Join our book community on Discord


Sometimes, a pretrained model will not provide the results you expect. Even if the pretrained model goes through additional training through fine-tuning, it still will not work as planned. At that point, one approach is to initiate pretraining from scratch through platforms like Hugging Face to leverage architectures such as GPT and BERT, among others. Once you have pretrained a model from scratch, you will know how to train other models you might need for a project.In this chapter, we will build a RoBERTa, an advanced variant of BERT, model from scratch. The model will use the bricks of the transformer construction kit we need for BERT models. Also, no pretrained tokenizers or models will be used. The RoBERTa model will be built following the fifteen-step process described in this chapter.We will use the knowledge of transformers acquired in the previous chapters to build a model that can perform language modeling on masked tokens step by step. In *Chapter 2*, *Getting Started with the Architecture of the Transformer Model*, we went through the building blocks of the original Transformer. In *Chapter 5*, *Diving into Fine-Tuning through BERT*, we explored how to fine-tune a pretrained model through a variation of a BERT model.This chapter will focus on building a pretrained transformer model from scratch using a Jupyter notebook based on Hugging Face's seamless modules. The model is named KantaiBERT. It's my project, named after Immanuel Kant, the philosopher. When you train and deploy a model in your environment, you can give it a "brand" to make your product unique.KantaiBERT first loads a compilation of Immanuel Kant's books created for this chapter. You will see how the data was obtained. You will also see how to create your own datasets for this notebook.KantaiBERT trains its own tokenizer from scratch. First, it will build its merge

and vocabulary files, which will be used during the pretraining process.KantaiBERT then processes the dataset, initializes a trainer, and trains the model.We will then run the trained model KantaiBERT to perform an experimental downstream language modeling task and fill a mask using Immanuel Kant's logic.Finally, we will put the knowledge you acquired in this chapter to work and pretrain a Generative AI Customer Support Model on X data (former Twitter). We will use a Kaggle dataset of support tweets of twenty major brands. You will discover how to build an open-source Generative AI chat agent prototype with a RoBERTa model. This opens the door to independent chat agents when necessary.By the end of the chapter, you will know how to build a transformer model from scratch. In addition, you will have enough knowledge of transformers to face a situation where you must train a model to match a specific need.This chapter covers the following topics:

RoBERTa- and DistilBERT-like models

Byte-level byte-pair encoding

Training a tokenizer

Defining the configuration of the model

Initializing a model from scratch

Exploring the parameters of a model

Building a dataset

Defining a data collator

Initializing a trainer

Pretraining the model

Saving the final model (+tokenizer + config) to disk

Language modeling with FillMaskPipeline

Applying the model to the downstream tasks of **Masked Language Modeling (MLM)**

Pretraining a Generative AI Chat agent based on an X(former Twitter) dataset

Our initial step involves outlining the structure of the transformer architecture we intend to build and the tasks we aim to accomplish.

# Training a tokenizer and pretraining a transformer

In this chapter, we will train a transformer model named KantaiBERT using the building blocks provided by Hugging Face for BERT-like models. We covered the theory of the building blocks of the model we will use in *Chapter 5*, *Diving into Fine-Tuning through BERT*.We will describe KantaiBERT, building on the knowledge we acquired in previous chapters.KantaiBERT is a **Robustly Optimized BERT Pretraining Approach** (**RoBERTa**)-like model. RoBERTa is an advanced version of BERT designed by Meta(former Facebook). The initial BERT models brought innovative features to the initial transformer models, as we saw in *Chapter 5, Diving into Fine-Tuning through BERT*. RoBERTa increases the performance of transformers for downstream tasks by improving the mechanics of the pretraining process.For example, it does not use `WordPiece` tokenization but goes down to byte-level **Byte-Pair Encoding** (**BPE**). This method paved the way for a wide variety of BERT and BERT-like models.In this chapter, KantaiBERT, like BERT, will be trained using **Masked Language Modeling** (**MLM**). MLM is a language modeling technique that masks a word in a sequence. Then, the transformer model must train to predict the masked word.KantaiBERT will be trained as a small model with 6 layers, 12 heads, and `83,504,416` parameters (this number might vary when the model is updated). It might seem that 83 million parameters are a lot. However, the parameters are spread over 12 heads, which makes it a relatively small model. A small model will make the pretraining experience smooth so that each step can be viewed in real-time without waiting for hours to see a result.KantaiBERT is a smaller version of RoBERTa using a DistilBERT(a distilled version of BERT) architecture with 6 layers and 12 heads. As such, KantaiBERT runs much faster, but the results are slightly less accurate than with the full configuration of a RoBERTa model.We know that large models achieve excellent performance. But what if you want to run a model on a smartphone? Miniaturization has been the key to technological evolution. Transformers will sometimes have to follow the same path during implementation. Distillation using fewer parameters or other such methods in the future is a clever way of taking the best of pretraining and making it efficient for the needs of many downstream tasks.It is essential to show all the possible architectures, including running a small model on a smartphone. However, the future of transformers will also be ready-to-use APIs, as we will see in *Chapter 7, From Basics to Disruption with ChatGPT*.KantaiBERT will implement a byte-level byte-pair

encoding tokenizer like the one used by GPT-2. The special tokens will be the ones used by RoBERTa. BERT models most often use a WordPiece tokenizer.There are no token type IDs to indicate which part of a segment a token is a part of. Instead, the segments will be separated with the separation token `</s>`.KantaiBERT will use a custom dataset, train a tokenizer, train the transformer model, save it, and run it with an MLM example.Let's get going and build a transformer from scratch.

---

# Building KantaiBERT from scratch

We will build KantaiBERT in 15 steps from scratch and run it on an MLM example.Open Google Colaboratory (you need a Gmail account). Then upload `KantaiBERT.ipynb`, which is on GitHub in this chapter's directory.The titles of the 15 steps of this section are similar to the titles of the notebook cells, which makes them easy to follow.Let's start by loading the dataset.

## Step 1: Loading the dataset

Ready-to-use datasets provide an objective way to train and compare transformers. This chapter aims to understand the training process of a transformer with notebook cells that can be run in real-time without waiting for hours to obtain a result.I chose to use the works of Immanuel Kant (1724-1804), the German philosopher who was the epitome of the *Age of Enlightenment*. The idea is to introduce human-like logic and pretrained reasoning for downstream reasoning tasks.Project Gutenberg, https://www.gutenberg.org, offers a wide range of free eBooks that can be downloaded in text format. You can use other books if you want to create customized datasets of your own based on books.I compiled the following three books by Immanuel Kant into a text file named `kant.txt`:*The Critique of Pure ReasonThe Critique of Practical ReasonFundamental Principles of the Metaphysic of Morals*`kant.txt` provides a small training dataset to train the transformer model of this chapter. The result obtained remains experimental. For a real-life project, I would add the complete works of Immanuel Kant, Rene Descartes, Pascal, and Leibnitz.The text file contains the raw text of the books:

Copy Explain

```
…For it is in reality vain to profess _indifference_ in regard to such inquiries, the
object of which cannot be indifferent to humanity.
```

The dataset is downloaded automatically from GitHub in the first cell of the `KantaiBERT.ipynb` notebook.You can also load `kant.txt`, which is in the directory of this chapter on GitHub, using Colab's file manager. In this case, `curl` is used to retrieve it from GitHub:

```
#1.Load kant.txt using the Colab file manager
#2.Downloading the file from GitHub
!curl -L https://raw.githubusercontent.com/Denis2054/ Transformers-for-NLP-and-
Computer-Vision-3rd-Edition/master/Chapter06/kant.txt --output "kant.txt"
```

