CHAPTER 10
Training Transformers from Scratch

In the opening paragraph of this book, we mentioned a sophisticated application called GitHub Copilot that uses GPT-like transformers to perform code autocompletion, a feature that is particularly useful when programming in a new language or framework or learning to code, or for automatically producing boilerplate code. Other products that use AI models for this purpose include TabNine and Kite. Later, in Chapter 5, we had a closer look at how we can use GPT models to generate high-quality text. In this chapter, we'll close the circle and build our very own GPT-like model for generating Python source code! We call the resulting model *CodeParrot*.

So far we've mostly worked on data-constrained applications where the amount of labeled training data is limited. In these cases, transfer learning helped us build performant models. We took transfer learning to the limit in Chapter 9, where we barely used any training data at all.

In this chapter we'll move to the other extreme and look at what we can do when we are drowning in all the data we could possibly want. We'll explore the pretraining step itself and learn how to train a transformer from scratch. In working through this problem, we'll look at some aspects of training that we have not considered yet, such as the following:

- Gathering and processing a very large dataset
- Creating a custom tokenizer for our dataset
- Training a model on multiple GPUs at scale

To efficiently train large models with billions of parameters, we'll need special tools for distributed training. Although the `Trainer` from 🤗 Transformers supports distributed training, we'll take the opportunity to showcase a powerful PyTorch library

called 🤗 Accelerate. We'll end up touching on some of the largest NLP models in use today—but first, we need to find a sufficiently large dataset.

Unlike the code in the others in this book (which can be run with a Jupyter notebook on a single GPU), the training code in this chapter is designed to be run as a script with multiple GPUs. If you want to train your own version of CodeParrot, we recommend running the script provided in the 🤗 Transformers repository.

Large Datasets and Where to Find Them

There are many domains where you may actually have a large amount of data at hand, ranging from legal documents to biomedical datasets to programming codebases. In most cases, these datasets are unlabeled, and their large size means that they can usually only be labeled through the use of heuristics, or by using accompanying metadata that is stored during the gathering process.

Nevertheless, a very large corpus can be useful even when it is unlabeled or only heuristically labeled. We saw an example of this in Chapter 9, where we used the unlabeled part of a dataset to fine-tune a language model for domain adaptation. This approach typically yields a performance gain when limited data is available. The decision to train from scratch rather than fine-tune an existing model is mostly dictated by the size of your fine-tuning corpus and the domain differences between the available pretrained models and the corpus.

Using a pretrained model forces you to use the model's corresponding tokenizer, but using a tokenizer that is trained on a corpus from another domain is typically suboptimal. For example, using GPT's pretrained tokenizer on legal documents, other languages, or even completely different sequences such as musical notes or DNA sequences will result in poor tokenization (as we will see shortly).

As the amount of training data you have access to gets closer to the amount of data used for pretraining, it thus becomes interesting to consider training the model and the tokenizer from scratch, provided the necessary computational resources are available. Before we discuss the different pretraining objectives further, we first need to build a large corpus suitable for pretraining. Building such a corpus comes with its own set of challenges, which we'll explore in the next section.

Challenges of Building a Large-Scale Corpus

The quality of a model after pretraining largely reflects the quality of the pretraining corpus. In particular, the model will inherit any defects in the pretraining corpus. Thus, before we attempt to create one of our own it's good to be aware of some of the

common issues and challenges that are associated with building large corpora for pretraining.

As the dataset gets larger and larger, the chances that you can fully control—or at least have a precise idea of—what is inside it diminish. A very large dataset will most likely not have been assembled by dedicated creators that craft one example at a time, while being aware and knowledgeable of the full pipeline and the task that the machine learning model will be applied to. Instead, it is much more likely that a very large dataset will have been created in an automatic or semiautomatic way by collecting data that is generated as a side effect of other activities. For instance, it may consist of all the documents (e.g., contracts, purchase orders, etc.) that a company stores, logs from user activities, or data gathered from the internet.

There are several important consequences that follow from the fact that large-scale datasets are mostly created with a high degree of automation. An important consideration is that there is limited control over both their content and the way they are created, and thus the risk of training a model on biased and lower-quality data increases. Recent investigations of famous large-scale datasets like BookCorpus and C4, which were used to train BERT and T5, respectively, have uncovered (among other things) that:[1]

- A significant proportion of the C4 corpus is machine-translated rather than translated by humans.
- Disparate erasure of African-American English as a result of stopword filtering in C4 has resulted in an underrepresentation of such content.
- It is typically difficult in a large text corpus to find a middle ground between including (often too much) sexually or other explicit content and totally erasing all mention of sexuality or gender. As a surprising consequence of this, a rather common word like "sex" (which can have both neutral and explicit meanings) is completely unknown to a tokenizer that is trained on C4, since this word is fully absent from the corpus.
- There are many occurrences of copyright violation in BookCorpus, and probably in other large-scale datasets as well.[2]
- There is genre skew toward "romance" novels in BookCorpus.

These discoveries might not be incompatible with downstream usage of the models trained on these corpora. For instance, the strong overrepresentation of romance

1 Y. Zhu et al., "Aligning Books and Movies: Towards Story-Like Visual Explanations by Watching Movies and Reading Books", (2015); J. Dodge et al., "Documenting the English Colossal Clean Crawled Corpus", (2021).

2 J. Bandy and N. Vincent, "Addressing *Documentation Debt* in Machine Learning Research: A Retrospective Datasheet for BookCorpus", (2021).

novels in BookCorpus is probably acceptable if the model is intended to be used as a romance novel writing tool or for a building a game.

Let's illustrate the notion of a model being skewed by the data by comparing text generations from GPT and GPT-2. GPT was mostly trained on BookCorpus, while GPT-2 was trained on web pages, blogs, and news articles linked from Reddit. We'll compare similar-sized versions of both models on the same prompt, so that the main difference is the pretraining dataset, and we'll use the `text-generation` pipeline to investigate the model outputs:

```
from transformers import pipeline, set_seed

generation_gpt = pipeline("text-generation", model="openai-gpt")
generation_gpt2 = pipeline("text-generation", model="gpt2")
```

Next, let's create a simple function to count the number of parameters in each model:

```
def model_size(model):
    return sum(t.numel() for t in model.parameters())

print(f"GPT  size: {model_size(generation_gpt.model)/1000**2:.1f}M parameters")
print(f"GPT2 size: {model_size(generation_gpt2.model)/1000**2:.1f}M parameters")
```

```
GPT  size: 116.5M parameters
GPT2 size: 124.4M parameters
```

The original GPT model is about the same size as the smallest GPT-2 model. Now we can generate three different completions from each model, each with the same input prompt:

```
def enum_pipeline_ouputs(pipe, prompt, num_return_sequences):
    out = pipe(prompt, num_return_sequences=num_return_sequences,
               clean_up_tokenization_spaces=True)
    return "\n".join(f"{i+1}." + s["generated_text"] for i, s in enumerate(out))

prompt = "\nWhen they came back"
print("GPT completions:\n" + enum_pipeline_ouputs(generation_gpt, prompt, 3))
print("")
print("GPT-2 completions:\n" + enum_pipeline_ouputs(generation_gpt2, prompt, 3))
```

```
GPT completions:
1.
When they came back.
 " we need all we can get, " jason said once they had settled into the back of
the truck without anyone stopping them. " after getting out here, it 'll be up
to us what to find. for now
2.
When they came back.
 his gaze swept over her body. he 'd dressed her, too, in the borrowed clothes
that she 'd worn for the journey.
 " i thought it would be easier to just leave you there. " a woman like
3.
When they came back to the house and she was sitting there with the little boy.
```

```
 " don't be afraid, " he told her. she nodded slowly, her eyes wide. she was so
lost in whatever she discovered that tom knew her mistake

GPT-2 completions:
1.
When they came back we had a big dinner and the other guys went to see what
their opinion was on her. I did an hour and they were happy with it.
2.
When they came back to this island there had been another massacre, but he could
not help but feel pity for the helpless victim who had been left to die, and
that they had failed that day. And so was very, very grateful indeed.
3.
When they came back to our house after the morning, I asked if she was sure. She
said, "Nope." The two kids were gone that morning. I thought they were back to
being a good friend.

When Dost
```

By just sampling a handful of outputs from both models we can already see the distinctive "romance" skew in GPT generation, which will typically imagine a dialogue with a romantic interaction between a woman and a man. On the other hand, GPT-2 was trained on webtext linked to and from Reddit articles and mostly adopts a neutral "they" in its generations, which contain "blog-like" or adventure-related elements.

In general, any model trained on a dataset will reflect the language bias and over- or underrepresentation of populations and events in its training data. These biases in the behavior of the model are important to take into consideration with regard to the target audience interacting with the model; for some useful guidelines, we refer you to a paper by Google that provides a framework for dataset development.[3]

This brief introduction should give you an idea of the difficult challenges you face when creating large text corpora. With these in mind, let's now take a look at creating our own dataset!

Building a Custom Code Dataset

To simplify the task a bit, we'll focus on building a code generation model for the Python programming language only.[4] The first thing we'll need is a large pretraining corpus consisting of Python source code. Fortunately, there is a natural resource that every software engineer knows: GitHub! The famous code-sharing website hosts terabytes of code repositories that are openly accessible and can be downloaded and used according to their respective licenses. At the time of this book's writing, GitHub

3 B. Hutchinson et al., "Towards Accountability for Machine Learning Datasets: Practices from Software Engineering and Infrastructure", (2020).

4 By comparison, GitHub Copilot supports over a dozen programming languages.

hosts more than 20 million code repositories. Many of them are small or test repositories created by users for learning, future side projects, or testing purposes.

