism of Transformer-XL into its framework, and it includes the careful design of the two-stream attention mechanism for target-aware representations.

Let's discuss the models, using both parts of the Transformers in the next section.

Working with Seq2Seq models

The left encoder and the right decoder part of the transformer are connected with cross-attention, which helps each decoder layer attend over the final encoder layer. This naturally pushes models toward producing output that closely ties to the original input. A Seq2Seq model, which is the original transformer, achieves this by using the following scheme:

Input tokens-> embeddings-> encoder-> decoder-> output tokens

Seq2Seq models keep the encoder and decoder part of the transformer. T5, **Bidirectional and Auto-Regressive Transformer (BART)**, and **Pre-training with Extracted Gap-sentences for Abstractive Summarization Sequence-to-Sequence models (PEGASUS)** are among the popular Seq2Seq models.

T5

Most NLP architectures, ranging from Word2Vec to transformers learn embeddings and other parameters by predicting the masked words using context (neighbor) words. We treat NLP problems as word prediction problems. Some studies cast almost all NLP problems as QA or token classification. Likewise, T5 (see the paper *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer* (2019)) proposed a unifying framework to solve many tasks by casting them to a text-to-text problem. The idea underlying T5 is to cast all NLP tasks to a text-to-text (Seq2Seq) problem where both input and output are a list of tokens because the text-to-text framework has been found to be beneficial in applying the same model to diverse NLP tasks from QA to text summarization.

The following diagram, which is inspired from the original paper, shows how T5 solves four different NLP problems—MT, linguistic acceptability, semantic similarity, and summarization—within a unified framework:

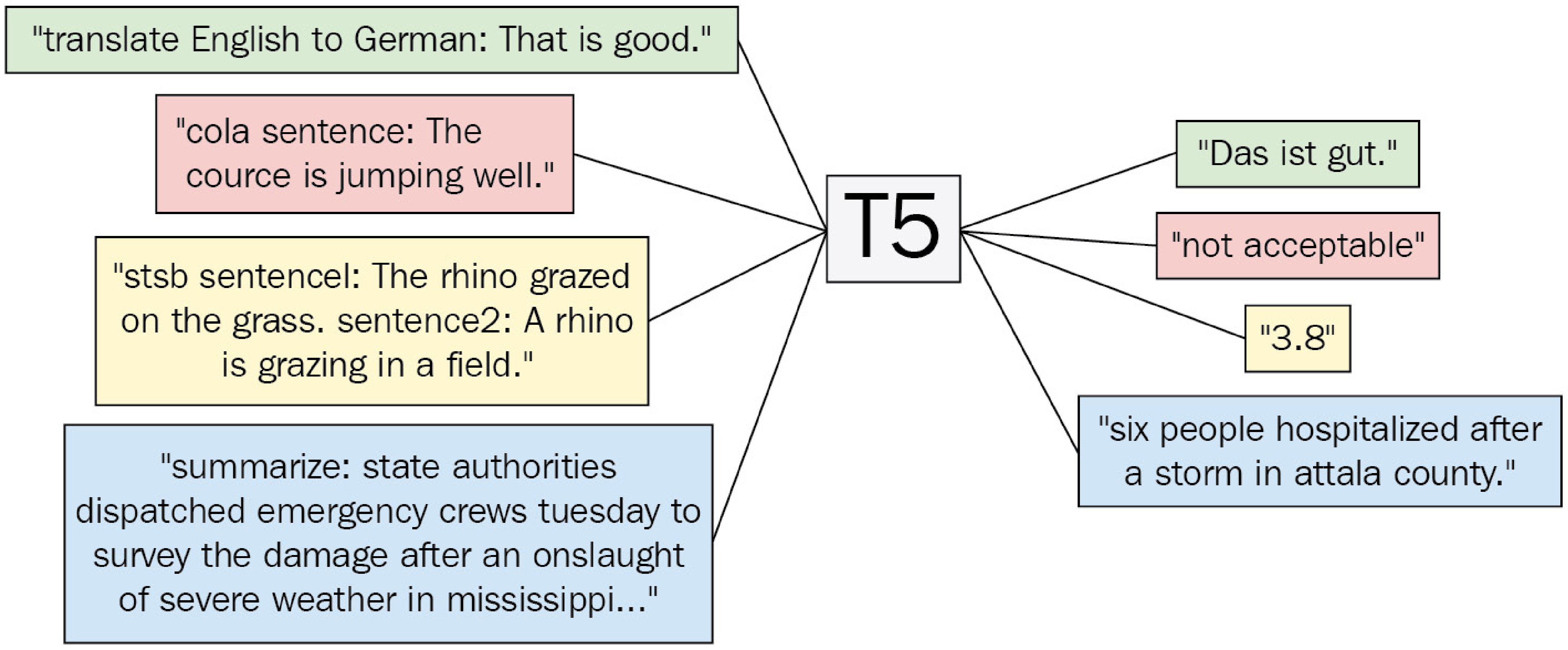

"translate English to German: That is good."
"cola sentence: The cource is jumping well."
"stsb sentence1: The rhino grazed on the grass. sentence2: A rhino is grazing in a field."
"summarize: state authorities dispatched emergency crews tuesday to survey the damage after an onslaught of severe weather in mississippi..."
T5
"Das ist gut."
"not acceptable"
"3.8"
"six people hospitalized after a storm in attala county."

Figure 4.3 – Diagram of the T5 framework

The T5 model roughly follows the original encoder-decoder transformer model. The modifications are done in the layer normalization and position embeddings scheme. Instead of using sinusoidal positional embedding or learned embedding, T5 uses relative positional embedding, which is becoming more common in transformer architectures. T5 is a single model that can work on a diverse set of tasks such as language generation. More importantly, it casts tasks into a text-to-text format. The model is fed with text that is made up of a task prefix and the input attached to it. We convert a labeled textual dataset to a `{'inputs': '....', 'targets': ...'}` format, where we insert the purpose in the input as a prefix. Then, we train the model with labeled data so that it learns what to do and how to do it. As shown in the preceding diagram, for the English-German translation task, the `"translate`

`English to German: That is good."` input is going to produce `"das is gut."`. Likewise, any input with a `"summarize:"` prefix will be summarized by the model.