You can see it appear in the Colab file manager pane once you have loaded or downloaded it:

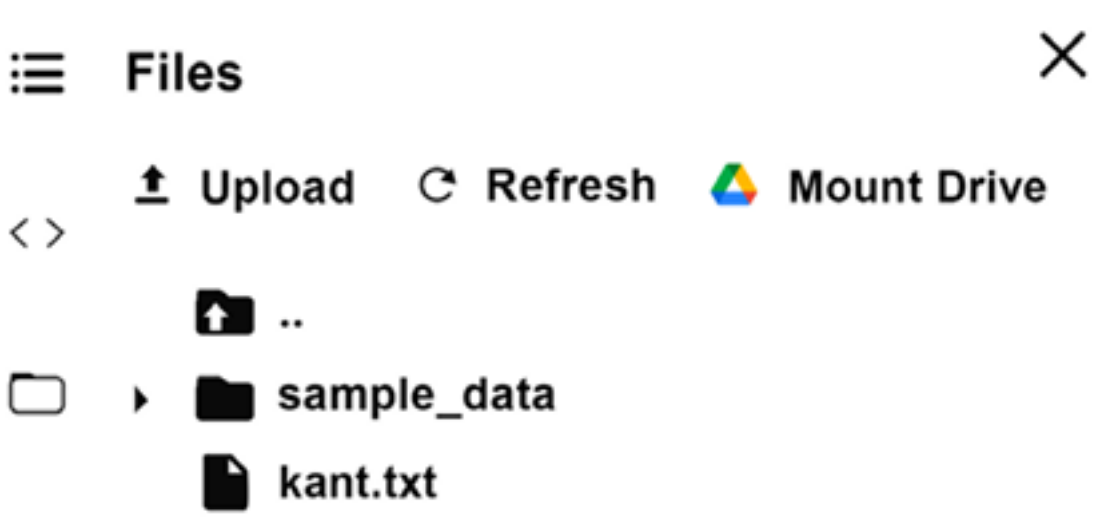


Figure 6.1: Colab file manager

Note that Google Colab deletes the files when you restart the VM.The dataset is defined and loaded.

Do not run the subsequent cells without `kant.txt` . Training data is a prerequisite. Now, the program will install the Hugging Face transformers.

## Step 2: Installing Hugging Face transformers

We will need to install Hugging Face transformers and tokenizers. We will also install an accelerator as recommended by Hugging Face and summed up in Step 2 of the notebook:

```
April 2023 update From Hugging Face Issue 22816:
https://github.com/huggingface/transformers/issues/22816
"The PartialState import was added as a dependency on the transformers development
branch yesterday. PartialState was added in the 0.17.0 release in accelerate, and so
for the development branch of transformers, accelerate >= 0.17.0 is required.
Downgrading the transformers version removes the code which is importing PartialState."
Denis Rothman: The following cell imports the latest version of Hugging Face
transformers but without downgrading it.
To adapt to the Hugging Face upgrade, A GPU accelerator was activated using the Google
Colab Pro with the following NVIDIA GPU: GPU Name: NVIDIA A100-SXM4-40GB
```

If the Accelerate module is not installed, an error message will appear before the training process can begin:

```
ImportError:Using the `Trainer` with `PyTorch` requires `accelerate`: Run `pip install --upgrade accelerate`
```

When installed, the module provides an abstraction layer that optimizes hardware accelerators (GPUs, TPUs). The Accelerate module is practical when implementing large transformer models such as GPT and BERT.The program installs Hugging Face Transformers and

```
!pip install Transformers
!pip install --upgrade accelerate
from accelerate import Accelerator
```

The program will now begin by training a tokenizer.

## Step 3: Training a tokenizer

In this section, the program does not use a pretrained tokenizer. For example, a pretrained GPT-2 tokenizer could be used. However, the training process in this chapter includes training a tokenizer from scratch.Hugging Face's ByteLevelBPETokenizer, when trained using kant.txt (or any other text), utilizes a **Byte-Pair Encoding** (**BPE**) tokenizer. A BPE tokenizer breaks down strings or words into subword units or substrings. This approach offers several advantages, including the following:- The tokenizer can break words into minimal components and then merge these components into statistically significant ones. For example, words like "smaller" and "smallest" can be represented as "small," "er," and "est." Furthermore, the tokenizer can go even further, generating subword parts like "sm" and "all." In essence, words are broken down into subword tokens and smaller units, such as "sm" and "all," rather than being represented as a single token like "small."- The chunks of text classified as unknown, typically represented as "unk_token," using WordPiece-level encoding, can effectively be minimized or eliminated.In this model, we will be training the tokenizer with the following parameters:

`files=paths` is the path to the dataset
`vocab_size=52_000` is the size of our tokenizer's model length`min_frequency=2` is the minimum frequency threshold`special_tokens=[]` is a list of special tokensIn this case, the list of special tokens is:

<s> : a start token
<pad>: a padding token</s>: an end token<unk>: an unknown token<mask>: the mask token for language modelingThe tokenizer will be trained to generate merged substring tokens and analyze their frequency.Let's take these two words in the middle of a sentence:

```
...the tokenizer...
```

Copy Explain

The first step will be to tokenize the string:

```
'Ġthe', 'Ġtoken',   'izer',
```

Copy Explain

The string is now tokenized into tokens with Ġ , which represents a whitespace(a space between words).The next step is to replace them with their indices:

| 'Ġthe' | 'Ġtoken' | 'izer' |
|---|---|---|
| 150 | 5430 | 4712 |

Table 6.1: Indices for the three tokens
The program runs the tokenizer as expected:

Copy Explain

```
from pathlib import Path
from tokenizers import ByteLevelBPETokenizer
paths = [str(x) for x in Path(".").glob("**/*.txt")]
# Read the content from the files, ignoring or replacing invalid characters
file_contents = []
for path in paths:
    try:
        with open(path, 'r', encoding='utf-8', errors='replace') as file:
            file_contents.append(file.read())
    except Exception as e:
        print(f"Error reading {path}: {e}")
# Join the contents into a single string
text = "\n".join(file_contents)
# Initialize a tokenizer
tokenizer = ByteLevelBPETokenizer()
# Customize training
tokenizer.train_from_iterator([text], vocab_size=52_000, min_frequency=2,
special_tokens=[
    "<s>",
    "<pad>",
    "</s>",
    "<unk>",
    "<mask>",
])
```

The tokenizer will skip the erroneous characters in the file and train the tokenizer. The tokenizer is trained and ready to be saved.