GitHub repositories can be accessed in two main ways:

- Via the GitHub REST API, like we saw in Chapter 9 when we downloaded all the GitHub issues of the 🤗 Transformers repository
- Via public dataset inventories like Google BigQuery

Since the REST API is rate limited and we need a lot data for our pretraining corpus, we'll use Google BigQuery to extract all the Python repositories. The `bigquery-public-data.github_repos.contents` table contains copies of all ASCII files that are less than 10 MB in size. Projects also need to be open source to be included, as determined by GitHub's License API.

The Google BigQuery dataset doesn't contain star or downstream usage information. For those attributes, we can use the GitHub REST API or a service like Libraries.io that monitors open source packages. Indeed, a team from GitHub recently released a dataset called CodeSearchNet that filtered repositories used in at least one downstream task using information from Libraries.io.

Let's have a look at what it takes to create our code dataset with Google BigQuery.

Creating a dataset with Google BigQuery

We'll begin by extracting all the Python files in GitHub public repositories from the snapshot on Google BigQuery. For the sake of reproducibility and in case the policy around free usage of BigQuery changes in the future, we will also share this dataset on the Hugging Face Hub. The steps to export these files are adapted from the TransCoder implementation and are as follows:[5]

1. Create a Google Cloud account (a free trial should be sufficient).
2. Create a Google BigQuery project under your account.
3. In this project, create a dataset.
4. In this dataset, create a table where the results of the SQL request will be stored.
5. Prepare and run the following SQL query on the `github_repos` (to save the query results, select More > Query Options, check the "Set a destination table for query results" box, and specify the table name):

5 M.-A. Lachaux et al., "Unsupervised Translation of Programming Languages", (2020).

```
SELECT
  f.repo_name, f.path, c.copies, c.size, c.content, l.license
FROM
  `bigquery-public-data.github_repos.files` AS f
JOIN
  `bigquery-public-data.github_repos.contents` AS c
ON
  f.id = c.id
JOIN
  `bigquery-public-data.github_repos.licenses` AS l
ON
  f.repo_name = l.repo_name
WHERE
  NOT c.binary
  AND ((f.path LIKE '%.py')
    AND (c.size BETWEEN 1024
      AND 1048575))
```

This command processes about 2.6 TB of data to extract 26.8 million files. The result is a dataset of about 50 GB of compressed JSON files, each containing the source code of Python files. We filtered to remove empty files and small files such as *__init__.py* that don't contain much useful information. We also filtered out files larger than 1 MB, and we downloaded the licenses for all the files so we can filter the training data based on licenses if we want later on.

Next, we'll download the results to our local machine. If you try this at home, make sure you have good bandwidth available and at least 50 GB of free disk space. The easiest way to get the resulting table to your local machine is to follow this two-step process:

1. Export your results to Google Cloud:
   a. Create a bucket and a folder in Google Cloud Storage (GCS).
   b. Export your table to this bucket by selecting Export > Export to GCS, with an export format of JSON and gzip compression.
2. To download the bucket to your machine, use the `gsutil` library:
   a. Install `gsutil` with `pip install gsutil`.
   b. Configure `gsutil` with your Google account: `gsutil config`.
   c. Copy your bucket on your machine:

```
$ gsutil -m -o
"GSUtil:parallel_process_count=1" cp -r gs://<name_of_bucket>
```

Alternatively, you can directly download the dataset from the Hugging Face Hub with the following command:

```
$ git clone https://huggingface.co/datasets/transformersbook/codeparrot
```

To Filter the Noise or Not?

Anybody can create a GitHub repository, so the quality of the projects varies. There are some conscious choices to be made regarding how we want the system to perform in a real-world setting. Having some noise in the training dataset will make our system more robust to noisy inputs at inference time, but will also make its predictions more random. Depending on the intended use and whole system integration, you may choose more or less noisy data and add pre- and postfiltering operations.

For the educational purposes of the present chapter and to keep the data preparation code concise, we will not filter according to stars or usage and will just grab all the Python files in the GitHub BigQuery dataset. Data preparation, however, is a crucial step, and you should make sure you clean up your dataset as much as possible. In our case a few things to consider are whether to balance the programming languages in the dataset; filter low-quality data (e.g., via GitHub stars or references from other repos); remove duplicated code samples; take copyright information into account; investigate the language used in documentation, comments, or docstrings; and remove personal identifying information such as passwords or keys.

Working with a 50 GB dataset can be challenging; it requires sufficient disk space, and one must be careful not to run out of RAM. In the following section, we'll have a look how 🤗 Datasets helps deal with these constraints of working with large datasets on small machines.

Working with Large Datasets

Loading a very large dataset is often a challenging task, in particular when the data is larger than your machine's RAM. For a large-scale pretraining dataset, this is a very common situation. In our example, we have 50 GB of compressed data and about 200 GB of uncompressed data, which is difficult to extract and load into the RAM memory of a standard-sized laptop or desktop computer.

Thankfully, 🤗 Datasets has been designed from the ground up to overcome this problem with two specific features that allow you to set yourself free from RAM and hard drive space limitations: memory mapping and streaming.

Memory mapping

To overcome RAM limitations, 🤗 Datasets uses a mechanism for zero-copy and zero-overhead memory mapping that is activated by default. Basically, each dataset is cached on the drive in a file that is a direct reflection of the content in RAM memory. Instead of loading the dataset in RAM, 🤗 Datasets opens a read-only pointer to this

file and uses it as a substitute for RAM, basically using the hard drive as a direct extension of the RAM memory.

Up to now we have mostly used 🤗 Datasets to access remote datasets on the Hugging Face Hub. Here, we will directly load our 50 GB of compressed JSON files that we have stored locally in the `codeparrot` repository. Since the JSON files are compressed, we first need to decompress them, which 🤗 Datasets takes care of for us. Be careful, because this requires about 180 GB of free disk space! However, it will use almost no RAM. By setting `delete_extracted=True` in the dataset's downloading configuration, we can make sure that we delete all the files we don't need anymore as soon as possible:

```
from datasets import load_dataset, DownloadConfig

download_config = DownloadConfig(delete_extracted=True)
dataset = load_dataset("./codeparrot", split="train",
                       download_config=download_config)
```

Under the hood, 🤗 Datasets extracted and read all the compressed JSON files by loading them in a single optimized cache file. Let's see how big this dataset is once loaded:

```
import psutil

print(f"Number of python files code in dataset : {len(dataset)}")
ds_size = sum(os.stat(f["filename"]).st_size for f in dataset.cache_files)
# os.stat.st_size is expressed in bytes, so we convert to GB
print(f"Dataset size (cache file) : {ds_size / 2**30:.2f} GB")
# Process.memory_info is expressed in bytes, so we convert to MB
print(f"RAM used: {psutil.Process(os.getpid()).memory_info().rss >> 20} MB")
```

```
Number of python files code in dataset : 18695559
Dataset size (cache file) : 183.68 GB
RAM memory used: 4924 MB
```

As we can see, the dataset is much larger than our typical RAM memory, but we can still load and access it, and we're actually using a very limited amount of memory.

You may wonder if this will make our training I/O-bound. In practice, NLP data is usually very lightweight to load in comparison to the model processing computations, so this is rarely an issue. In addition, the zero-copy/zero-overhead format uses Apache Arrow under the hood, which makes it very efficient to access any element. Depending on the speed of your hard drive and the batch size, iterating over the dataset can typically be done at a rate of a few tenths of a GB/s to several GB/s. This is great, but what if you can't free enough disk space to store the full dataset locally? Everybody knows the feeling of helplessness when you get a full disk warning and need to painfully try to reclaim a few GB by looking for hidden files to delete. Luckily, you don't need to store the full dataset locally if you use the streaming feature of 🤗 Datasets!

Streaming

Some datasets (reaching up to 1 TB or more) will be difficult to fit even on a standard hard drive. In this case, an alternative to scaling up the server you are using is to *stream* the dataset. This is also possible with 🤗 Datasets for a number of compressed or uncompressed file formats that can be read line by line, like JSON Lines, CSV, or text (either raw or zip, gzip, or zstandard compressed). Let's load our dataset directly from the compressed JSON files instead of creating a cache file from them:

```
streamed_dataset = load_dataset('./codeparrot', split="train", streaming=True)
```

As you'll see, loading the dataset is instantaneous! In streaming mode, the compressed JSON files will be opened and read on the fly. Our dataset is now an `IterableDataset` object. This means that we cannot access random elements of it, like `streamed_dataset[1264]`, but we need to read it in order, for instance with `next(iter(streamed_dataset))`. It's still possible to use methods like `shuffle()`, but these will operate by fetching a buffer of examples and shuffling within this buffer (the size of the buffer is adjustable). When several files are provided as raw files (like our 184 files here), `shuffle()` will also randomize the order of files for the iteration.

The samples of a streamed dataset are identical to the samples of a nonstreamed dataset, as we can see:

```
iterator = iter(streamed_dataset)

print(dataset[0] == next(iterator))
print(dataset[1] == next(iterator))
```

```
True
True
```

The main interest of using a streaming dataset is that loading this dataset will not create a cache file on the drive or require any (significant) RAM memory. The original raw files are extracted and read on the fly when a new batch of examples is requested, and only that batch is loaded in memory. This reduces the memory footprint of our dataset from 180 GB to 50 GB. But we can take this one step further—instead of pointing to the local dataset we can reference the dataset on the Hub, and then directly download samples without downloading the raw files locally:

```
remote_dataset = load_dataset('transformersbook/codeparrot', split="train",
                              streaming=True)
```

This dataset behaves exactly like the previous one, but behind the scenes downloads the examples on the fly. With such a setup, we can then use arbitrarily large datasets on an (almost) arbitrarily small server. Let's push our dataset with a train and validation split to the Hugging Face Hub and access it with streaming.