Introducing BART

As with XLNet, the BART model (see the paper *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension* (2019)) takes advantage of the schemes of AE and AR models. It uses standard Seq2Seq transformer architecture, with a small modification. BART is a pre-trained model using a variety of noising approaches that corrupt documents. The major contribution of the study to the field is that it allows us to apply several types of creative corruption schemes, as shown in the following diagram:

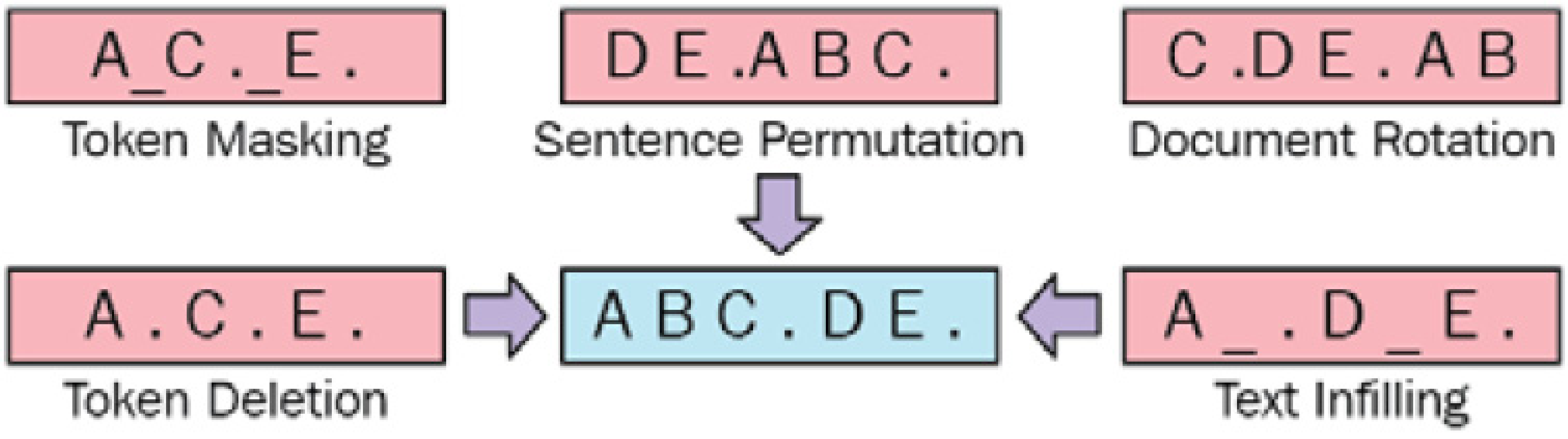


Figure 4.4 – Diagram inspired by the original BART paper

We will look at each scheme in detail, as follows:

- **Token Masking:** Tokens are randomly masked with a `[MASK]` symbol, the same as with the BERT model.
- **Token Deletion:** Tokens are randomly removed from the documents. The model is forced to determine which positions are removed.
- **Text Infilling:** Following SpanBERT, a number of text spans are sampled, and then they are replaced by a single `[MASK]` token. There is also `[MASK]` token insertion.
- **Sentence Permutation:** The sentences in the input are segmented and shuffled in random order.
- **Document Rotation:** The document is rotated so that it begins with a randomly selected token, **C** in the case in the preceding diagram. The objective is to find the start position of a document.

The BART model can be fine-tuned in several ways for downstream applications such as BERT. For the task of sequence classification, the input is passed through encoder and decoder, and the final hidden state of the decoder is considered the learned representation. Then, a simple linear classifier can make predictions. Likewise, for token classification tasks, the entire document is fed into the encoder and decoder, and the last state of the final decoder is the representation for each token. Based on these representations, we can solve the token classification problem, which we will discuss in *Chapter 6*, *Fine-Tuning Language Models for Token Classification*. **Named-Entity Recognition** (**NER**) and **Part-Of-Speech** (**POS**) tasks can be solved using this final representation, where NER identifies entities such as person and organization in a text and POS associates each token with their lexical categories, such as noun, adjective, and so on.

For sequence generation, the decoder block of the BART model, which is an AR decoder, can be directly fine-tuned for sequence-generation tasks such as abstractive QA or summarization. The BART authors (Lewis, Mike, et al.) trained the models using two standard summarization datasets: CNN/DailyMail and XSum. The authors also showed that it is possible to use both the encoder part—which consumes a source language—and the decoder part, which produces the words in the target language as a single pre-trained decoder for MT. They replaced the encoder embedding layer with a new randomly initialized encoder in order to learn words in the source language. Then, the model is trained in an end-to-end fashion, which trains the new encoder to map foreign words into an input that BART can denoise to the target language. The new encoder can use a separate vocabulary, including foreign language, from the original BART model.

In the HuggingFace platform, we can access the original pre-trained BART model with the following line of code:

Copy Explain

```
AutoModel.from_pretrained('facebook/bart-large')
```

When we call the standard `summarization` pipeline of the `transformers` library, as shown in the following line of code, a distilled pre-trained BART model is loaded. This call implicitly loads the `"sshleifer/distilbart-cnn-12-6"` model and the corresponding tokenizers, as follows:

Copy Explain

```
summarizer = pipeline("summarization")
```

The following code explicitly loads the same model and the corresponding tokenizer. The code example takes a text to be summarized and outputs the results:

```
from transformers import BartTokenizer, BartForConditionalGeneration, BartConfig
from transformers import pipeline
model = \
BartForConditionalGeneration.from_pretrained('sshleifer/distilbart-cnn-12-6')
tokenizer = BartTokenizer.from_pretrained('sshleifer/distilbart-cnn-12-6')
nlp=pipeline("summarization", model=model, tokenizer=tokenizer)
text='''
We order two different types of jewelry from this
company the other jewelry we order is perfect.
However with this jewelry I have a few things I
don't like. The little Stone comes out of these
and customers are complaining and bringing them
back and we are having to put new jewelry in their
holes. You cannot sterilize these in an autoclave
as well because it heats up too much and the glue
does not hold up so the second group of these that
we used I did not sterilize them that way and the
stones still came out. When I use a dermal clamp
to put the top on the stones come out immediately.
DO not waste your money on this particular product
buy the three mm. that has the claws that hold the
jewelry in those are perfect. So now I'm stuck
with jewelry that I can't sell not good for
business.'''
q=nlp(text)
import pprint
pp = pprint.PrettyPrinter(indent=0, width=100)
pp.pprint(q[0]['summary_text'])
(' The little Stone comes out of these little stones and customers are complaining and
bringing ' 'them back and we are having to put new jewelry in their holes . You cannot
sterilize these in an ' 'autoclave because it heats up too much and the glue does not
hold up so the second group of ' 'these that we used I did not sterilize them that way
and the stones still came out .')
```

In the next section, we get our hands dirty and learn how to train such models.