## Step 4: Saving the files to disk

The tokenizer will generate two files when trained:

`merges.txt` , which contains the merged tokenized substrings

`vocab.json`, which contains the indices of the tokenized substringsThe program first creates the `KantaiBERT` directory and then saves the two files:

Copy Explain

```
import os
token_dir = '/content/KantaiBERT'
if not os.path.exists(token_dir):
  os.makedirs(token_dir)
tokenizer.save_model('KantaiBERT')
```

The program output shows that the two files have been saved:

```
['KantaiBERT/vocab.json', 'KantaiBERT/merges.txt']
```

The two files should appear in the file manager pane:

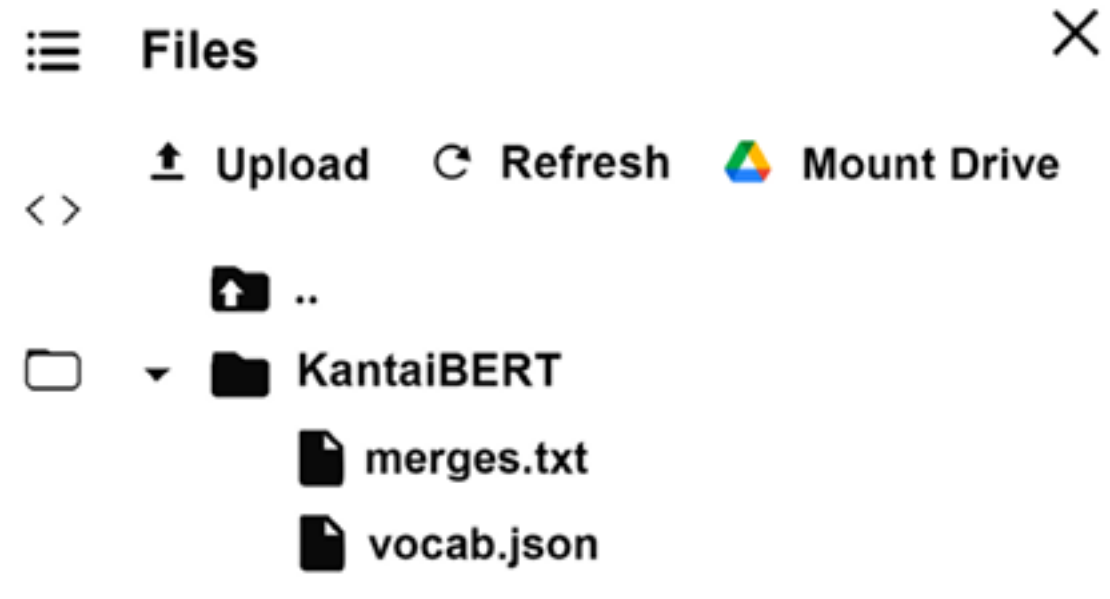


Figure 6.2: Colab file manager

The files in this example are small. You can double-click on them to view their content. `merges.txt` contains the tokenized substrings as planned:

```
#version: 0.2 - Trained by 'huggingface/tokenizers'
Ġ t
h e
Ġ a
o n
i n
Ġ o
Ġt he
r e
i t
Ġo f
```

`vocab.json` contains the indices:

```
[…,"Ġthink":955,"preme":956,"ĠE":957,"Ġout":958,"Ġdut":959,"aly":960,"Ġexp":961,…]
```

The trained tokenized dataset files are ready to be processed.

## Step 5: Loading the trained tokenizer files

We could have loaded pretrained tokenizer files. However, we trained our own tokenizer and are now ready to load the files:

```
from tokenizers.implementations import ByteLevelBPETokenizer
from tokenizers.processors import BertProcessing
tokenizer = ByteLevelBPETokenizer(
    "./KantaiBERT/vocab.json",
    "./KantaiBERT/merges.txt",
)
```

The tokenizer can encode a sequence:

```
tokenizer.encode("The Critique of Pure Reason.").tokens
```

"The Critique of Pure Reason" will become:

```
['The', 'ĠCritique', 'Ġof', 'ĠPure', 'ĠReason', '.']
```

We can also ask to see the number of tokens in this sequence:

```
tokenizer.encode("The Critique of Pure Reason.")
```

The output will show that there are 6 tokens in the sequence:

```
Encoding(num_tokens=6, attributes=[ids, type_ids, tokens, offsets, attention_mask,
special_tokens_mask, overflowing])
```

The tokenizer now processes the tokens to fit the BERT model variant used in this notebook. The post-processor will add a start and end token; for example:

```
tokenizer._tokenizer.post_processor = BertProcessing(
    ("</s>", tokenizer.token_to_id("</s>")),
    ("<s>", tokenizer.token_to_id("<s>")),
)
tokenizer.enable_truncation(max_length=512)
```

Let's encode a post-processed sequence:

```
tokenizer.encode("The Critique of Pure Reason.")
```

The output shows that we now have 8 tokens:

```
Encoding(num_tokens=8, attributes=[ids, type_ids, tokens, offsets, attention_mask,
special_tokens_mask, overflowing])
```

If we want to see what was added, we can ask the tokenizer to encode the post-processed sequence by running the following cell:

```
tokenizer.encode("The Critique of Pure Reason.").tokens
```

The output shows that the start and end tokens have been added, which brings the number of tokens to 8, including start and end tokens:

```
['<s>', 'The', 'ĠCritique', 'Ġof', 'ĠPure', 'ĠReason', '.', '</s>']
```

The data for the training model is now ready to be trained. We will now check the system information of the machine on which we are running the notebook.

# Step 6: Checking resource constraints: GPU and CUDA

KantaiBERT runs at optimal speed with a **Graphics Processing Unit (GPU)**.We will first run a command to see if an NVIDIA GPU card is present:

```
!nvidia-smi
```

The output displays the information and version on the card depending on the machine you are working on. In this case, it was Tesla V100-SXM2:

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.85.12    Driver Version: 525.85.12    CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   34C    P0    24W / 300W |      0MiB / 16384MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
```

Figure 6.3: Information on the NVIDIA card

The output may vary with each Google Colab VM configuration.We will now check to make sure `PyTorch` sees CUDA:

```
import torch
torch.cuda.is_available()
```

The result should be `True`:

```
True
```

**Compute Unified Device Architecture (CUDA)** was developed by NVIDIA to use the parallel computing power of its GPUs.We are now ready to define the configuration of the model.

## Step 7: Defining the configuration of the model

We will be pretraining a RoBERTa-type transformer model using the same number of layers and heads as a DistilBERT transformer. The model will have a vocabulary size set to 52,000, 12 attention heads, and 6 layers:

```
from transformers import RobertaConfig
config = RobertaConfig(
    vocab_size=52_000,
    max_position_embeddings=514,
    num_attention_heads=12,
    num_hidden_layers=6,
    type_vocab_size=1,
)
```

We will explore the configuration in more detail in *Step 9: Initializing a model from scratch*.Let's first recreate the tokenizer in our model.

## Step 8: Reloading the tokenizer in transformers

We are now ready to load our trained tokenizer, which is our pretrained tokenizer in `RobertaTokenizer.from_pretained()`:

Copy Explain

```
from transformers import RobertaTokenizer
from transformers import RobertaTokenizer
tokenizer = RobertaTokenizer.from_pretrained("./KantaiBERT", max_length=512)
```

Now that we have loaded our trained tokenizer let's initialize a RoBERTa model from scratch.

## Step 9: Initializing a model from scratch

In this section, we will initialize a model from scratch and examine the size of the model.The program first imports a RoBERTa masked model for language modeling:

Copy Explain

```
from transformers import RobertaForMaskedLM
```

The model is initialized with the configuration defined in *Step 7*:

Copy Explain

```
model = RobertaForMaskedLM(config=config)
```

If we print the model, we can see that it is a BERT model with 6 layers and 12 heads:

Copy Explain

```
print(model)
```

The building blocks of the encoder of the original Transformer model are present with different dimensions, as shown in this excerpt of the output:

Copy Explain

```
RobertaForMaskedLM(
  (roberta): RobertaModel(
    (embeddings): RobertaEmbeddings(
      (word_embeddings): Embedding(52000, 768, padding_idx=1)
      (position_embeddings): Embedding(514, 768, padding_idx=1)
      (token_type_embeddings): Embedding(1, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
…/…
```

Take some time to review the details of the configuration output before continuing. You will get to know the model from the inside.The LEGO®-type building blocks of transformers make it fun to analyze. For example, you will note that dropout regularization is present throughout the sublayers as in classical neural networks.Now, let's explore the parameters.

### Exploring the parameters

Exploring the model's parameters can provide some additional insights into its architecture. The model is small and contains `83,504,416` parameters (the number might change with model updates).We can check its size:

```
print(model.num_parameters())
```

The output shows the approximate number of parameters, which might vary from one transformer version to another:

```
83504416
```

Let's now look into the parameters. We first store the parameters in LP and calculate the length of the list of parameters:

```
LP=list(model.parameters())
lp=len(LP)
print(lp)
```

The output shows that there are approximately 106 matrices and vectors, *which might vary from one transformer model(or update) to another:*

```
106
```

Now, let's display the 106 matrices and vectors in the tensors that contain them:

```
for p in range(0,lp):
  print(LP[p])
```

The output displays all the parameters as shown in the following excerpt of the output:

Copy Explain

```
Parameter containing:
tensor([[-0.0175, -0.0210, -0.0334,  ...,  0.0054, -0.0113,  0.0183],
        [ 0.0020, -0.0354, -0.0221,  ...,  0.0220, -0.0060, -0.0032],
        [ 0.0001, -0.0002,  0.0036,  ..., -0.0265, -0.0057, -0.0352],
        ...,
        [-0.0125, -0.0418,  0.0190,  ..., -0.0069,  0.0175, -0.0308],
        [ 0.0072, -0.0131,  0.0069,  ...,  0.0002, -0.0234,  0.0042],
        [ 0.0008,  0.0281,  0.0168,  ..., -0.0113, -0.0075,  0.0014]],
       requires_grad=True)
```

Take a few minutes to peek inside the parameters to understand how transformers are built.The number of parameters is calculated by taking all parameters in the model and adding them up; for example:The vocabulary (52,000) x dimensions (768)The size of the vectors is 1 x 768

The many other dimensions found

You will note that *d*model = 768. There are 12 heads in the model. The dimension of *d*k for each head will thus be

$$d_k = \frac{d_{model}}{12} = 64$$

.

This shows, once again, the optimized LEGO® concept of the building blocks of a transformer.We will now see how the number of parameters of a model is calculated and how the figure 83,504,416 is reached.The following cell displays the size of each of the 106 tensors in this model:

Copy Explain

```
#Shape of each tensor in the model
LP = list(model.parameters())
for i, tensor in enumerate(LP):
    print(f"Shape of tensor {i}: {tensor.shape}")
```

The output will display the tensor and its size:

Copy Explain

```
Shape of tensor 0: torch.Size([52000, 768])
Shape of tensor 1: torch.Size([514, 768])
Shape of tensor 2: torch.Size([1, 768])
Shape of tensor 3: torch.Size([768])
Shape of tensor 4: torch.Size([768])
Shape of tensor 5: torch.Size([768, 768])
Shape of tensor 6: torch.Size([768])
Shape of tensor 7: torch.Size([768, 768])
Shape of tensor 8: torch.Size([768])
Shape of tensor 9: torch.Size([768, 768])
Shape of tensor 10: torch.Size([768])
Shape of tensor 11: torch.Size([768, 768])
Shape of tensor 12: torch.Size([768])
Shape of tensor 13: torch.Size([768])
Shape of tensor 14: torch.Size([768])
Shape of tensor 15: torch.Size([3072, 768])
Shape of tensor 16: torch.Size([3072])…
```

Note that the numbers might vary depending on the version of the transformer module you use.We can tentatively describe what the tensors represent, for example:

- `torch.Size([52000, 768])`: probably the embedding matrix containing the 52,000 words of the dictionary and 768-dimensional embedding
- `torch.Size([514, 768])`: this could be the positional embeddings. RoBERTa accepts 512 tokens, to which we add the start and end tokens.
- `torch.Size([1, 768])`: this could be the embedding of a special token such as `<cls>`

You can see that a transformer model is the mathematical representation of sequences it will learn.We will take this further and count the number of parameters of each tensor. First, the program initializes a parameter counter named `np` (number of parameters) and goes through the `lp` (`106`) number of elements in the list of parameters:

Copy Explain

```
#counting the parameters
np=0
for p in range(0,lp):#number of tensors
```

The parameters are matrices and vectors of different sizes; for example:768 x 768

768 x 1

768

We can see that some parameters are two-dimensional, and some are one-dimensional.An easy way to see if a parameter `p` in the list `LP[p]` has two dimensions or not is by doing the following:

```
PL2=True
try:
  L2=len(LP[p][0]) #check if 2D
except:
  L2=1             #not 2D but 1D
  PL2=False
```

If the parameter has two dimensions, its second dimension will be `L2>0` and `PL2=True (2 dimensions=True)`. If the parameter has only one dimension, its second dimension will be `L2=1` and `PL2=False (2 dimensions=False)`.`L1` is the size of the first dimension of the parameter. `L3` is the size of the parameters defined by:

```
L1=len(LP[p])
L3=L1*L2
```

We can now add the parameters at each step of the loop:

```
np+=L3              # number of parameters per tensor
```

We will obtain the sum of the parameters, but we also want to see exactly how the number of parameters of a transformer model is calculated:

Copy Explain

```
  if PL2==True:
    print(p,L1,L2,L3)  # displaying the sizes of the parameters
  if PL2==False:
    print(p,L1,L3)  # displaying the sizes of the parameters
print(np)              # total number of parameters
```

Note that if a parameter only has one dimension, `PL2=False`, then we only display the first dimension.The output is the list of how the number of parameters was calculated for all the tensors in the model, as shown in the following excerpt:

Copy Explain

```
0 52000 768 39936000
1 514 768 394752
2 1 768 768
3 768 768
4 768 768
5 768 768 589824
6 768 768
7 768 768 589824
8 768 768
9 768 768 589824
10 768 768
11 768 768 589824
12 768 768
13 768 768
14 768 768
15 3072 768 2359296
16 3072 3072
17 768 3072 2359296
18 768 768
19 768 768
20 768 768
21 768 768 589824
22 768 768
23 768 768 589824
24 768 768
25 768 768 589824
…
94 768 768
95 3072 768 2359296
96 3072 3072
97 768 3072 2359296
98 768 768
99 768 768
100 768 768
101 52000 52000
102 768 768 589824
103 768 768
104 768 768
105 768 768
83504416
```

**The total number of parameters of the RoBERTa model is displayed at the end of the list:**

Copy Explain

```
83504416
```

*The number of parameters might vary with the version of the libraries used.*We now know precisely what the number of parameters represents in a transformer model. Take a few minutes to go back and look at the output of the configuration, the parameters' content, and the parameters' size.*At this point, you will have a precise mental representation of the building blocks of the model.*The program now builds the dataset.

## Step 10: Building the dataset

The program will now load the dataset line by line to generate samples for batch training with `block_size=128` limiting the length of an example:

Copy Explain

```
%%time
from transformers import LineByLineTextDataset
dataset = LineByLineTextDataset(
    tokenizer=tokenizer,
    file_path="./kant.txt",
    block_size=128,
)
```

The output shows that Hugging Face has invested a considerable amount of resources in optimizing the time it takes to process data:

Copy Explain

```
CPU times: user 25.9 s, sys: 375 ms, total: 26.3 s
Wall time: 26.9 s
```

The wall time, the actual time the processors were active, is optimized.The program will now define a data collator to create an object for backpropagation.

## Step 11: Defining a data collator

We need to run a data collator before initializing the trainer. A data collator will collate samples from the dataset to prepare batch processing. We are also preparing a batched sample process for MLM by setting `mlm=True`.We also set the number of masked tokens to train `mlm_probability=0.15`. This will determine the percentage of tokens masked during the pretraining process.We now initialize `data_collator` with our tokenizer, MLM activated, and the proportion of masked tokens set to `0.15`:

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

We are now ready to initialize the trainer.

## Step 12: Initializing the trainer

The previous steps have prepared the information required to initialize the trainer. The dataset has been tokenized and loaded. Our model is built. The data collator has been created.The program can now initialize the trainer. For educational purposes, the program trains the model quickly. The number of epochs is limited to one. The GPU comes in handy since we can share the batches and multi-process the training tasks:

```
from transformers import Trainer, TrainingArguments
training_args = TrainingArguments(
    output_dir="./KantaiBERT",
    overwrite_output_dir=True,
    num_train_epochs=1,
    per_device_train_batch_size=64,
    save_steps=10_000,
    save_total_limit=2,
)
trainer = Trainer(
    model=model,
    args=training_args,
    data_collator=data_collator,
    train_dataset=dataset,
)
```

Let's go through each line of the training arguments and the trainer:

`training_args = TrainingArguments(...)`: creating an instance of the TrainingArguments. Here, we're creating an instance of the TrainingArguments class to store the hyperparameters and other arguments for training.