Adding Datasets to the Hugging Face Hub

Pushing our dataset to the Hugging Face Hub will allow us to:

- Easily access it from our training server.
- See how streaming datasets work seamlessly with datasets from the Hub.
- Share it with the community, including you, dear reader!

To upload the dataset, we first need to log in to our Hugging Face account by running the following command in the terminal and providing the relevant credentials:

```
$ huggingface-cli login
```

This is equivalent to the `notebook_login()` helper function we used in previous chapters. Once this is done, we can directly create a new dataset on the Hub and upload the compressed JSON files. To simplify things, we will create two repositories: one for the train split and one for the validation split. We can do this by running the `repo create` command of the CLI as follows:

```
$ huggingface-cli repo create --type dataset --organization transformersbook \
codeparrot-train
$ huggingface-cli repo create --type dataset --organization transformersbook \
codeparrot-valid
```

Here we've specified that the repository should be a dataset (in contrast to the model repositories used to store weights), along with the organization we'd like to store the repositories under. If you're running this code under your personal account, you can omit the `--organization` flag. Next, we need to clone these empty repositories to our local machine, copy the JSON files to them, and push the changes to the Hub. We will take the last compressed JSON file out of the 184 we have as the validation file (i.e., roughly 0.5 percent of our dataset). Execute these commands to clone the repository from the Hub to your local machine:

```
$ git clone https://huggingface.co/datasets/transformersbook/codeparrot-train
$ git clone https://huggingface.co/datasets/transformersbook/codeparrot-valid
```

Next, copy all but the last GitHub file as the training set:

```
$ cd codeparrot-train
$ cp ../codeparrot/*.json.gz .
$ rm ./file-000000000183.json.gz
```

Then commit the files and push them to the Hub:

```
$ git add .
$ git commit -m "Adding dataset files"
$ git push
```

Now, repeat the process for the validation set:

```
$ cd ../codeparrot-valid
$ cp ../codeparrot/file-000000000183.json.gz .
$ mv ./file-000000000183.json.gz ./file-000000000183_validation.json.gz
$ git add .
$ git commit -m "Adding dataset files"
$ git push
```

The `git add .` step can take a couple of minutes since a hash of all the files is computed. Uploading all the files will also take a little while. Since this will enable us to use streaming later in the chapter, however, this is not lost time, and this step will allow us to go significantly faster in the rest of our experiments. Note that we added a `_validation` suffix to the validation filename. This will enable us to load it later as a validation split.

And that's it! Our two splits of the dataset as well as the full dataset are now live on the Hugging Face Hub at the following URLs:

- *https://huggingface.co/datasets/transformersbook/codeparrot*
- *https://huggingface.co/datasets/transformersbook/codeparrot-train*
- *https://huggingface.co/datasets/transformersbook/codeparrot-valid*

It's good practice to add README cards that explain how the datasets were created and provide as much useful information about them as possible. A well-documented dataset is more likely to be useful to other people, as well as your future self. You can read the 🤗 Datasets README guide for a detailed description of how to write good dataset documentation. You can also use the web editor to modify your README cards directly on the Hub later.

Building a Tokenizer

Now that we have gathered and loaded our large dataset, let's see how we can efficiently process the data to feed to our model. In the previous chapters we've used tokenizers that accompanied the models we used. This made sense since these models were pretrained using data passed through a specific preprocessing pipeline defined in the tokenizer. When using a pretrained model, it's important to stick with the same preprocessing design choices selected for pretraining. Otherwise the model may be fed out-of-distribution patterns or unknown tokens.

However, when we train a new model, using a tokenizer prepared for another dataset can be suboptimal. Here are a few examples of the kinds of problems we might run into when using an existing tokenizer:

- The T5 tokenizer was trained on the C4 corpus that we encountered earlier, but an extensive step of stopword filtering was used to create it. As a result, the T5 tokenizer has never seen common English words such as "sex."
- The CamemBERT tokenizer was also trained on a very large corpus of text, but only comprising French text (the French subset of the OSCAR corpus). As such, it is unaware of common English words such "being."

We can easily test these features of each tokenizer in practice:

```
from transformers import AutoTokenizer

def tok_list(tokenizer, string):
    input_ids = tokenizer(string, add_special_tokens=False)["input_ids"]
    return [tokenizer.decode(tok) for tok in input_ids]

tokenizer_T5 = AutoTokenizer.from_pretrained("t5-base")
tokenizer_camembert = AutoTokenizer.from_pretrained("camembert-base")

print(f'T5 tokens for "sex": {tok_list(tokenizer_T5,"sex")}')
print(f'CamemBERT tokens for "being": {tok_list(tokenizer_camembert,"being")}')
```

```
T5 tokens for "sex": ['', 's', 'ex']
CamemBERT tokens for "being": ['be', 'ing']
```

In many cases, splitting such short and common words into subparts will be inefficient, since this will increase the input sequence length of the model (which has limited context). Therefore, it's important to be aware of the domain and preprocessing of the dataset that was used to train the tokenizer. The tokenizer and model can encode bias from the dataset that has an impact on the downstream behavior of the model. To create an optimal tokenizer for our dataset, we thus need to train one ourselves. Let's see how this can be done.

Training a model involves starting from a given set of weights and using backpropagation from an error signal on a designed objective to minimize the loss of the model and find an optimal set of weights for the model to perform the task defined by the training objective. Training a tokenizer, on the other hand, does *not* involve backpropagation or weights. It is a way to create an optimal mapping from a string of text to a list of integers that can be ingested by the model. In today's tokenizers, the optimal string-to-integer conversion involves a vocabulary consisting of a list of atomic strings and an associated method to convert, normalize, cut, or map a text string into a list of indices with this vocabulary. This list of indices is then the input for our neural network.

The Tokenizer Model

As you saw in Chapter 4, the tokenizer is a processing pipeline consisting of four steps: normalization, pretokenization, the tokenizer model, and postprocessing. The part of the tokenizer pipeline that can be trained on data is the tokenizer model. As we discussed in Chapter 2, there are several subword tokenization algorithms that can be used, such as BPE, WordPiece, and Unigram.

BPE starts from a list of basic units (single characters) and creates a vocabulary by a process of progressively creating new tokens formed by merging the most frequently co-occurring basic units and adding them to the vocabulary. This process is reiterated until a predefined vocabulary size is reached.

Unigram starts from the other end, by initializing its base vocabulary with all the words in the corpus, and potential subwords. Then it progressively removes or splits the less useful tokens to obtain a smaller and smaller vocabulary, until the target vocabulary size is reached. WordPiece is a predecessor of Unigram, and its official implementation was never open-sourced by Google.

The impact of these various algorithms on downstream performance varies depending on the task, and overall it's quite difficult to identify if one algorithm is clearly superior to the others. Both BPE and Unigram have reasonable performance in most cases, but let's have a look at some aspects to consider when evaluating.

Measuring Tokenizer Performance

The optimality and performance of a tokenizer are challenging to measure in practice. Some possible metrics include:

- *Subword fertility*, which calculates the average number of subwords produced per tokenized word
- *Proportion of continued words*, which refers to the proportion of tokenized words in a corpus that are split into at least two subtokens
- *Coverage metrics* like the proportion of unknown words or rarely used tokens in a tokenized corpus

In addition, robustness to misspelling or noise is often estimated, as well as model performance on such out-of-domain examples, as this strongly depends on the tokenization process.

These measures give a set of different views on the tokenizer's performance, but they tend to ignore the interaction of the tokenizer with the model. For example, subword fertility can be minimized by including all the possible words in the vocabulary, but this will produce a very large vocabulary for the model.

In the end, the performance of the various tokenization approaches is thus generally best estimated by using the downstream performance of the model as the ultimate metric. For instance, the good performance of early BPE approaches was demonstrated by showing improved performance on machine translation tasks by models trained using these tokenizers and vocabularies instead of character- or word-based tokenization.

Let's see how we can build our own tokenizer optimized for Python code.

A Tokenizer for Python

We need a custom tokenizer for our use case: tokenizing Python code. The question of pretokenization merits some discussion for programming languages. If we split on whitespaces and remove them, we will lose all the indentation information, which in Python is important for the semantics of the program (just think about `while` loops, or `if-then-else` statements). On the other hand, line breaks are not meaningful and can be added or removed without impact on the semantics. Similarly, splitting on punctuation, like an underscore, which is used to compose a single variable name from several subparts, might not make as much sense as it would in natural language. Using a natural language pretokenizer for tokenizing code thus seems potentially suboptimal.

Let's see if there are any tokenizers in the collection provided on the Hub that might be useful to us. We want a tokenizer that preserves spaces, so a good candidate could be a byte-level tokenizer like the one from GPT-2. Let's load this tokenizer and explore its tokenization properties:

```
from transformers import AutoTokenizer

python_code = r"""def say_hello():
    print("Hello, World!")

# Print it
say_hello()
"""
tokenizer = AutoTokenizer.from_pretrained("gpt2")
print(tokenizer(python_code).tokens())
```

```
['def', 'Ġsay', '_', 'hello', '():', 'Ċ', 'Ġ', 'Ġ', 'Ġ', 'Ġprint', '("',
'Hello', ',', 'ĠWorld', '!"', ')', 'Ġ#', 'ĠPrint', 'Ġit', 'Ċ', 'Ċ', 'say', '_',
'hello', '()', 'Ċ']
```

Python has a built-in `tokenize` module that splits Python code strings into meaningful units (code operation, comments, indent and dedent, etc.). One issue with using this approach is that this pretokenizer is Python-based and as such is typically rather slow and limited by the Python global interpreter lock (GIL). On the other hand, most of the tokenizers in the 🤗 Transformers library are provided by the 🤗 Tokenizers library and are coded in Rust. The Rust tokenizers are many orders of magnitude faster to train and to use, and we will thus likely want to use them given the size of our corpus.