AR language model training

In this section, you will learn how it is possible to train your own AR language models. We will start with GPT-2 and get a deeper look inside its different functions for training, using the `transformers` library.

You can find any specific corpus to train your own GPT-2, but for this example, we used *Emma* by Jane Austen, which is a romantic novel. Training on a much bigger corpus is highly recommended to have a more general language generation.

Before we start, it's good to note that we used TensorFlow's native training functionality to show that all Hugging Face models can be directly trained on TensorFlow or PyTorch if you wish to. Follow these steps:

1. You can download the *Emma* novel raw text by using the following command:

```
wget
https://raw.githubusercontent.com/teropa/nlp/master/resources/corpora/gutenberg/aust
emma.txt
```

2. The first step is to train the `BytePairEncoding` tokenizer for GPT-2 on a corpus that you intend to train your GPT-2 on. The following code will import the `BPE` tokenizer from the `tokenizers` library:

```
from tokenizers.models import BPE
from tokenizers import Tokenizer
from tokenizers.decoders import ByteLevel as ByteLevelDecoder
from tokenizers.normalizers import Sequence, Lowercase
from tokenizers.pre_tokenizers import ByteLevel
from tokenizers.trainers import BpeTrainer
```

3. As you see, in this example, we intend to train a more advanced tokenizer by adding more functionality, such as the `Lowercase` normalization. To make a `tokenizer` object, you can use the following code:

```
tokenizer = Tokenizer(BPE())
tokenizer.normalizer = Sequence([
    Lowercase()
])
tokenizer.pre_tokenizer = ByteLevel()
tokenizer.decoder = ByteLevelDecoder()
```

The first line makes a tokenizer from the `BPE` tokenizer class. For the normalization part, `Lowercase` has been added, and the `pre_tokenizer` attribute is set to be as `ByteLevel` to ensure we have bytes as our input. The `decoder` attribute must be also set to `ByteLevelDecoder` to be able to decode correctly.

4. Next, the tokenizer will be trained using a `50000` maximum vocabulary size and an initial alphabet from `ByteLevel`, as follows:

   Copy Explain

   ```
   trainer = BpeTrainer(vocab_size=50000, inital_alphabet=ByteLevel.alphabet(),
   special_tokens=[
               "<s>",
               "<pad>",
               "</s>",
               "<unk>",
               "<mask>"
           ])
   tokenizer.train(["austen-emma.txt"], trainer)
   ```

5. It is also necessary to add special tokens to be considered. To save the tokenizer, you are required to create a directory, as follows:

   Copy Explain

   ```
   !mkdir tokenizer_gpt
   ```

6. You can save the tokenizer by running the following command:

   Copy Explain

   ```
   tokenizer.save("tokenizer_gpt/tokenizer.json")
   ```

7. Now that the tokenizer is saved, it's time to preprocess the corpus and make it ready for GPT-2 training using the saved tokenizer, but first, important imports must not be forgotten. The code to do the imports is illustrated in the following snippet:

   Copy Explain

   ```
   from transformers import GPT2TokenizerFast, GPT2Config, TFGPT2LMHeadModel
   ```

8. And the tokenizer can be loaded by using `GPT2TokenizerFast`, as follows:

   Copy Explain

   ```
   tokenizer_gpt = GPT2TokenizerFast.from_pretrained("tokenizer_gpt")
   ```

9. It is also essential to add special tokens with their marks, like this:

Copy Explain

```
tokenizer_gpt.add_special_tokens({
  "eos_token": "</s>",
  "bos_token": "<s>",
  "unk_token": "<unk>",
  "pad_token": "<pad>",
  "mask_token": "<mask>"
})
```

10. You can also double-check to see if everything is correct or not by running the following code:

Copy Explain

```
tokenizer_gpt.eos_token_id
>> 2
```

This code will output the **End-of-Sentence (EOS)** token **Identifier (ID)**, which is `2` for the current tokenizer.

11. You can also test it for a sentence by executing the following code:

Copy Explain

```
tokenizer_gpt.encode("<s> this is </s>")
>> [0, 265, 157, 56, 2]
```

For this output, `0` is the beginning of the sentence, `265`, `157`, and `56` are related to the sentence itself, and the EOS is marked as `2`, which is `</s>`.

12. These settings must be used when creating a configuration object. The following code will create a `config` object and the TensorFlow version of the GPT-2 model:

Copy Explain

```
config = GPT2Config(
  vocab_size=tokenizer_gpt.vocab_size,
  bos_token_id=tokenizer_gpt.bos_token_id,
  eos_token_id=tokenizer_gpt.eos_token_id
)
model = TFGPT2LMHeadModel(config)
```

13. On running the `config` object, you can see the configuration in dictionary format, as follows:

Copy Explain

```
config
>> GPT2Config {  "activation_function": "gelu_new",  "attn_pdrop": 0.1,
"bos_token_id": 0,  "embd_pdrop": 0.1,  "eos_token_id": 2,
"gradient_checkpointing": false,  "initializer_range": 0.02,
"layer_norm_epsilon": 1e-05,  "model_type": "gpt2",  "n_ctx": 1024,  "n_embd":
768,  "n_head": 12,  "n_inner": null,  "n_layer": 12,  "n_positions": 1024,
"resid_pdrop": 0.1,  "summary_activation": null,  "summary_first_dropout": 0.1,
"summary_proj_to_labels": true,  "summary_type": "cls_index",
"summary_use_proj": true,  "transformers_version": "4.3.2",  "use_cache": true,
"vocab_size": 11750}
```

As you can see, other settings are not touched, and the interesting part is that `vocab_size` is set to `11750`. The reason behind this is that we set the maximum vocabulary size to be `50000`, but the corpus had less, and its **Byte-Pair Encoding (BPE)** token created `11750`.