```
output_dir="./KantaiBERT": checkpoint directory
```

`overwrite_output_dir=True`: overwrite content in the output directory.

`num_train_epochs=1`: number of training epochs

`per_device_train_batch_size=64`: the batch size of the number of examples for training. In this case, 64 examples will be trained at the same time.

`save_steps=10_000`: a checkpoint will be saved every 10,000 steps.

`save_total_limit=2`: maximum number of checkpoints that can be saved before the old ones are deleted

Note that batch processing on the device optimizes the architecture of the attention layer of the original Transformer.In *Chapter 2, Getting Started with the Architecture of the Transformer Model,* we defined attention as "Scaled Dot-Product Attention," which is represented by the following equation in which we plug *Q*, *K*, and *V*:

$$Attention(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = softmax\left(\frac{\boldsymbol{Q}\boldsymbol{K}^{\boldsymbol{T}}}{\sqrt{\boldsymbol{d}_{\boldsymbol{k}}}}\right)V$$

Adding batches will increase the learning speed instead of processing one sequence at a time: Q =*[*per_device_train_batch_size=64*,* *dmodel,* dk] K = *[*per_device_train_batch_size=64*,* *dmodel,* dk] V = *[*per_device_train_batch_size=64*,* *dmodel,* dv] If necessary, take a few minutes to go back to *Chapter 2* to reread the attention layer section to review the role of each parameter, including Q, K, V, `dmodel,` and `dk.` Batch processing increases the performance of a model. However, a large batch might speed the process up but limit the learning quality, whereas smaller batches force the model to learn more relationships. Experiment with different batch sizes and measure the performance of the model.

`trainer = Trainer`(...): creating the instance of the trainer class

`model=model`: the model that will be trained. In this case, the model is already defined earlier in our code, as shown when we defined the configuration in step 7 of the notebook:

model_type: "roberta"

`args=training_args`: passing TrainingArguments object we created above.

`data_collator=data_collator`: The data_collator defined in step 11 of the notebook takes samples from the dataset and returns a batch.

`train_dataset=dataset`: the training dataset, which is kant.txt in this notebook.

The model is now ready for training.

## Step 13: Pretraining the model

Everything is ready. The trainer is launched with one line of code:

Copy Explain

```
%%time
trainer.train()
```

The output displays the training process in real time, showing the steps and loss:The model has been trained. It's time to save our work.

## Step 14: Saving the final model (+tokenizer + config) to disk

We will now save the model and configuration:

Copy Explain

```
trainer.save_model("./KantaiBERT")
```

Click on **Refresh** in the file manager, and the files should appear:

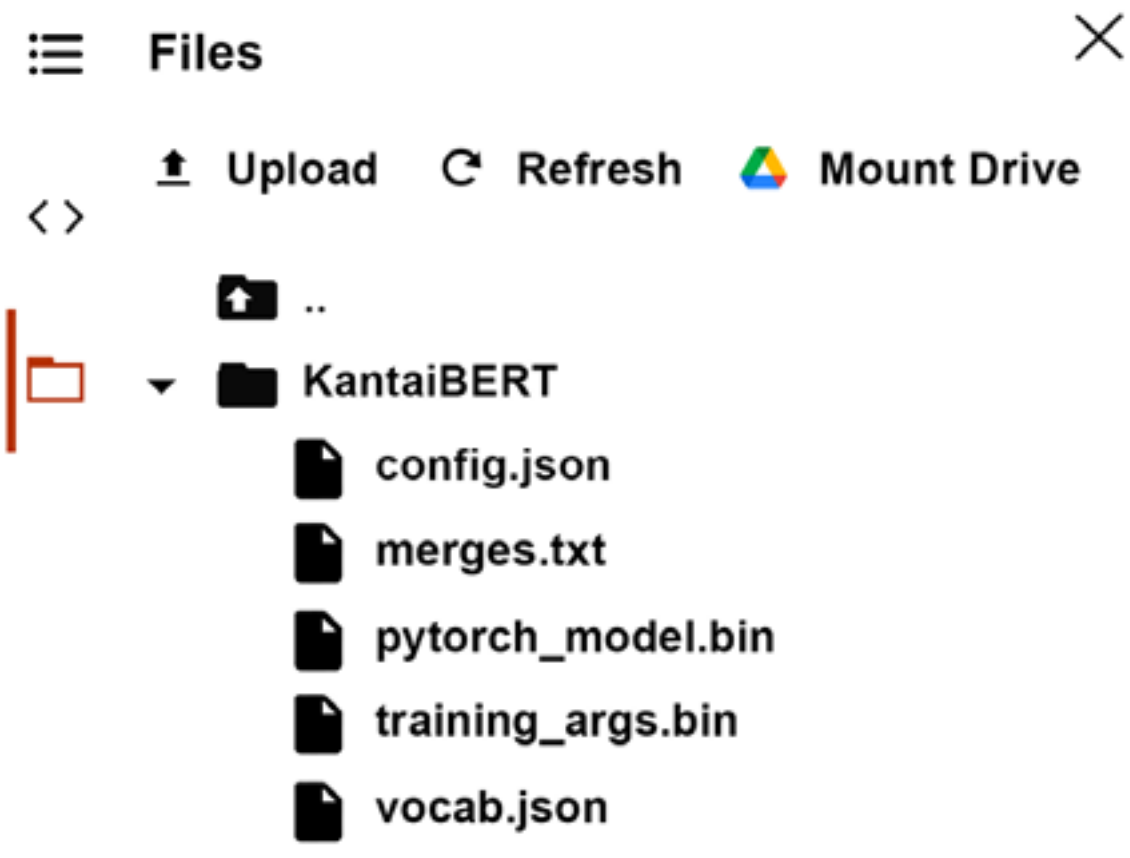


Figure 6.4: Colab file manage with all the KantaiBERT model files

`config.json`, `pytorh_model.bin`, and `training_args.bin` should now appear in the file manager.`merges.txt` and `vocab.json` contain the pretrained tokenization of the dataset.We have built a model from scratch. Let's import the pipeline to perform a language modeling task with our pretrained model and tokenizer.

## Step 15: Language modeling with FillMaskPipeline

We will now import a language modeling `fill-mask` task. We will use our trained model and trained tokenizer to perform MLM:

```
from transformers import pipeline
fill_mask = pipeline(
    "fill-mask",
    model="./KantaiBERT",
    tokenizer="./KantaiBERT"
)
```

We can now ask our model to think like Immanuel Kant:

```
fill_mask("Human thinking involves human <mask>.")
```

The output will likely change after each run because the model is stochastic. Also, we are pretraining the model from scratch with a limited amount of data. However, the output obtained in this run is interesting because it introduces conceptual language modeling:

```
[{'score': 0.038927942514419556,
  'token': 393,
  'token_str': ' reason',
  'sequence': 'Human thinking involves human reason.'},
 {'score': 0.014990510419011116,
  'token': 446,
  'token_str': ' law',
  'sequence': 'Human thinking involves human law.'},
 {'score': 0.011723626405000687,
  'token': 418,
  'token_str': ' conception',
  'sequence': 'Human thinking involves human conception.'},
 {'score': 0.01163031067699194,
  'token': 531,
  'token_str': ' experience',
  'sequence': 'Human thinking involves human experience.'},
 {'score': 0.010935084894299507,
  'token': 600,
  'token_str': ' understanding',
  'sequence': 'Human thinking involves human understanding.'}]
```


The predictions might vary each run and when Hugging Face updates its models.However, the following output comes out often:

```
Human thinking involves human reason
```

The goal here was to see how to train a transformer model. We can see that interesting human-like predictions can be made.These results are experimental and subject to variations during the training process. Therefore, they will change each time we train the model again.The model would require much more data from other *Age of Enlightenment* thinkers.*However, the goal of this model is to show that we can create datasets to train a transformer for a specific type of complex language modeling task*. Let's do that now by building a generative AI model.

---

# Pretraining a Generative AI Customer Support Model on X Data

In this section, we will pretrain a Hugging Face `RobertaForCausalLM` model to be a Generative AI customer support chat agent for X (former Twitter). RoBERTa is an encoder-only model. As such, it is mainly designed to *understand and encode* inputs. In *Chapter 2, Getting Started with the Architecture of the Transformer Model,* we saw how the encoder *learns to understand* and then sends the information to the decoder, generating content. However, in this section, we will use the Hugging Face functionality to adapt a RoBERTa model to run an autoregressive generative AI task. The experiment has limitations, but it shows the inner workings of content generation.The knowledge you acquired in this chapter through building a KantaiBERT from scratch will enable you to enjoy the ride!The generative model and dataset are free, making the exercise particularly interesting. With some work, domain-specific generative AI agents can help companies that want their own models.Generative AI has spread exponentially since the arrival of ChatGPT. Your experience building a prototype chat agent in this section will provide insights into this new era of automated dialogs.We will not repeat all the details of the notebook we went through when building KantaiBERT. Run the notebook cell by cell using the experience gained in this chapter to make sure you don't miss anything. The titles of the steps in the notebook match the ones in this section.Open `Customer_Support_for_X.ipynb` in the chapter's directory on GitHub, and let's get to work!We can get our Generative AI prototype working in only 10 steps.

## Step 1 Downloading the dataset

We will be training the model on Kaggle's *Customer Support on Twitter* dataset that contains 2,800,000+ customer support tweets from the biggest brands on Twitter, such as Apple, American Airlines, Amazon, and many more top corporations.For more on this dataset:https://www.kaggle.com/datasets/thoughtvector/customer-support-on-twitterYou will need a Kaggle account and an API key and will have to activate the authentication. The dataset can then be downloaded with one line of code:

```
!kaggle datasets download -d thoughtvector/customer-support-on-twitter
```

Once the file is downloaded, it is unzipped:

```
import zipfile
with zipfile.ZipFile('/content/customer-support-on-twitter.zip', 'r') as zip_ref:
    zip_ref.extractall('/content/')
print("File Unzipped!")
```

# Step 2: Installing Hugging Face transformers

When installing the transformers library, note that you must also install the accelerator library to leverage the GPU's power.

# Step 3: Loading and filtering the data.

For a prototype, we can fastrack preprocessing the tweets.The raw format of the dataset contains noise we don't need to pretrain a language model as shown by loading and printing the head of the DataFrame:

Copy Explain

```
import pandas as pd
# Load the dataset
df = pd.read_csv('/content/twcs/twcs.csv')
# Check the first few rows to understand the data
print(df.head())
```

The output contains information we don't need to train the model to understand the relationship between the words of the tweets for this prototype:

Copy Explain

```
   tweet_id   author_id  inbound                      created_at  \
0         1  sprintcare    False  Tue Oct 31 22:10:47 +0000 2017
1         2      115712     True  Tue Oct 31 22:11:45 +0000 2017
2         3      115712     True  Tue Oct 31 22:08:27 +0000 2017
3         4  sprintcare    False  Tue Oct 31 21:54:49 +0000 2017
4         5      115712     True  Tue Oct 31 21:49:35 +0000 2017
                                              text response_tweet_id  \
0  @115712 I understand. I would like to assist y...                2
1      @sprintcare and how do you propose we do that              NaN
2  @sprintcare I have sent several private messag...                1
3  @115712 Please send us a Private Message so th...                3
4                                @sprintcare I did.                4
```

The fastest way is to extract words containing the characters we need, such as the alphabet and some punctuation. Run the cells up to this cell and look into it:

Copy Explain

```
import re
def filter_tweet(tweet):
    # Keep only characters a to z, spaces, and apostrophes, then convert to lowercase
    return re.sub(r'[^a-z\s\']', '', tweet.lower())
filtered_tweets = [filter_tweet(tweet) for tweet in tweets]
```

We extracted enough information to pretrain a prototype.We will take the filtering further and only keep tweets that are at least 30 words:

```
f=30
filtered_tweets = [tweet for tweet in filtered_tweets if len(tweet.split()) > f]  #
Only keep tweets with more than f words
```

This way, we will force the model to learn long-term dependencies in sentences with sufficient content, as shown in the output which contains long sequences of words:

```
ags which only contain certain items such as unwrapped food raw meat and fish where
there is a food safety risk prescription medicines uncovered blades seeds bulbs amp s
 hi you can change your microsoft account email through the steps here
httpstcodkehohboyy
```

Run the following cells to save and display information on the data.

# Step 4: Checking Resource Constraints: GPU and CUDA

We need a GPU to pretrain this model for PyTorch. Make sure to activate a GPU and then check your system with the following code:

```
!nvidia-smi
```

You should see a GPU in the following excerpt of the output:

```
NVIDIA-SMI 525.105.17   Driver Version: 525.105.17   CUDA Version: 12.0
```

Also, make sure PyTorch sees CUDA:

```
#@title Checking that PyTorch Sees CUDA
import torch
torch. cuda.is_available()
```

The output should be:

```
True
```

# Step 5: Defining the configuration of the model

The main line of the configuration cell is:

```
from transformers import RobertaConfig, RobertaForCausalLM
```

`RobertaForCausalLM` can be used for generative AI, which constitutes a valuable asset for those who want to pretrain their models.Another feature to focus on is the configuration of the model:

```
config = RobertaConfig(
    vocab_size=52_000,
    max_position_embeddings=514,
    num_attention_heads=12,
    num_hidden_layers=6,
    type_vocab_size=1,
    is_decoder=True,  # Set up the model for potential seq2seq use, allowing for
autoregressive outputs
)
is_decoder=True adds autoregressive functionality to the RoBERTa model, often
implemented with masks. A mask can be anywhere in a sequence of words. An
autoregressive model continues a sequence from the last token forward.
```

We can now define our generative AI model:

```
# Create the RobertaForCausalLM model with the specified config
model = RobertaForCausalLM(config=config)
print(model)
```

We will use Hugging Face's RoBERTa tokenizer:

```
tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
```

The tokenizer contains special tokens, but we will not need the mask, for example.

# Step 6: Creating and processing the dataset

We now install Hugging Face's datasets library:

Copy Explain