This is quite a strange output, so let's try to understand what is happening here by running the various submodules of the tokenizer's pipeline. First let's see what normalization is applied in this tokenizer:

```
print(tokenizer.backend_tokenizer.normalizer)
```

```
None
```

As we can see, the GPT-2 tokenizer uses no normalization. It works directly on the raw Unicode inputs without any normalization steps. Let's now take a look at the pretokenization:

```
print(tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str(python_code))
```

```
[('def', (0, 3)), ('Ġsay', (3, 7)), ('_', (7, 8)), ('hello', (8, 13)), ('():',
(13, 16)), ('ĊĠĠĠ', (16, 20)), ('Ġprint', (20, 26)), ('("', (26, 28)), ('Hello',
(28, 33)), (',', (33, 34)), ('ĠWorld', (34, 40)), ('!")', (40, 43)), ('Ġ#', (43,
45)), ('ĠPrint', (45, 51)), ('Ġit', (51, 54)), ('Ċ', (54, 55)), ('Ċ', (55, 56)),
('say', (56, 59)), ('_', (59, 60)), ('hello', (60, 65)), ('()', (65, 67)), ('Ċ',
(67, 68))]
```

What are all these Ġ symbols, and what are the numbers accompanying the tokens? Let's explain both and see if we can understand better how this tokenizer works.

Let's start with the numbers. 🤗 Tokenizers has a very useful feature for switching between strings and tokens, called *offset tracking*. All the operations on the input string are tracked so that it's possible to know exactly what part of the input string a token after tokenization corresponds to. These numbers simply indicate where in the original string each token comes from; for instance, the word `'hello'` in the first line corresponds to the characters 8 to 13 in the original string. If some characters are removed in a normalization step, we are thus still able to associate each token with the respective part in the original string.

The other curious feature of the tokenized text is the odd-looking characters, such as Ċ and Ġ. *Byte-level* means that this tokenizer works on bytes instead of Unicode characters. Each Unicode character is composed of between 1 and 4 bytes, depending on the character. The nice thing about bytes is that while there are 143,859 Unicode characters in the Unicode alphabet, there are only 256 elements in the byte alphabet,

and you can express each Unicode character as a sequence of these bytes. If we work on bytes we can thus express all the strings composed from the UTF-8 world as longer strings in this alphabet of 256 values. That is, we can have a model using an alphabet of only 256 words and be able to process any Unicode string. Let's have a look at what the byte representations of some characters look like:

```
a, e = u"a", u"€"
byte = ord(a.encode("utf-8"))
print(f'`{a}` is encoded as `{a.encode("utf-8")}` with a single byte: {byte}')
byte = [ord(chr(i)) for i in e.encode("utf-8")]
print(f'`{e}` is encoded as `{e.encode("utf-8")}` with three bytes: {byte}')
```

```
`a` is encoded as `b'a'` with a single byte: 97
`€` is encoded as `b'\xe2\x82\xac'` with three bytes: [226, 130, 172]
```

At this point you might wonder: why work on a byte level? Think back to our discussion in Chapter 2 about the trade-offs between character and word tokens. We could decide to build our vocabulary from the 143,859 Unicode characters, but we would also like to include words—i.e., combinations of Unicode characters—in our vocabulary, so this (already very large) size is only a lower bound for the total size of the vocabulary. This will make our model's embedding layer very large because it comprises one vector for each vocabulary token.

On the other extreme, if we only use the 256 byte values as our vocabulary, the input sequences will be segmented in many small pieces (each byte constituting the Unicode characters), and as such our model will have to work on long inputs and spend significant compute power on reconstructing Unicode characters from their separate bytes, and then words from these characters. See the paper accompanying the ByT5 model release for a detailed study of this overhead.[6]

A middle-ground solution is to construct a medium-sized vocabulary by extending the 256-word vocabulary with the most common combinations of bytes. This is the approach taken by the BPE algorithm. The idea is to progressively construct a vocabulary of a predefined size by creating new vocabulary tokens through iteratively merging the most frequently co-occurring pair of tokens in the vocabulary. For instance, if `t` and `h` occur very frequently together, like in English, we'll add a token `th` to the vocabulary to model this pair of tokens instead of keeping them separated. The `t` and `h` tokens are kept in the vocabulary to tokenize instances where they do not occur together. Starting from a basic vocabulary of elementary units, we can then model any string efficiently.

6 L. Xue et al., "ByT5: Towards a Token-Free Future with Pre-Trained Byte-to-Byte Models", (2021).

Be careful not to confuse the "byte" in "Byte-Pair Encoding" with the "byte" in "byte-level." The name Byte-Pair Encoding comes from a data compression technique proposed by Philip Gage in 1994, originally operating on bytes.[7] Unlike what this name might indicate, standard BPE algorithms in NLP typically operate on Unicode strings rather than bytes (although there is a new type of BPE that specifically works on bytes, called *byte-level BPE*). If we read our Unicode strings in bytes we can thus reuse a simple BPE subword splitting algorithm.

There is just one issue when using a typical BPE algorithm in NLP. These algorithms are designed to work with clean Unicode string as inputs, not bytes, and expect regular ASCII characters in the inputs, without spaces or control characters. But in the Unicode characters corresponding to the 256 first bytes, there are many control characters (newline, tab, escape, line feed, and other nonprintable characters). To overcome this problem, the GPT-2 tokenizer first maps all the 256 input bytes to Unicode strings that can easily be digested by the standard BPE algorithms—that is, we will map our 256 elementary values to Unicode strings that all correspond to standard printable Unicode characters.

It's not very important that these Unicode characters are each encoded with 1 byte or more; what is important is that we have 256 single values at the end, forming our base vocabulary, and that these 256 values are correctly handled by our BPE algorithm. Let's see some examples of this mapping with the GPT-2 tokenizer. We can access the entire mapping as follows:

```
from transformers.models.gpt2.tokenization_gpt2 import bytes_to_unicode

byte_to_unicode_map = bytes_to_unicode()
unicode_to_byte_map = dict((v, k) for k, v in byte_to_unicode_map.items())
base_vocab = list(unicode_to_byte_map.keys())

print(f'Size of our base vocabulary: {len(base_vocab)}')
print(f'First element: `{base_vocab[0]}`, last element: `{base_vocab[-1]}`')
```

```
Size of our base vocabulary: 256
First element: `!`, last element: `Ń`
```

And we can take a look at some common values of bytes and associated mapped Unicode characters in Table 10-1.

7 P. Gage, "A New Algorithm for Data Compression," *The C Users Journal* 12, no. 2 (1994): 23–38, *https://dx.doi.org/10.14569/IJACSA.2012.030803*.

Table 10-1. Examples of character mappings in BPE

Description	Character	Bytes	Mapped bytes
Regular characters	`a` and `?`	97 and 63	`a` and `?`
A nonprintable control character (carriage return)	`U+000D`	13	`č`
A space	` `	32	`Ġ`
A nonbreakable space	`\xa0`	160	`ł`
A newline character	`\n`	10	`Ċ`

We could have used a more explicit conversion, like mapping newlines to a `NEWLINE` string, but BPE algorithms are typically designed to work on characters. For this reason, keeping one Unicode character for each byte character is easier to handle with an out-of-the-box BPE algorithm. Now that we have been introduced to the dark magic of Unicode encodings, we can understand our tokenization conversion a bit better:

```
print(tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str(python_code))
```

```
[('def', (0, 3)), ('Ġsay', (3, 7)), ('_', (7, 8)), ('hello', (8, 13)), ('():',
(13, 16)), ('ĊĠĠĠ', (16, 20)), ('Ġprint', (20, 26)), ('("', (26, 28)), ('Hello',
(28, 33)), (',', (33, 34)), ('ĠWorld', (34, 40)), ('!")', (40, 43)), ('Ġ#', (43,
45)), ('ĠPrint', (45, 51)), ('Ġit', (51, 54)), ('Ċ', (54, 55)), ('Ċ', (55, 56)),
('say', (56, 59)), ('_', (59, 60)), ('hello', (60, 65)), ('()', (65, 67)), ('Ċ',
(67, 68))]
```

We can recognize the newlines, which as we now know are mapped to Ċ, and the spaces, mapped to Ġ. We also see that:

- Spaces, and in particular consecutive spaces, are conserved (for instance, the three spaces in `'ĊĠĠĠ'`).
- Consecutive spaces are considered as a single word.
- Each space preceding a word is attached to and considered a part of the subsequent word (e.g., in `'Ġsay'`).

Let's now experiment with the BPE model. As we've mentioned, it's in charge of splitting the words into subunits until all subunits belong to the predefined vocabulary.