14. Now, you can get your corpus ready for pre-training, as follows:

Copy Explain

```
with open("austen-emma.txt", "r", encoding='utf-8') as f:
    content = f.readlines()
```

15. The content will now include all raw text from the raw file, but it is required to remove `'\n'` from each line and drop lines with fewer than `10` characters, as follows:

Copy Explain

```
content_p = []
for c in content:
    if len(c)>10:
        content_p.append(c.strip())
content_p = " ".join(content_p)+tokenizer_gpt.eos_token
```

16. Dropping short lines will ensure that the model is trained on long sequences, to be able to generate longer sequences. At the end of the preceding snippet, `content_p` has the concatenated raw file with `eos_token` added to the end. But you can follow different strategies too—for example, you can separate each line by adding `</s>` to each line, which will help the model to recognize when the sentence ends. However, we intend to make it work for much longer sequences without encountering EOS. The code is illustrated in the following snippet:

Copy Explain

```
tokenized_content = tokenizer_gpt.encode(content_p)
```

The GPT tokenizer from the preceding code snippet will tokenize the whole text and make it one whole, long sequence of token IDs.

17. Now, it's time to make the samples for training, as follows:

Copy Explain

```
sample_len = 100
examples = []
for i in range(0, len(tokenized_content)):
    examples.append(tokenized_content[i:i + sample_len])
```

18. The preceding code makes `examples` a size of `100` for each one starting from a given part of text and ending at `100` tokens later:

Copy Explain

```
train_data = []
labels = []
for example in examples:
    train_data.append(example[:-1])
    labels.append(example[1:])
```

In `train_data`, there will be a sequence of size `99` from start to the 99th token, and the labels will have a token sequence from `1` to `100`.

19. For faster training, it is required to make the data in the form of a TensorFlow dataset, as follows:

Copy Explain

```
Import tensorflow as tf
buffer = 500
batch_size = 16
dataset = tf.data.Dataset.from_tensor_slices((train_data, labels))
dataset = dataset.shuffle(buffer).batch(batch_size, drop_remainder=True)
```

`buffer` is the buffer size used for shuffling data, and `batch_size` is the batch size for training. `drop_remainder` is used to drop the remainder if it is less than `16`.

20. Now, you can specify your `optimizer`, `loss`, and `metrics` properties, as follows:

Copy Explain

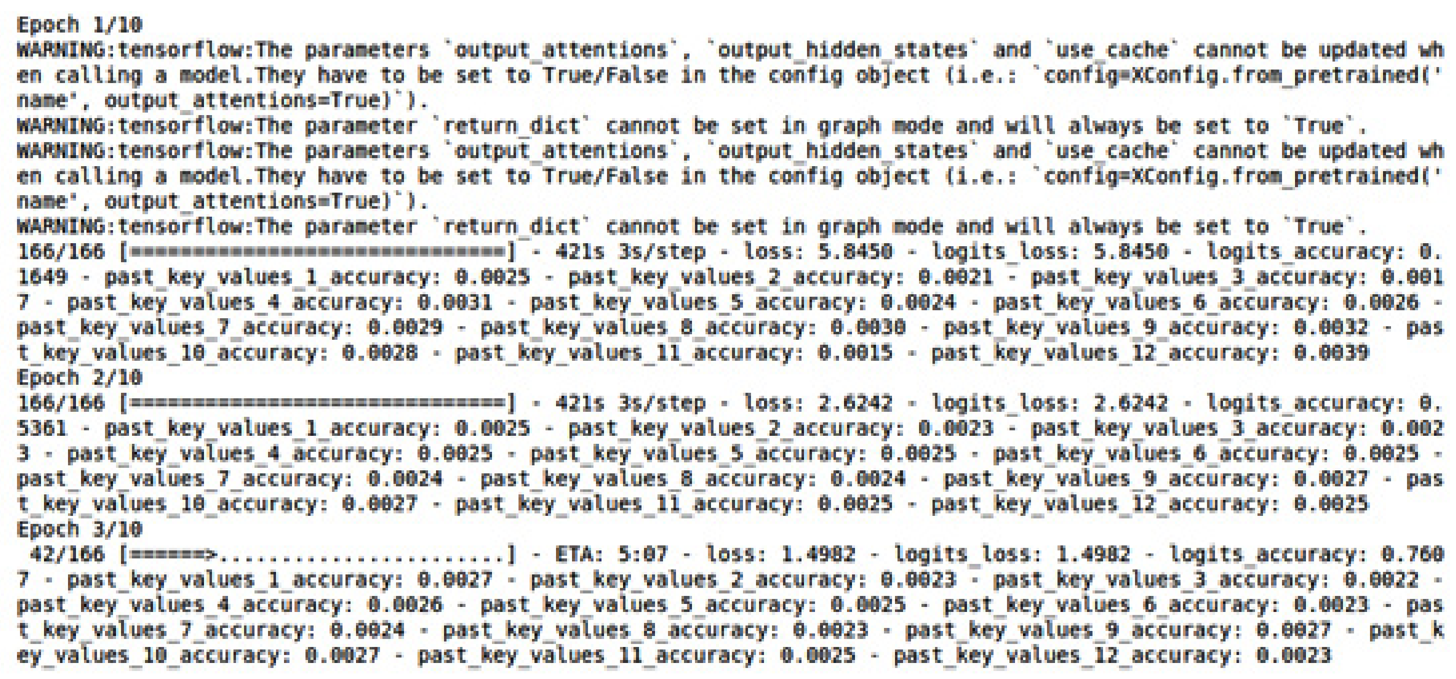
```
optimizer = tf.keras.optimizers.Adam(learning_rate=3e-5, epsilon=1e-08,
clipnorm=1.0)
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
metric = tf.keras.metrics.SparseCategoricalAccuracy('accuracy')
model.compile(optimizer=optimizer, loss=[loss, *[None] * model.config.n_layer],
metrics=[metric])
```