```
#installing Hugging Face datasets for data loading and preprocessing
!pip install datasets
```

What does it do? Everything we need in no time. The library:Loads the dataset

Copy Explain

```
#load dataset
from datasets import load_dataset
dataset = load_dataset('csv',
data_files='/content/model/dataset/processed_tweets.csv', column_names=["text"])
```

Splits the dataset into a training and evaluation set

Copy Explain

```
# split datasets into train and eval
from datasets import DatasetDict
dataset = dataset['train'].train_test_split(test_size=0.1)  # 10% for evaluation
dataset = DatasetDict(dataset)
```

Tokenizes the dataset with useful rules.If a record's length is less than `max_length`, it's padded to ensure all records have the same length. If a record's length exceeds `max_length`, it's truncated to the specified max length. The code is compact, padding and truncating all in one line:

Copy Explain

```
def tokenize_function(examples):
    return tokenizer(examples["text"], padding="max_length", truncation=True,
max_length=128)
tokenized_datasets = dataset.map(tokenize_function, batched=True)
```

Data collator to batch items together

Copy Explain

```
The data collator is creating a few lines of code:
# Define the data collator
data_collator = DataCollatorForLanguageModeling(
        tokenizer=tokenizer,
        mlm=False  # For causal (autoregressive) language modeling
)
```

Note that mlm=False turns Masked Language Modeling(MLM) off for autoregressive, generative language modeling.

## Step 7: Initializing the trainer

We first create a class to check if the "step" key exists in the logs and, if it does, prints the current time when every eval_steps number of steps is reached during training:

Copy Explain

```
class CustomTrainer(Trainer):
    def log(self, logs: Dict[str, Any]) -> None:
        super().log(logs)
        if "step" in logs:  # Check if "step" key is in the logs dictionary
            step = int(logs["step"])
            if step % self.args.eval_steps == 0:
                print(f"Current time at step {step}: {datetime.now()}")
```

Then we define the training arguments:

Copy Explain

```
training_args = TrainingArguments(
    output_dir="/content/model/model/",
    overwrite_output_dir=True,
    num_train_epochs=2,                 # can be increased to increase accuracy if
productive
    per_device_train_batch_size=64,     # batch size per device
    save_steps=10_000,                  # save a checkpoint every save_steps=10000
    save_total_limit=2,                 # the maximum number of checkpoint model
files to keep
    logging_dir='/content/model/logs/',  # directory for storing logs
    logging_steps=100,                  # Log every 100 steps
    logging_first_step=True,            # Log the first step
    evaluation_strategy="steps",        # Evaluate every "eval_steps"
    eval_steps=500,                     # Evaluate every 500 steps
)
```

The `TrainingArguments` object is used to set various training parameters for the Hugging Face transformers library, which specifies that the model should be saved in the `"/content/model/model/"` directory and that any existing outputs in this directory should be overwritten. The training will run for two epochs with a batch size of 64 per device. Logging settings determine that logs will be saved in `'/content/model/logs/'`, they will be printed every 100 steps, and evaluation will occur every 500 steps. The model will save a checkpoint twice.Finally, we define the `trainer` in a few lines referring to the steps we ran up to this cell:

Copy Explain

```
trainer = CustomTrainer(
    model=model,
    args=training_args,
    data_collator=data_collator,
    train_dataset = tokenized_datasets["train"],
    eval_dataset = tokenized_datasets["test"]
)
```

## Step 8: Pretraining the model

trainer.train()activates the pretraining process. The progress bar information displays the critical information to monitor to improve the model between each run. The information can change between runs due to the stochastic nature of transformer models. For example:- 218/3216**: Completed 218 tasks out of 3216.- 01:24: Time elapsed since the start.- < 19:39: Approximate time remaining.- 2.54 it/s: Processing speed of 2.54 iterations per second.- Epoch 0.07/1: Currently at 7% of 1 full epoch.Under the progress bar, we can visualize:

`step:` number of steps

`training loss`: must diminish

`validation loss`: should be inferior or converge with the training loss. If it is superior, the model may be overfitting.

## Step 9: Saving the model

You can save the model to run a standalone session to chat with the generative AI agent once it's trained:

Copy Explain

```
trainer.save_model("/content/model/model/")
```

Google Colab will recycle the model when the session is closed. You can save it to any location you wish, for example in Google Drive:

Copy Explain

```
#Uncomment the following line to save the output for future use
#trainer.save_model("drive/MyDrive/files/model_C6/model/")
```

# Step 10: User Interface to Chat with the Generative AI Agent

After installing `ipywidgets` to build the interface and making sure that the model and tokenizer are initialized, we can evaluate the prototype by creating a function to generate content:

Copy Explain