The vocabulary of our GPT-2 tokenizer comprises 50,257 words:

- The base vocabulary with the 256 values of the bytes
- 50,000 additional tokens created by repeatedly merging the most commonly co-occurring tokens
- A special character added to the vocabulary to represent document boundaries

We can easily check that by looking at the length attribute of the tokenizer:

```
print(f"Size of the vocabulary: {len(tokenizer)}")
```

```
Size of the vocabulary: 50257
```

Running the full pipeline on our input code gives us the following output:

```
print(tokenizer(python_code).tokens())
```

```
['def', 'Ġsay', '_', 'hello', '():', 'Ċ', 'Ġ', 'Ġ', 'Ġ', 'Ġprint', '("',
'Hello', ',', 'ĠWorld', '!"', ')', 'Ġ#', 'ĠPrint', 'Ġit', 'Ċ', 'Ċ', 'say', '_',
'hello', '()', 'Ċ']
```

As we can see, the BPE tokenizer keeps most of the words but will split the multiple spaces of our indentation into several consecutive spaces. This happens because this tokenizer is not specifically trained on code, but mostly on texts where consecutive spaces are rare. The BPE model thus doesn't include a specific token in the vocabulary for indentation. This is a case where the tokenizer model is poorly suited for the dataset's domain. As we discussed earlier, the solution is to retrain the tokenizer on the target corpus. So let's get to it!

Training a Tokenizer

Let's retrain our byte-level BPE tokenizer on a slice of our corpus to get a vocabulary better adapted to Python code. Retraining a tokenizer provided by 🤗 Transformers is simple. We just need to:

- Specify our target vocabulary size.
- Prepare an iterator to supply lists of input strings to process to train the tokenizer's model.
- Call the `train_new_from_iterator()` method.

Unlike deep learning models, which are often expected to memorize a lot of specific details from the training corpus, tokenizers are really just trained to extract the main statistics. In a nutshell, the tokenizer is just trained to know which letter combinations are the most frequent in our corpus.

Therefore, you don't necessarily need to train your tokenizer on a very large corpus; the corpus just needs to be representative of your domain and big enough for the tokenizer to extract statistically significant measures. But depending on the vocabulary size and the exact texts in the corpus, the tokenizer can end up storing unexpected words. We can see this, for instance, when looking at the longest words in the vocabulary of the GPT-2 tokenizer:

```
tokens = sorted(tokenizer.vocab.items(), key=lambda x: len(x[0]), reverse=True)
print([f'{tokenizer.convert_tokens_to_string(t)}' for t, _ in tokens[:8]]);
```

```
['ÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂÃÂ', '
================================================================', '
----------------------------------------------------------------
',
```

```
'.............................................................',
'ĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂĂ',
'
-------------------------------------------------------------
',
'=============================================================',
'_____________________________________________________________']
```

These tokens look like separator lines that are likely to be used on forums. This makes sense since GPT-2 was trained on a corpus centered around Reddit. Now let's have a look at the last words that were added to the vocabulary, and thus the least frequent ones:

```
tokens = sorted(tokenizer.vocab.items(), key=lambda x: x[1], reverse=True)
print([f'{tokenizer.convert_tokens_to_string(t)}' for t, _ in tokens[:12]]);
```

```
['<|endoftext|>', ' gazed', ' informants', ' Collider', ' regress', 'ominated',
' amplification', 'Compar', '..."', ' (/', 'Commission', ' Hitman']
```

The first token, `<|endoftext|>`, is the special token used to specify the end of a text sequence and was added after the BPE vocabulary was built. For each of these tokens our model will have to learn an associated word embedding, and we probably don't want the embedding matrix to contain too many noisy words. Also note how some very time- and space-specific knowledge of the world (e.g., proper nouns like `Hitman` and `Commission`) is embedded at a very low level in our modeling approach by these words being granted separate tokens with associated vectors in the vocabulary. The creation of such specific tokens by a BPE tokenizer can also be an indication that the target vocabulary size is too large or that the corpus contains idiosyncratic tokens.

Let's train a fresh tokenizer on our corpus and examine its learned vocabulary. Since we just need a corpus reasonably representative of our dataset statistics, let's select about 1–2 GB of data, or about 100,000 documents from our corpus:

```
from tqdm.auto import tqdm

length = 100000
dataset_name = 'transformersbook/codeparrot-train'
dataset = load_dataset(dataset_name, split="train", streaming=True)
iter_dataset = iter(dataset)

def batch_iterator(batch_size=10):
    for _ in tqdm(range(0, length, batch_size)):
        yield [next(iter_dataset)['content'] for _ in range(batch_size)]

new_tokenizer = tokenizer.train_new_from_iterator(batch_iterator(),
                                                  vocab_size=12500,
                                                  initial_alphabet=base_vocab)
```

Let's investigate the first and last words created by our BPE algorithm to see how relevant our vocabulary is. We skip the 256 byte tokens and look at the first tokens added thereafter:

```
tokens = sorted(new_tokenizer.vocab.items(), key=lambda x: x[1], reverse=False)
print([f'{tokenizer.convert_tokens_to_string(t)}' for t, _ in tokens[257:280]]);
```

```
[' ', '    ', '   ', '        ', 'se', 'in', '       ', 're', 'on', 'te', '\n
', '\n        ', 'or', 'st', 'de', '\n   ', 'th', 'le', ' =', 'lf', 'self',
'me', 'al']
```

Here we can see various standard levels of indentation and whitespace tokens, as well as short common Python keywords like `self`, `or`, and `in`. This is a good sign that our BPE algorithm is working as intended. Now let's check out the last words:

```
print([f'{new_tokenizer.convert_tokens_to_string(t)}' for t,_ in tokens[-12:]]);
```

```
[' capt', ' embedded', ' regarding', 'Bundle', '355', ' recv', ' dmp', ' vault',
' Mongo', ' possibly', 'implementation', 'Matches']
```

Here there are still some relatively common words, like `recv`, as well as some more noisy words probably coming from the comments.

We can also tokenize our simple example of Python code to see how our tokenizer is behaving on a simple example:

```
print(new_tokenizer(python_code).tokens())
```

```
['def', 'Ġs', 'ay', '_', 'hello', '():', 'ĊĠĠĠ', 'Ġprint', '("', 'Hello', ',',
'ĠWor', 'ld', '!")', 'Ġ#', 'ĠPrint', 'Ġit', 'Ċ', 'Ċ', 's', 'ay', '_', 'hello',
'()', 'Ċ']
```

Even though they are not code keywords, it's a little annoying to see common English words like `World` or `say` being split by our tokenizer, since we'd expect them to occur rather frequently in the corpus. Let's check if all the Python reserved keywords are in the vocabulary:

```
import keyword

print(f'There are in total {len(keyword.kwlist)} Python keywords.')
for keyw in keyword.kwlist:
    if keyw not in new_tokenizer.vocab:
        print(f'No, keyword `{keyw}` is not in the vocabulary')
```

```
There are in total 35 Python keywords.
No, keyword `await` is not in the vocabulary
No, keyword `finally` is not in the vocabulary
No, keyword `nonlocal` is not in the vocabulary
```

It appears that several quite frequent keywords, like `finally`, are not in the vocabulary either. Let's try building a larger vocabulary using a larger sample of our dataset. For instance, we can build a vocabulary of 32,768 words (multiples of 8 are better for some efficient GPU/TPU computations) and train the tokenizer on a twice as large slice of our corpus:

```
length = 200000
new_tokenizer_larger = tokenizer.train_new_from_iterator(batch_iterator(),
    vocab_size=32768, initial_alphabet=base_vocab)
```

We don't expect the most frequent tokens to change much when adding more documents, but let's look at the last tokens:

```
tokens = sorted(new_tokenizer_larger.vocab.items(), key=lambda x: x[1],
                reverse=False)
print([f'{tokenizer.convert_tokens_to_string(t)}' for t, _ in tokens[-12:]]);
```

```
['lineEdit', 'spik', ' BC', 'pective', 'OTA', 'theus', 'FLUSH', ' excutils',
'00000002', ' DIVISION', 'CursorPosition', ' InfoBar']
```

A brief inspection doesn't show any regular programming keywords here, which is promising. Let's try tokenizing our sample code example with the new larger tokenizer:

```
print(new_tokenizer_larger(python_code).tokens())
```

```
['def', 'Ġsay', '_', 'hello', '():', 'ĊĠĠĠ', 'Ġprint', '("', 'Hello', ',',
'ĠWorld', '!")', 'Ġ#', 'ĠPrint', 'Ġit', 'Ċ', 'Ċ', 'say', '_', 'hello', '()',
'Ċ']
```

Here also the indents are conveniently kept in the vocabulary, and we see that common English words like `Hello`, `World`, and `say` are also included as single tokens. This seems more in line with our expectations of the data the model may see in the downstream task. Let's investigate the common Python keywords, as we did before:

```
for keyw in keyword.kwlist:
    if keyw not in new_tokenizer_larger.vocab:
        print(f'No, keyword `{keyw}` is not in the vocabulary')
```

```
No, keyword `nonlocal` is not in the vocabulary
```

We are still missing the nonlocal keyword, but it's also rarely used in practice as it makes the syntax more complex. Keeping it out of the vocabulary seems reasonable. After this manual inspection, our larger tokenizer seems well adapted for our task—but as we mentioned earlier, objectively evaluating the performance of a tokenizer is a challenging task without measuring the model's performance. We will proceed with this one and train a model to see how well it works in practice.

You can easily verify that the new tokenizer is about twice as efficient than the standard GPT-2 tokenizer by comparing the sequence lengths of tokenized code examples. Our tokenizer uses approximately half as many tokens as the existing one to encode a text, which gives us twice the effective model context for free. When we train a new model with the new tokenizer on a context window of size 1,024 it is equivalent to training the same model with the old tokenizer on a context window of size 2,048, with the advantage of being much faster and more memory efficient.

Saving a Custom Tokenizer on the Hub

Now that our tokenizer is trained, we should save it. The simplest way to save it and be able to access it from anywhere later is to push it to the Hugging Face Hub. This will be especially useful later, when we use a separate training server.