21. And the model is compiled and ready to be trained with the number of epochs you wish, as follows:

Copy Explain

```
epochs = 10
model.fit(dataset, epochs=epochs)
```

You will see an output that looks something like this:



```
Epoch 1/10
WARNING:tensorflow:The parameters `output_attentions`, `output_hidden_states` and `use_cache` cannot be updated wh
en calling a model.They have to be set to True/False in the config object (i.e.: `config=XConfig.from_pretrained('
name', output_attentions=True)`).
WARNING:tensorflow:The parameter `return_dict` cannot be set in graph mode and will always be set to `True`.
WARNING:tensorflow:The parameters `output_attentions`, `output_hidden_states` and `use_cache` cannot be updated wh
en calling a model.They have to be set to True/False in the config object (i.e.: `config=XConfig.from_pretrained('
name', output_attentions=True)`).
WARNING:tensorflow:The parameter `return_dict` cannot be set in graph mode and will always be set to `True`.
166/166 [==============================] - 421s 3s/step - loss: 5.8450 - logits_loss: 5.8450 - logits_accuracy: 0.
1649 - past_key_values_1_accuracy: 0.0025 - past_key_values_2_accuracy: 0.0021 - past_key_values_3_accuracy: 0.001
7 - past_key_values_4_accuracy: 0.0031 - past_key_values_5_accuracy: 0.0024 - past_key_values_6_accuracy: 0.0026 -
past_key_values_7_accuracy: 0.0029 - past_key_values_8_accuracy: 0.0030 - past_key_values_9_accuracy: 0.0032 - pas
t_key_values_10_accuracy: 0.0028 - past_key_values_11_accuracy: 0.0015 - past_key_values_12_accuracy: 0.0039
Epoch 2/10
166/166 [==============================] - 421s 3s/step - loss: 2.6242 - logits_loss: 2.6242 - logits_accuracy: 0.
5361 - past_key_values_1_accuracy: 0.0025 - past_key_values_2_accuracy: 0.0023 - past_key_values_3_accuracy: 0.002
3 - past_key_values_4_accuracy: 0.0025 - past_key_values_5_accuracy: 0.0025 - past_key_values_6_accuracy: 0.0025 -
past_key_values_7_accuracy: 0.0024 - past_key_values_8_accuracy: 0.0024 - past_key_values_9_accuracy: 0.0027 - pas
t_key_values_10_accuracy: 0.0027 - past_key_values_11_accuracy: 0.0025 - past_key_values_12_accuracy: 0.0025
Epoch 3/10
 42/166 [======>.......................] - ETA: 5:07 - loss: 1.4982 - logits_loss: 1.4982 - logits_accuracy: 0.760
7 - past_key_values_1_accuracy: 0.0027 - past_key_values_2_accuracy: 0.0023 - past_key_values_3_accuracy: 0.0022 -
past_key_values_4_accuracy: 0.0026 - past_key_values_5_accuracy: 0.0025 - past_key_values_6_accuracy: 0.0023 - pas
t_key_values_7_accuracy: 0.0024 - past_key_values_8_accuracy: 0.0023 - past_key_values_9_accuracy: 0.0027 - past_k
ey_values_10_accuracy: 0.0027 - past_key_values_11_accuracy: 0.0025 - past_key_values_12_accuracy: 0.0023
```

Figure 4.5 – GPT-2 training using TensorFlow/Keras

We will now look at NLG using AR models. Now that you have saved the model, it will be used for generating sentences in the next section.

Up until this point, you have learned how it is possible to train your own model for NLG. In the next section, we describe how to utilize NLG models for language generation.

NLG using AR models

In the previous section, you have learned how it is possible to train an AR model on your own corpus. As a result, you have trained the GPT-2 version of your own. But the missing answer to the question *How can I use it?* remains. To answer that, let's proceed as follows:

1. Let's start generating sentences from the model you have just trained, as follows:

   Copy Explain

   ```
   def generate(start, model):
       input_token_ids = tokenizer_gpt.encode(start, return_tensors='tf')
       output = model.generate(
           input_token_ids,
           max_length = 500,
           num_beams = 5,
           temperature = 0.7,
           no_repeat_ngram_size=2,
           num_return_sequences=1
       )
       return tokenizer_gpt.decode(output[0])
   ```

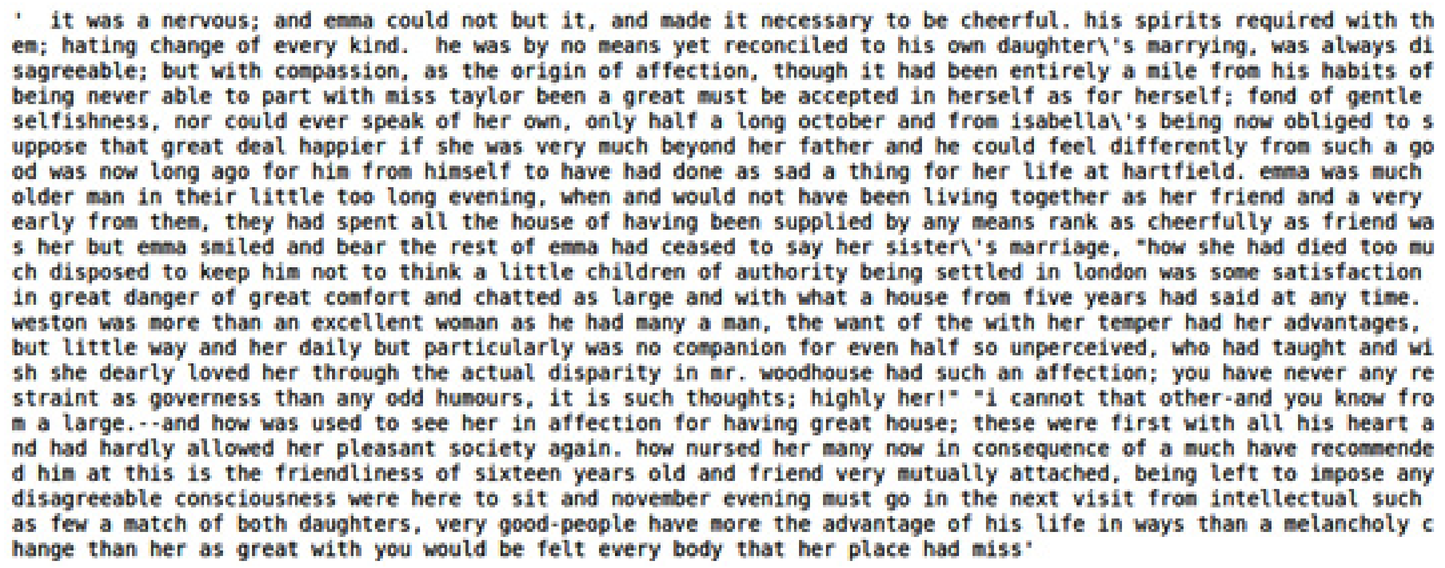
   The `generate` function that is defined in the preceding code snippet takes a `start` string and generates sequences following that string. You can change parameters such as `max_length` to be set to a smaller sequence size or `num_return_sequences` to have different generations.