```
!pip install ipywidgets
import ipywidgets as widgets
from IPython.display import display, clear_output
from transformers import RobertaTokenizer, RobertaForCausalLM
# Define the function to generate response
def generate_response(prompt):
    # Reload model and tokenizer for each request
    model = RobertaForCausalLM.from_pretrained('roberta-base')
    tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
    inputs = tokenizer(prompt, return_tensors="pt", max_length=50, truncation=True)
    output = model.generate(**inputs, max_length=100, temperature=0.9,
num_return_sequences=1)
    generated_text = tokenizer.decode(output[0], skip_special_tokens=True)
    return generated_text
# Create widgets
text_input = widgets.Textarea(
    description='Prompt:',
    placeholder='Enter your prompt here...'
)
button = widgets.Button(
    description='Generate',
    button_style='success'
)
output_text = widgets.Output(layout={'border': '1px solid black', 'height': '100px'})
# Define button click event handler
def on_button_clicked(b):
    with output_text:
        clear_output()
        response = generate_response(text_input.value)
        print(response)
button.on_click(on_button_clicked)
# Display widgets
display(text_input, button, output_text)
```

We can now enter a prompt as shown in Figure xxx.xxx

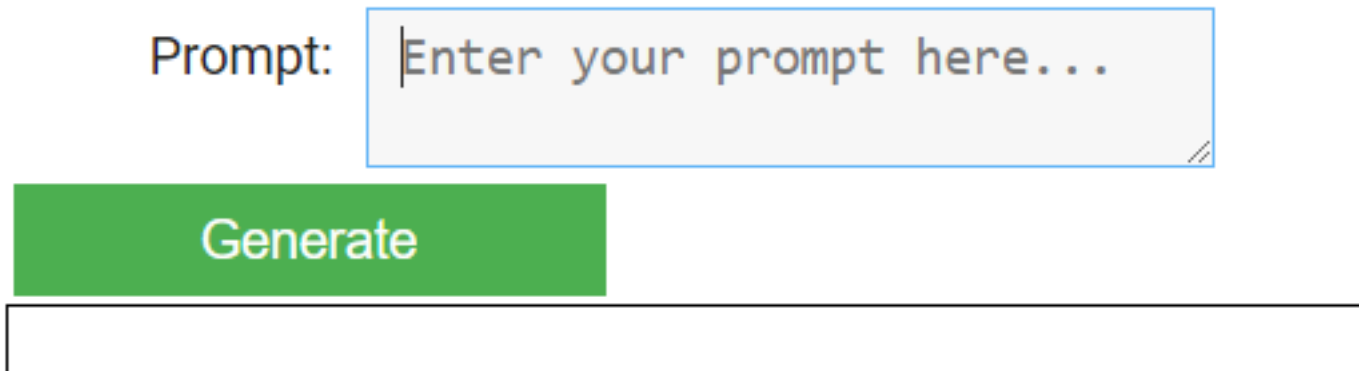


Figure xxx.xxx; User Interface

We can try a prompt and click on Generate to display the generative AI output as shown in Figure xxx.xxx.

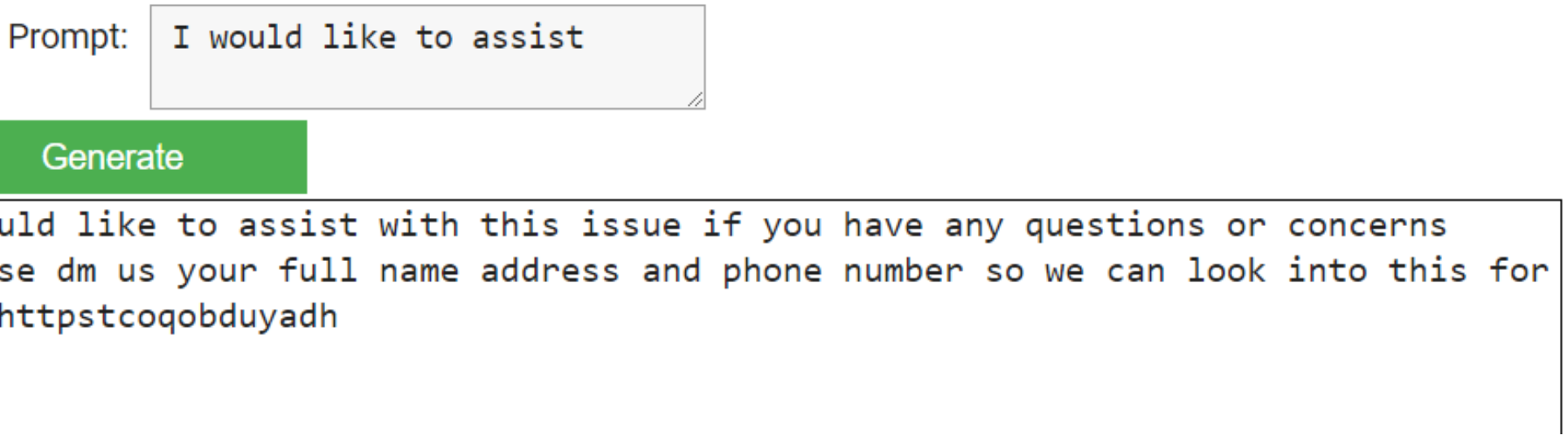


*Figure xxx.xxx* Prompt and generated response.

In this case, the output is experimental and interesting. The last part could be filtered with a dictionary of known words. However, generative AI is stochastic and can produce errors. Try input close to the trained samples first, for example, with no punctuation:

I would like to assist you

I need help to find the battery of

I would like to know why they moved us

Further training is required if you wish to go further.

## Further pretraining

The model was trained on a subset of the original dataset for a limited number of epochs. To expand the pretraining process, modify the dataset filter (`f=30` in **step 3**), increase the number of epochs, try other models, modify the hyperparameters, increase the size/quality of the datasets, and more. Although running a reasonably sized model is exciting, there are limitations.

# Limitations

This is only a functional interface POC (Proof of Concept) to verify that this approach works. RoBERTa wasn't designed for generative AI, but Hugging Face provided the environment to implement it. It isn't a native generative AI model approach. However, who knows? The knowledge of fully trained RoBERTa can be leveraged by transferring its knowledge to a decoder model through transfer learning bridges. We opened a door in this section that you can choose to explore further or not, depending on the goals of your project.*You're entering the challenging but exciting world of generative AI designers!*We created an open-source generative AI Model that can be considerably expanded.Thanks to transformers, we are only at the beginning of a new era of AI!

---

# Next steps

You have trained two transformers from scratch. Take some time to imagine what you could do in your personal or corporate environment. You could create a dataset for a specific task and train it from scratch. Many other Hugging Face models are available for training in the BERT family, GPT models, T5, and more!Use your areas of interest or company projects to experiment with the fascinating world of transformer construction kits!Once you have made a model you like, you can share it with the Hugging Face community. Your model will appear on the Hugging Face models page: https://huggingface.co/modelsYou can upload your model in a few steps using the instructions described on this page: https://huggingface.co/transformers/model_sharing.htmlYou can also download models the Hugging Face community has shared to get new ideas for your personal and professional projects.

---

# Summary

In this chapter, we built `KantaiBERT`, a RoBERTa-like model transformer, from scratch using the building blocks provided by Hugging Face.We first started by loading a customized dataset on a specific topic related to the works of Immanuel

Kant. Depending on your goals, you can load an existing dataset or create your own. We saw that using a customized dataset provides insights into how a transformer model thinks. However, this experimental approach has its limits. Training a model beyond educational purposes would take a much larger dataset.The KantaiBERT project was used to train a tokenizer on the `kant.txt` dataset. The trained `merges.txt` and `vocab.json` files were saved. A tokenizer was recreated with our pretrained files. KantaiBERT built the customized dataset and defined a data collator to process the training batches for backpropagation. The trainer was initialized, and we explored the parameters of the RoBERTa model in detail. The model was trained and saved.We saved the model and loaded it for a downstream language modeling task. The goal was to fill the mask using Immanuel Kant's logic.Finally, we pretrained a Generative AI Customer Support Model on X (former Twitter). We used an open-source RoBERTa model and a free dataset. This prototype showed that we can build chat agents independently for specific domains when necessary. The door is now wide open for you to experiment on existing or customized datasets to see what results you get. You can share your model with the Hugging Face community. Transformers are data-driven. You can use this to your advantage to discover new ways of using transformers.You are now ready to learn how to run ready-to-use transformer engines with APIs that require no pretraining or fine-tuning. *Chapter 7, From Basics to Disruption with ChatGPT,* will take you into the world of suprahuman Large Language Models (LMMs). And with the knowledge of this chapter and the past chapters, you will be ready!

---

# Questions

1. RoBERTa uses a byte-level byte-pair encoding tokenizer. (True/False)
2. A trained Hugging Face tokenizer produces `merges.txt` and `vocab.json`. (True/False)
3. RoBERTa does not use token-type IDs. (True/False)
4. DistilBERT has 6 layers and 12 heads. (True/False)
5. A transformer model with 80 million parameters is enormous. (True/False)
6. We cannot train a tokenizer. (True/False)
7. A BERT-like model has 6 decoder layers. (True/False)
8. Masked Language Modeling (MLM) predicts a word contained in a mask token in a sentence. (True/False)
9. A BERT-like model has no self-attention sublayers. (True/False)

10. Data collators are helpful for backpropagation. (True/False)

# References


Hugging Face Tokenizer documentation: https://huggingface.co/transformers/main_classes/tokenizer.html?highlight=tokenizer

The Hugging Face reference notebook: https://colab.research.google.com/github/huggingface/blog/blob/master/notebooks/01_how_to_train.ipynb

The Hugging Face reference blog: https://colab.research.google.com/github/huggingface/blog/blob/master/notebooks/01_how_to_train.ipynb

More on BERT: https://huggingface.co/transformers/model_doc/bert.html

More DistilBERT: https://arxiv.org/pdf/1910.01108.pdf

More on RoBERTa: https://huggingface.co/transformers/model_doc/roberta.html

Even more on DistilBERT: https://huggingface.co/transformers/model_doc/distilbert.html


# Further Reading


Pretraining of Deep Bidirectional Transformers for Language Understanding, Devlin, et al.(2018): https://arxiv.org/abs/1810.04805

RoBERTa: A Robustly Optimized BERT Pretraining, Liu et al.(2019) Approach: https://arxiv.org/abs/1907.11692

# Join our book's Discord space

Join the book's Discord workspace:https://www.packt.link/Transformers

Previous Chapter

Next Chapter