To create a private model repository and save our tokenizer in it as a first file, we can directly use the `push_to_hub()` method of the tokenizer. Since we already authenticated our account with `huggingface-cli login`, we can simply push the tokenizer as follows:

```
model_ckpt = "codeparrot"
org = "transformersbook"
new_tokenizer_larger.push_to_hub(model_ckpt, organization=org)
```

If you don't want to push to an organization, you can simply omit the `organization` argument. This will create a repository in your namespace named `codeparrot`, which anyone can then load by running:

```
reloaded_tokenizer = AutoTokenizer.from_pretrained(org + "/" + model_ckpt)
print(reloaded_tokenizer(python_code).tokens())
```

```
['def', 'Ġsay', '_', 'hello', '():', 'ĊĠĠĠ', 'Ġprint', '("', 'Hello', ',',
'ĠWorld', '!")', 'Ġ#', 'ĠPrint', 'Ġit', 'Ċ', 'Ċ', 'say', '_', 'hello', '()',
'Ċ']
```

The tokenizer loaded from the Hub behaves exactly as we just saw. We can also investigate its files and saved vocabulary on the Hub. For reproducibility, let's save our smaller tokenizer as well:

```
new_tokenizer.push_to_hub(model_ckpt+ "-small-vocabulary", organization=org)
```

This was a deep dive into building a tokenizer for a specific use case. Next, we will finally create a new model and train it from scratch.

Training a Model from Scratch

Here's the part you've probably been waiting for: the model training. In this section we'll decide which architecture works best for the task, initialize a fresh model without pretrained weights, set up a custom data loading class, and create a scalable training loop. In the grand finale we will train small and large GPT-2 models with 111 million and 1.5 billion parameters, respectively! But let's not get ahead ourselves. First, we need to decide which architecture is best suited for code autocompletion.

In this section we will implement a longer than usual script to train a model on a distributed infrastructure. Therefore, you should not run each code snippet independently, but instead download the script provided in the 🤗 Transformers repository. Follow the accompanying instructions to execute the script with 🤗 Accelerate on your hardware.

A Tale of Pretraining Objectives

Now that we have access to a large-scale pretraining corpus and an efficient tokenizer, we can start thinking about how to pretrain a transformer model. With such a large codebase consisting of code snippets like the one shown in Figure 10-1, we can tackle several tasks. Which one we choose will influence our choice of pretraining objectives. Let's have a look at three common tasks.

Example from corpus

```
def add_numbers(a,b):
    "add two numbers"
    return a+b
```

Figure 10-1. An example of a Python function that could be found in our dataset

Causal language modeling

A natural task with textual data is to provide a model with the beginning of a code sample and ask it to generate possible completions. This is a self-supervised training objective in which we can use the dataset without annotations. This should ring a bell: it's the *causal language modeling* task we encountered in Chapter 5. A directly related downstream task is code autocompletion, so we'll definitely put this model on the shortlist. A decoder-only architecture such as the GPT family of models is usually best suited for this task, as shown in Figure 10-2.

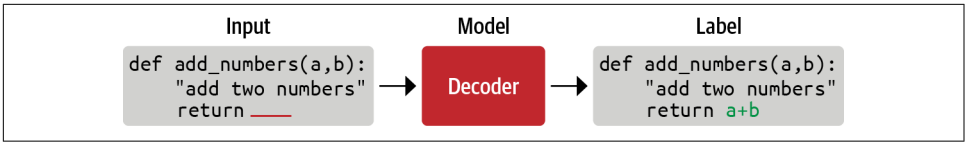

Figure 10-2. In causal language modeling, the future tokens are masked and the model has to predict them; typically a decoder model such as GPT is used for such a task

Masked language modeling

A related but slightly different task is to provide a model with a noisy code sample, for instance with a code instruction replaced by a random or masked word, and ask it to reconstruct the original clean sample, as illustrated in Figure 10-3. This is also a self-supervised training objective and is commonly called *masked language modeling* or the *denoising objective*. It's harder to think about a downstream task directly related to denoising, but denoising is generally a good pretraining task to learn general representations for later downstream tasks. Many of the models that we have used in the previous chapters (like BERT and XLM-RoBERTa) are pretrained in that way. Training a masked language model on a large corpus can thus be combined with fine-tuning the model on a downstream task with a limited number of labeled examples.

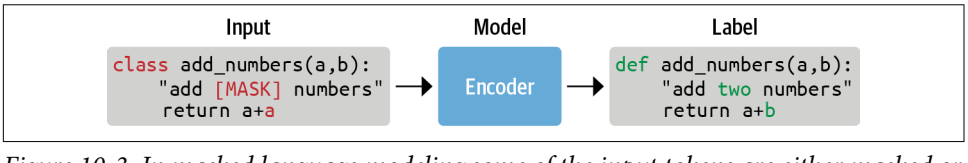

Figure 10-3. In masked language modeling some of the input tokens are either masked or replaced, and the model's task is to predict the original tokens; this is the architecture underlying the encoder branch of transformer models

Sequence-to-sequence training

An alternative task is to use a heuristic like regular expressions to separate comments or docstrings from code and build a large-scale dataset of (code, comments) pairs that can be used as an annotated dataset. The training task is then a supervised training objective in which one category (code or comment) is used as input for the model and the other category (comment or code) is used as labels. This is a case of *supervised learning* with (input, labels) pairs, as highlighted in Figure 10-4. With a large, clean, and diverse dataset as well as a model with sufficient capacity, we can try to train a model that learns to transcript comments in code or vice versa. A downstream task directly related to this supervised training task is then documentation generation from code or code generation from documentation, depending on how we set our input/outputs. In this setting a sequence is translated into another sequence, which is where encoder-decoder architectures such as T5, BART, and PEGASUS shine.

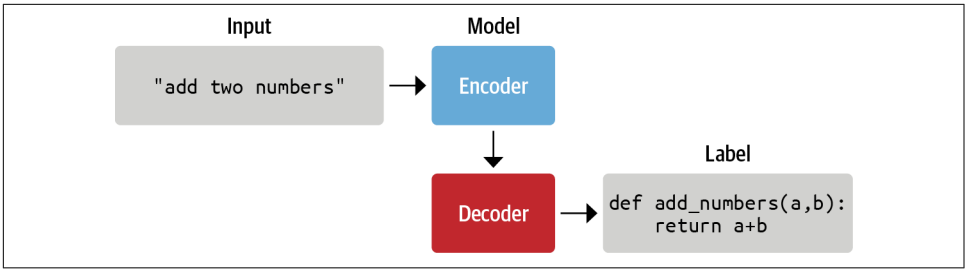

Figure 10-4. Using an encoder-decoder architecture for a sequence-to-sequence task where the inputs are split into comment/code pairs using heuristics: the model gets one element as input and needs to generate the other one

Since we want to build a code autocompletion model, we'll select the first objective and choose a GPT architecture for the task. So let's initialize a fresh GPT-2 model!

Initializing the Model

This is the first time in this book that we won't use the `from_pretrained()` method to load a model but initialize the new model. We will, however, load the configuration of `gpt2-xl` so that we use the same hyperparameters and only adapt the vocabulary size for the new tokenizer. We then initialize a new model with this configuration with the `from_config()` method:

```
from transformers import AutoConfig, AutoModelForCausalLM, AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
config = AutoConfig.from_pretrained("gpt2-xl", vocab_size=len(tokenizer))
model = AutoModelForCausalLM.from_config(config)
```

Let's check how large the model actually is:

```
print(f'GPT-2 (xl) size: {model_size(model)/1000**2:.1f}M parameters')
```

```
GPT-2 (xl) size: 1529.6M parameters
```

This is a 1.5B parameter model! This is a lot of capacity, but we also have a large dataset. In general, large language models are more efficient to train as long as the dataset is reasonably large. Let's save the newly initialized model in a *models/* folder and push it to the Hub:

```
model.save_pretrained("models/" + model_ckpt, push_to_hub=True,
                      organization=org)
```

Pushing the model to the Hub may take a few minutes given the size of the checkpoint (> 5 GB). Since this model is quite large, we'll also create a smaller version that we can train to make sure everything works before scaling up. We will take the standard GPT-2 size as a base:

```
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
config_small = AutoConfig.from_pretrained("gpt2", vocab_size=len(tokenizer))
model_small = AutoModelForCausalLM.from_config(config_small)

print(f'GPT-2 size: {model_size(model_small)/1000**2:.1f}M parameters')

GPT-2 size: 111.0M parameters
```

And let's save it to the Hub as well for easy sharing and reuse:

```
model_small.save_pretrained("models/" + model_ckpt + "-small", push_to_hub=True,
                            organization=org)
```

Now that we have two models we can train, we need to make sure we can feed them the input data efficiently during training.

Implementing the Dataloader

To be able to train with maximal efficiency, we will want to supply our model with sequences filling its context. For example, if the context length of our model is 1,024 tokens, we always want to provide 1,024-token sequences during training. But some of our code examples might be shorter or longer than 1,024 tokens. To feed batches with full sequences of `sequence_length` to our model, we should thus either drop the last incomplete sequence or pad it. However, this will render our training slightly less efficient and force us to take care of padding and masking padded token labels. We are much more compute- than data-constrained, so we'll take the easy and efficient way here. We can use a little trick to make sure we don't lose too many trailing segments: we can tokenize several examples and then concatenate them, separated by the special end-of-sequence token, to get a very long sequence. Finally, we split this sequence into equally sized chunks as shown in Figure 10-5. With this approach, we lose at most a small fraction of the data at the end.