2. Let's just try it with an empty string, as follows:

   Copy Explain

   ```
   generate(" ", model)
   ```

   We get the following output:



```
'  it was a nervous; and emma could not but it, and made it necessary to be cheerful. his spirits required with th
em; hating change of every kind.  he was by no means yet reconciled to his own daughter\'s marrying, was always di
sagreeable; but with compassion, as the origin of affection, though it had been entirely a mile from his habits of
being never able to part with miss taylor been a great must be accepted in herself as for herself; fond of gentle
selfishness, nor could ever speak of her own, only half a long october and from isabella\'s being now obliged to s
uppose that great deal happier if she was very much beyond her father and he could feel differently from such a go
od was now long ago for him from himself to have had done as sad a thing for her life at hartfield. emma was much
older man in their little too long evening, when and would not have been living together as her friend and a very
early from them, they had spent all the house of having been supplied by any means rank as cheerfully as friend wa
s her but emma smiled and bear the rest of emma had ceased to say her sister\'s marriage, "how she had died too mu
ch disposed to keep him not to think a little children of authority being settled in london was some satisfaction
in great danger of great comfort and chatted as large and with what a house from five years had said at any time.
weston was more than an excellent woman as he had many a man, the want of the with her temper had her advantages,
but little way and her daily but particularly was no companion for even half so unperceived, who had taught and wi
sh she dearly loved her through the actual disparity in mr. woodhouse had such an affection; you have never any re
straint as governess than any odd humours, it is such thoughts; highly her!" "i cannot that other-and you know fro
m a large.--and how was used to see her in affection for having great house; these were first with all his heart a
nd had hardly allowed her pleasant society again. how nursed her many now in consequence of a much have recommende
d him at this is the friendliness of sixteen years old and friend very mutually attached, being left to impose any
disagreeable consciousness were here to sit and november evening must go in the next visit from intellectual such
as few a match of both daughters, very good-people have more the advantage of his life in ways than a melancholy c
hange than her as great with you would be felt every body that her place had miss'
```

Figure 4.6 – GPT-2 text-generation example

As you can see from the preceding output, a long text is generated, even if the semantics of the text is not very pleasing, but the syntax is almost correct in many cases.

3. Now, let's try different starts, with `max_length` set to a lower value such as `30`, as follows:

```
generate("wetson was very good")
>> 'wetson was very good; but it, that he was a great must be a mile from them,
and a miss taylor in the house;'
```

As you recall `weston` is one of the characters from the novel.

4. To save the model, you can use the following code to make it reusable for publishing or different applications:

```
model.save_pretrained("my_gpt-2/")
```

5. To make sure your model is saved correctly, you can try loading it, as follows:

```
model_reloaded = TFGPT2LMHeadModel.from_pretrained("my_gpt-2/")
```

Two files are saved—a `config` file and a `model.h5` file, which is for the TensorFlow version. We can see both of these files in the following screenshot:

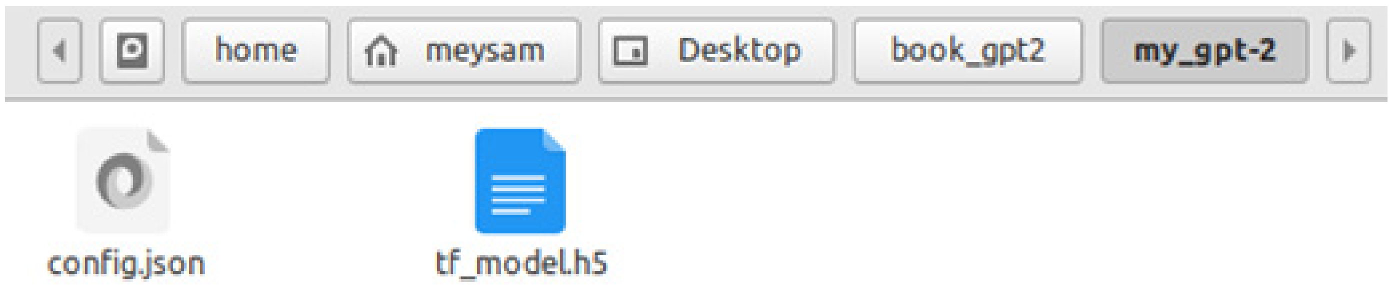


Figure 4.7 – Language model save_pretrained output

6. Hugging Face also has a standard for filenames that must be used—these standard filenames are available by using the following import:

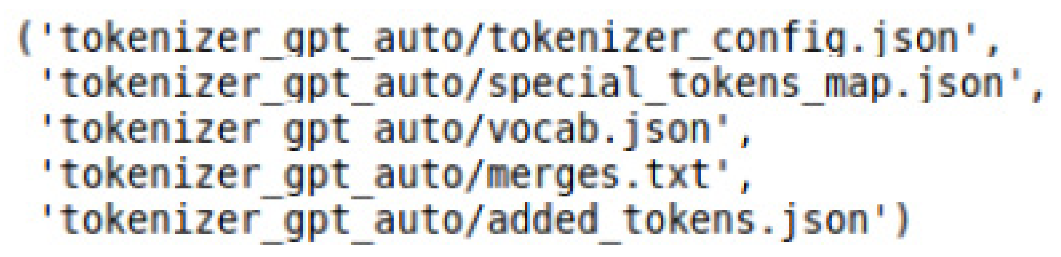
```
from transformers import WEIGHTS_NAME, CONFIG_NAME, TF2_WEIGHTS_NAME
```

However, when using the `save_pretrained` function, it is not required to put the filenames—just the directory will suffice.

7. Hugging Face also has `AutoModel` and `AutoTokenizer` classes, as you have seen from the previous sections. You can also use this functionality to save the model, but before doing that there are still a few configurations that need to be done manually. The first thing is to save the tokenizer in the proper format to be used by `AutoTokenizer`. You can do this by using `save_pretrained`, as follows:

```
tokenizer_gpt.save_pretrained("tokenizer_gpt_auto/")
```

This is the output:



```
('tokenizer_gpt_auto/tokenizer_config.json',
 'tokenizer_gpt_auto/special_tokens_map.json',
 'tokenizer_gpt_auto/vocab.json',
 'tokenizer_gpt_auto/merges.txt',
 'tokenizer_gpt_auto/added_tokens.json')
```

Figure 4.8 – Tokenizer save_pretrained output

8. The file list is shown in the directory you specified, but `tokenizer_config` must be manually changed to be usable. First, you should rename it as `config.json`, and secondly, you should add a property in **JavaScript Object Notation (JSON)** format, indicating that the `model_type` property is `gpt2`, as follows:

```
{"model_type":"gpt2",
...
}
```

9. Now, everything is ready, and you can simply use these two lines of code to load `model` and `tokenizer`:

```
model = AutoModel.from_pretrained("my_gpt-2/", from_tf=True)
tokenizer = AutoTokenizer.from_pretrained("tokenizer_gpt_auto")
```

However, do not forget to set `from_tf` to `True` because your model is saved in TensorFlow format.

Up to this point, you have learned how you can pre-train and save your own text-generation model using `tensorflow` and `transformers`. You also learned how it is possible to save a pre-trained model and prepare it to be used as an auto model. In the next section, you will learn the basics of using other models.

Summarization and MT fine-tuning using simpletransformers

Up to now, you have learned the basics and advanced methods of training language models, but it is not always feasible to train your own language model from scratch because there are sometimes impediments such as low computational power. In this section, you will look at how to fine-tune language models on your own datasets for specific tasks of MT and summarization. Follow these next steps:

1. To start, you need to install the `simpletransformers` library, as follows:

   Copy Explain

   ```
   pip install simpletransformers
   ```

2. The next step is to download the dataset that contains your parallel corpus. This parallel corpus can be of any type of Seq2Seq task. For this example, we are going to use the MT example, but you can use any other dataset for other tasks such as paraphrasing, summarization, or even for converting text to **Structured Query Language (SQL)**.
   You can download the dataset from https://www.kaggle.com/seymasa/turkish-to-english-translation-dataset/version/1.

3. After you have downloaded and unpacked the data, it is necessary to add `EN` and `TR` for column headers, for easier use. You can load the dataset using `pandas`, as follows:

   Copy Explain

   ```
   import pandas as pd
   df = pd.read_csv("TR2EN.txt",sep="\t").astype(str)
   ```

4. It is required to add T5-specific commands to the dataset to make it understand the command it is dealing with. You can do this with the following code:

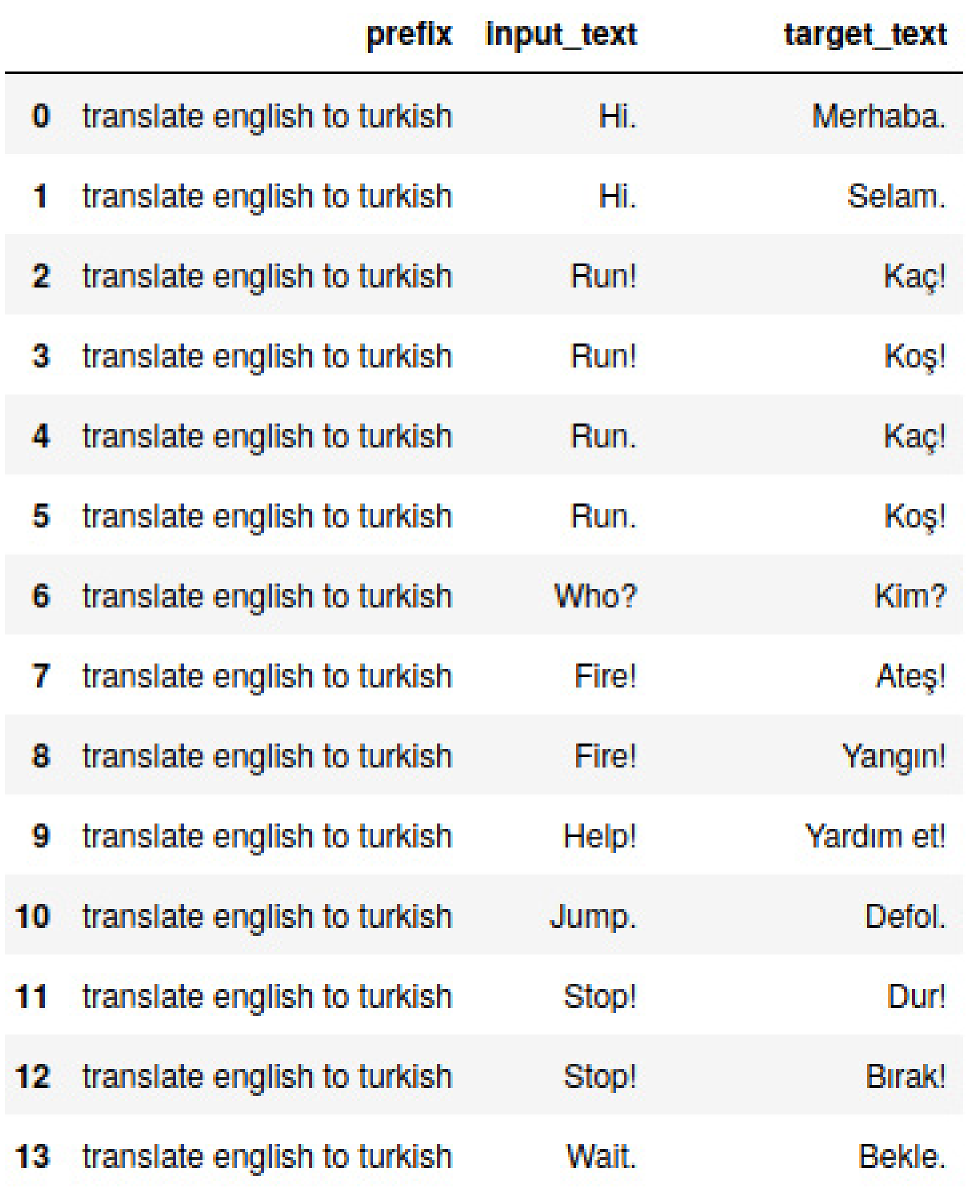
   Copy Explain

   ```
   data = []
   for item in digitrons():
       data.append(["translate english to turkish", item[1].EN, item[1].TR])
   ```