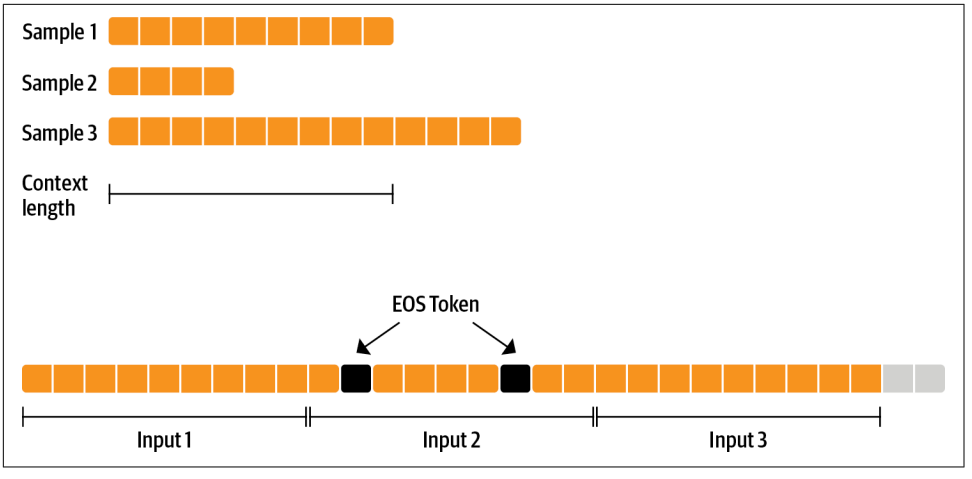

Figure 10-5. Preparing sequences of varying length for causal language modeling by concatenating several tokenized examples with an EOS token before chunking them

We can, for instance, make sure we have roughly one hundred full sequences in our tokenized examples by defining our input string character length as:

```
input_characters = number_of_sequences * sequence_length * characters_per_token
```

where:

- `input_characters` is the number of characters in the string input to our tokenizer.
- `number_of_sequences` is the number of (truncated) sequences we would like from our tokenizer, (e.g., 100).
- `sequence_length` is the number of tokens per sequence returned by the tokenizer, (e.g., 1,024).
- `characters_per_token` is the average number of characters per output token that we first need to estimate.

If we input a string with `input_characters` characters we will thus get on average `number_of_sequences` output sequences, and we can easily calculate how much input data we are losing by dropping the last sequence. If `number_of_sequences=100` it means that we stack roughly 100 sequences and at most lose the last element, which might be too short or too long. This corresponds to at most losing 1% of our dataset. At the same time, this approach ensures that we don't introduce a bias by cutting off the majority of file endings.

Let's first estimate the average character length per token in our dataset:

```
examples, total_characters, total_tokens = 500, 0, 0
dataset = load_dataset('transformersbook/codeparrot-train', split='train',
                       streaming=True)

for _, example in tqdm(zip(range(examples), iter(dataset)), total=examples):
    total_characters += len(example['content'])
    total_tokens += len(tokenizer(example['content']).tokens())

characters_per_token = total_characters / total_tokens

print(characters_per_token)
```

```
3.6233025034779565
```

With that we have all that's needed to create our own `IterableDataset` (which is a helper class provided by PyTorch) for preparing constant-length inputs for the model. We just need to inherit from `IterableDataset` and set up the `__iter__()` function that yields the next element with the logic we just walked through:

```
import torch
from torch.utils.data import IterableDataset

class ConstantLengthDataset(IterableDataset):

    def __init__(self, tokenizer, dataset, seq_length=1024,
                 num_of_sequences=1024, chars_per_token=3.6):
        self.tokenizer = tokenizer
        self.concat_token_id = tokenizer.eos_token_id
        self.dataset = dataset
        self.seq_length = seq_length
        self.input_characters = seq_length * chars_per_token * num_of_sequences

    def __iter__(self):
        iterator = iter(self.dataset)
        more_examples = True
        while more_examples:
            buffer, buffer_len = [], 0
            while True:
                if buffer_len >= self.input_characters:
                    m=f"Buffer full: {buffer_len}>={self.input_characters:.0f}"
                    print(m)
                    break
                try:
                    m=f"Fill buffer: {buffer_len}<{self.input_characters:.0f}"
                    print(m)
                    buffer.append(next(iterator)["content"])
                    buffer_len += len(buffer[-1])
                except StopIteration:
                    iterator = iter(self.dataset)

            all_token_ids = []
```

```
            tokenized_inputs = self.tokenizer(buffer, truncation=False)
            for tokenized_input in tokenized_inputs["input_ids'"]:
            for tokenized_input in tokenized_inputs:
                all_token_ids.extend(tokenized_input + [self.concat_token_id])

            for i in range(0, len(all_token_ids), self.seq_length):
                input_ids = all_token_ids[i : i + self.seq_length]
                if len(input_ids) == self.seq_length:
                    yield torch.tensor(input_ids)
```

The `__iter__()` function builds up a buffer of strings until it contains enough characters. All the elements in the buffer are tokenized and concatenated with the EOS token, then the long sequence in `all_token_ids` is chunked in `seq_length`-sized slices. Normally, we need attention masks to stack padded sequences of varying length and make sure the padding is ignored during training. We have taken care of this by only providing sequences of the same (maximal) length, so we don't need the masks here and only return the `input_ids`. Let's test our iterable dataset:

```
shuffled_dataset = dataset.shuffle(buffer_size=100)
constant_length_dataset = ConstantLengthDataset(tokenizer, shuffled_dataset,
                                                 num_of_sequences=10)
dataset_iterator = iter(constant_length_dataset)

lengths = [len(b) for _, b in zip(range(5), dataset_iterator)]
print(f"Lengths of the sequences: {lengths}")
```

```
Fill buffer: 0<36864
Fill buffer: 3311<36864
Fill buffer: 9590<36864
Fill buffer: 22177<36864
Fill buffer: 25530<36864
Fill buffer: 31098<36864
Fill buffer: 32232<36864
Fill buffer: 33867<36864
Buffer full: 41172>=36864
Lengths of the sequences: [1024, 1024, 1024, 1024, 1024]
```

Nice, this works as intended and we get constant-length inputs for the model. Now that we have a reliable data source for the model, it's time to build the actual training loop.

Notice that we shuffled the raw dataset before creating a `ConstantLengthDataset`. Since this is an iterable dataset, we can't just shuffle the whole dataset at the beginning. Instead, we set up a buffer with size `buffer_size` and shuffle the elements in this buffer before we get elements from the dataset.

Defining the Training Loop

We now have all the elements to write our training loop. One obvious limitation of training our own language model is the memory limits on the GPUs we will use. Even on a modern graphics card you can't train a model at GPT-2 scale in reasonable time. In this tutorial we will implement *data parallelism*, which will help us utilize several GPUs for training. Fortunately, we can use 🤗 Accelerate to make our code scalable. The 🤗 Accelerate library is designed to make distributed training—and changing the underlying hardware for training—easy. We can also use the `Trainer` for distributed training but 🤗 Accelerate gives us full control over the training loop, which is what we want to explore here.

🤗 Accelerate provides an easy API to make training scripts run with mixed precision and in any kind of distributed setting (single GPU, multiple GPUs, and TPUs). The same code can then run seamlessly on your local machine for debugging purposes or your beefy training cluster for the final training run. You only need to make a handful of changes to your native PyTorch training loop:

```
  import torch
  import torch.nn.functional as F
  from datasets import load_dataset
+ from accelerate import Accelerator

- device = 'cpu'
+ accelerator = Accelerator()

- model = torch.nn.Transformer().to(device)
+ model = torch.nn.Transformer()
  optimizer = torch.optim.Adam(model.parameters())
  dataset = load_dataset('my_dataset')
  data = torch.utils.data.DataLoader(dataset, shuffle=True)
+ model, optimizer, data = accelerator.prepare(model, optimizer, data)

  model.train()
  for epoch in range(10):
      for source, targets in data:
-         source = source.to(device)
-         targets = targets.to(device)
          optimizer.zero_grad()
          output = model(source)
          loss = F.cross_entropy(output, targets)
-         loss.backward()
+         accelerator.backward(loss)
          optimizer.step()
```

The core part of the changes is the call to `prepare()`, which makes sure the model, optimizers, and dataloaders are all prepared and distributed on the infrastructure. These minor changes to the PyTorch training loop enable you to easily scale training across different infrastructures. With that in mind, let's start building up our training

script and define a few helper functions. First we set up the hyperparameters for training and wrap them in a `Namespace` for easy access:

```
from argparse import Namespace

# Commented parameters correspond to the small model
config = {"train_batch_size": 2, # 12
          "valid_batch_size": 2, # 12
          "weight_decay": 0.1,
          "shuffle_buffer": 1000,
          "learning_rate": 2e-4, # 5e-4
          "lr_scheduler_type": "cosine",
          "num_warmup_steps": 750, # 2000
          "gradient_accumulation_steps": 16, # 1
          "max_train_steps": 50000, # 150000
          "max_eval_steps": -1,
          "seq_length": 1024,
          "seed": 1,
          "save_checkpoint_steps": 50000} # 15000

args = Namespace(**config)
```

Next, we set up logging for training. Since we are training a model from scratch, the training run will take a while and require expensive infrastructure. Therefore, we want to make sure that all the relevant information is stored and easily accessible. The `setup_logging()` method sets up three levels of logging: using a standard Python `Logger`, TensorBoard, and Weights & Biases. Depending on your preferences and use case, you can add or remove logging frameworks here:

```
from torch.utils.tensorboard import SummaryWriter
import logging
import wandb

def setup_logging(project_name):
    logger = logging.getLogger(__name__)
    logging.basicConfig(
        format="%(asctime)s - %(levelname)s - %(name)s - %(message)s",
        datefmt="%m/%d/%Y %H:%M:%S", level=logging.INFO, handlers=[
        logging.FileHandler(f"log/debug_{accelerator.process_index}.log"),
        logging.StreamHandler()])
    if accelerator.is_main_process: # We only want to set up logging once
        wandb.init(project=project_name, config=args)
        run_name = wandb.run.name
        tb_writer = SummaryWriter()
        tb_writer.add_hparams(vars(args), {'0': 0})
        logger.setLevel(logging.INFO)
        datasets.utils.logging.set_verbosity_debug()
        transformers.utils.logging.set_verbosity_info()
    else:
        tb_writer = None
        run_name = ''
        logger.setLevel(logging.ERROR)
```

```
        datasets.utils.logging.set_verbosity_error()
        transformers.utils.logging.set_verbosity_error()
    return logger, tb_writer, run_name
```

Each worker gets a unique `accelerator.process_index`, which we use with the `FileHandler` to write the logs of each worker to an individual file. We also use the `accelerator.is_main_process` attribute, which is only `true` for the main worker. We make sure we don't initialize the TensorBoard and Weights & Biases loggers several times, and we decrease the logging levels for the other workers. We return the autogenerated, unique `wandb.run.name`, which we use later to name our experiment branch on the Hub.