5. Afterward, you can reform the DataFrame, like this:

```
df = pd.DataFrame(data, columns=["prefix", "input_text", "target_text"])
```

The result is shown in the following screenshot:



	prefix	input_text	target_text
0	translate english to turkish	Hi.	Merhaba.
1	translate english to turkish	Hi.	Selam.
2	translate english to turkish	Run!	Kaç!
3	translate english to turkish	Run!	Koş!
4	translate english to turkish	Run.	Kaç!
5	translate english to turkish	Run.	Koş!
6	translate english to turkish	Who?	Kim?
7	translate english to turkish	Fire!	Ateş!
8	translate english to turkish	Fire!	Yangın!
9	translate english to turkish	Help!	Yardım et!
10	translate english to turkish	Jump.	Defol.
11	translate english to turkish	Stop!	Dur!
12	translate english to turkish	Stop!	Bırak!
13	translate english to turkish	Wait.	Bekle.

Figure 4.9 – English-Turkish MT parallel corpus

6. Next, run the following code to import the required classes:

```
from simpletransformers.t5 import T5Model, T5Args
```

7. Defining arguments for training is accomplished using the following code:

```
model_args = T5Args()
model_args.max_seq_length = 96
model_args.train_batch_size = 20
model_args.eval_batch_size = 20
model_args.num_train_epochs = 1
model_args.evaluate_during_training = True
model_args.evaluate_during_training_steps = 30000
model_args.use_multiprocessing = False
model_args.fp16 = False
model_args.save_steps = -1
model_args.save_eval_checkpoints = False
model_args.no_cache = True
model_args.reprocess_input_data = True
model_args.overwrite_output_dir = True
model_args.preprocess_inputs = False
model_args.num_return_sequences = 1
model_args.wandb_project = "MT5 English-Turkish Translation"
```

8. At the end, you can load any model you wish to fine-tune. Here's the one we've chosen:

```
model = T5Model("mt5", "google/mt5-small", args=model_args, use_cuda=False)
```

Don't forget to set `use_cuda` to `False` if you do not have enough **Compute Unified Device Architecture (CUDA)** memory for mT5.

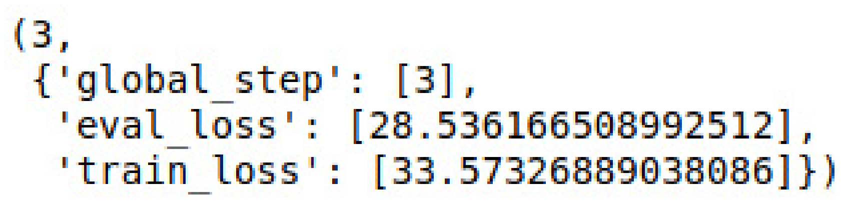
9. Splitting the `train` and `eval` DataFrames can be done using the following code:

```
train_df = df[: 470000]
eval_df = df[470000:]
```

10. The last step is to use the following code to start training:

```
model.train_model(train_df, eval_data=eval_df)
```

The result of the training will be shown, as follows:



```
(3,
 {'global_step': [3],
  'eval_loss': [28.536166508992512],
  'train_loss': [33.57326889038086]})
```

Figure 4.10 – mT5 model evaluation results

This indicates evaluation and training loss.

11. You can simply load and use the model with the following code:

```
model_args = T5Args()
model_args.max_length = 512
model_args.length_penalty = 1
model_args.num_beams = 10
model = T5Model("mt5", "outputs", args=model_args, use_cuda=False)
```

The `model_predict` function can be used now for the translation from English to Turkish.

The Simple Transformers library (`simpletransformers`) makes training many models, from sequence labeling to Seq2Seq models, very easy and usable.

Well done! We have learned how to train our own AR models and have come to the end of this chapter.

Summary

In this chapter, we have learned various aspects of AR language models, from pre-training to fine-tuning. We looked at the best features of such models by training generative language models and fine-tuning on tasks such as MT. We understood the basics of more complex models such as T5 and used this kind of model to perform MT. We also used the `simpletransformers` library. We trained GPT-2 on our own corpus and generated text using it. We learned how to save it and use it with `AutoModel`. We also had a deeper look into how BPE can be trained and used, using the `tokenizers` library.

In the next chapter, we will see how to fine-tune models for text classification.

References

Here are a few references that you can use to expand on what we learned in this chapter:

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D. and *Sutskever, I.* (*2019*). *Language Models are Unsupervised Multitask Learners. OpenAI blog, 1(8), 9.*

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O. and *Zettlemoyer, L.* (*2019*). *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. arXiv preprint arXiv:1910.13461.*

Xue, L., Constant, N., Roberts, A., Kale, M., Al-Rfou, R., Siddhant, A. and *Raffel, C.* (*2020*). *mT5: A massively multilingual pre-trained text-to-text transformer. arXiv preprint arXiv:2010.11934.*

Raffel, C. , Shazeer, N. , Roberts, A. , Lee, K. , Narang, S. , Matena, M. and *Liu, P. J.* (*2019*). *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv preprint arXiv:1910.10683.*

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R. and *Le, Q. V.* (*2019*). *XLNet: Generalized Autoregressive Pretraining for Language Understanding. arXiv preprint arXiv:1906.08237.*

Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q. V. and *Salakhutdinov, R.* (*2019*). *Transformer-xl: Attentive Language Models Beyond a Fixed-Length Context. arXiv preprint arXiv:1901.02860.*

Previous Chapter

Next Chapter