We'll also define a function to log the metrics with TensorBoard and Weights & Biases. We again use the `accelerator.is_main_process` here to ensure that we only log the metrics once and not for each worker:

```
def log_metrics(step, metrics):
    logger.info(f"Step {step}: {metrics}")
    if accelerator.is_main_process:
        wandb.log(metrics)
        [tb_writer.add_scalar(k, v, step) for k, v in metrics.items()]
```

Next, let's write a function that creates the dataloaders for the training and validation sets with our brand new `ConstantLengthDataset` class:

```
from torch.utils.data.dataloader import DataLoader

def create_dataloaders(dataset_name):
    train_data = load_dataset(dataset_name+'-train', split="train",
                              streaming=True)
    train_data = train_data.shuffle(buffer_size=args.shuffle_buffer,
                                    seed=args.seed)
    valid_data = load_dataset(dataset_name+'-valid', split="validation",
                              streaming=True)

    train_dataset = ConstantLengthDataset(tokenizer, train_data,
                                          seq_length=args.seq_length)
    valid_dataset = ConstantLengthDataset(tokenizer, valid_data,
                                          seq_length=args.seq_length)

    train_dataloader=DataLoader(train_dataset, batch_size=args.train_batch_size)
    eval_dataloader=DataLoader(valid_dataset, batch_size=args.valid_batch_size)
    return train_dataloader, eval_dataloader
```

At the end we wrap the dataset in a `DataLoader`, which also handles the batching. 🤗 Accelerate will take care of distributing the batches to each worker.

Another aspect we need to implement is optimization. We will set up the optimizer and learning rate schedule in the main loop, but we define a helper function here to differentiate the parameters that should receive weight decay. In general, biases and LayerNorm weights are not subject to weight decay:

```
def get_grouped_params(model, no_decay=["bias", "LayerNorm.weight"]):
    params_with_wd, params_without_wd = [], []
    for n, p in model.named_parameters():
        if any(nd in n for nd in no_decay):
            params_without_wd.append(p)
        else:
            params_with_wd.append(p)
    return [{'params': params_with_wd, 'weight_decay': args.weight_decay},
            {'params': params_without_wd, 'weight_decay': 0.0}]
```

Finally, we want to evaluate the model on the validation set from time to time, so let's add an evaluation function we can call that calculates the loss and perplexity on the evaluation set:

```
def evaluate():
    model.eval()
    losses = []
    for step, batch in enumerate(eval_dataloader):
        with torch.no_grad():
            outputs = model(batch, labels=batch)
        loss = outputs.loss.repeat(args.valid_batch_size)
        losses.append(accelerator.gather(loss))
        if args.max_eval_steps > 0 and step >= args.max_eval_steps: break
    loss = torch.mean(torch.cat(losses))
    try:
        perplexity = torch.exp(loss)
    except OverflowError:
        perplexity = torch.tensor(float("inf"))
    return loss.item(), perplexity.item()
```

The perplexity measures how well the model's output probability distributions predict the targeted tokens. So a lower perplexity corresponds to a better performance. Note that we can compute the perplexity by exponentiating the cross-entropy loss which we get from the model's output. Especially at the start of training when the loss is still high, it is possible to get a numerical overflow when calculating the perplexity. We catch this error and set the perplexity to infinity in these instances.

Before we put it all together in the training script, there is one more additional function that we'll use. As you know by now, the Hugging Face Hub uses Git under the hood to store and version models and datasets. With the `Repository` class from the *huggingface_hub* library you can programmatically access the repository and pull, branch, commit, or push. We'll use this in our script to continuously push model checkpoints to the Hub during training.

Now that we have all these helper functions in place, we are ready to write the heart of the training script:

```
set_seed(args.seed)

# Accelerator
accelerator = Accelerator()
```

```
samples_per_step = accelerator.state.num_processes * args.train_batch_size

# Logging
logger, tb_writer, run_name = setup_logging(project_name.split("/")[1])
logger.info(accelerator.state)

# Load model and tokenizer
if accelerator.is_main_process:
    hf_repo = Repository("./", clone_from=project_name, revision=run_name)
model = AutoModelForCausalLM.from_pretrained("./", gradient_checkpointing=True)
tokenizer = AutoTokenizer.from_pretrained("./")

# Load dataset and dataloader
train_dataloader, eval_dataloader = create_dataloaders(dataset_name)

# Prepare the optimizer and learning rate scheduler
optimizer = AdamW(get_grouped_params(model), lr=args.learning_rate)
lr_scheduler = get_scheduler(name=args.lr_scheduler_type, optimizer=optimizer,
                             num_warmup_steps=args.num_warmup_steps,
                             num_training_steps=args.max_train_steps,)
def get_lr():
    return optimizer.param_groups[0]['lr']

# Prepare everything with our `accelerator` (order of args is not important)
model, optimizer, train_dataloader, eval_dataloader = accelerator.prepare(
    model, optimizer, train_dataloader, eval_dataloader)

# Train model
model.train()
completed_steps = 0
for step, batch in enumerate(train_dataloader, start=1):
    loss = model(batch, labels=batch).loss
    log_metrics(step, {'lr': get_lr(), 'samples': step*samples_per_step,
                       'steps': completed_steps, 'loss/train': loss.item()})
    loss = loss / args.gradient_accumulation_steps
    accelerator.backward(loss)
    if step % args.gradient_accumulation_steps == 0:
        optimizer.step()
        lr_scheduler.step()
        optimizer.zero_grad()
        completed_steps += 1
    if step % args.save_checkpoint_steps == 0:
        logger.info('Evaluating and saving model checkpoint')
        eval_loss, perplexity = evaluate()
        log_metrics(step, {'loss/eval': eval_loss, 'perplexity': perplexity})
        accelerator.wait_for_everyone()
        unwrapped_model = accelerator.unwrap_model(model)
        if accelerator.is_main_process:
            unwrapped_model.save_pretrained("./")
            hf_repo.push_to_hub(commit_message=f'step {step}')
        model.train()
    if completed_steps >= args.max_train_steps:
```

```
            break

    # Evaluate and save the last checkpoint
    logger.info('Evaluating and saving model after training')
    eval_loss, perplexity = evaluate()
    log_metrics(step, {'loss/eval': eval_loss, 'perplexity': perplexity})
    accelerator.wait_for_everyone()
    unwrapped_model = accelerator.unwrap_model(model)
    if accelerator.is_main_process:
        unwrapped_model.save_pretrained("./")
        hf_repo.push_to_hub(commit_message=f'final model')
```

This is quite a code block, but remember that this is all the code you need to train a fancy, large language model on a distributed infrastructure. Let's deconstruct the script a little bit and highlight the most important parts:

Model saving
: We run the script from within the model repository, and at the start we check out a new branch named after the `run_name` we get from Weights & Biases. Later, we commit the model at each checkpoint and push it to the Hub. With that setup each experiment is on a new branch and each commit represents a model checkpoint. Note that we need to call `wait_for_everyone()` and `unwrap_model()` to make sure the model is properly synchronized when we store it.

Optimization
: For the model optimization we use `AdamW` with a cosine learning rate schedule after a linear warming-up period. For the hyperparameters, we closely follow the parameters described in the GPT-3 paper for similar-sized models.[8]

Evaluation
: We evaluate the model on the evaluation set every time we save—that is, every `save_checkpoint_steps` and after training. Along with the validation loss we also log the validation perplexity.

Gradient accumulation and checkpointing
: The required batch sizes don't fit in a GPU's memory, even when we run on the latest GPUs. Therefore, we implement gradient accumulation, which gathers gradients over several backward passes and optimizes once enough gradients are accumulated. In Chapter 6, we saw how we can do this with the `Trainer`. For the large model, even a single batch does not quite fit on a single GPU. Using a method called *gradient checkpointing* we can trade some of the memory footprint

8 T. Brown et al., "Language Models Are Few-Shot Learners", (2020).

for an approximately 20% training slowdown.[9] This allows us to fit even the large model in a single GPU.

One aspect that might still be a bit obscure is what it means to train a model on multiple GPUs. There are several approaches to train models in a distributed fashion depending on the size of your model and volume of data. The approach utilized by 🤗 Accelerate is called `DataDistributedParallelism` (DDP). The main advantage of this approach is that it allows you to train models faster with larger batch sizes that wouldn't fit into any single GPU. The process is illustrated in Figure 10-6.

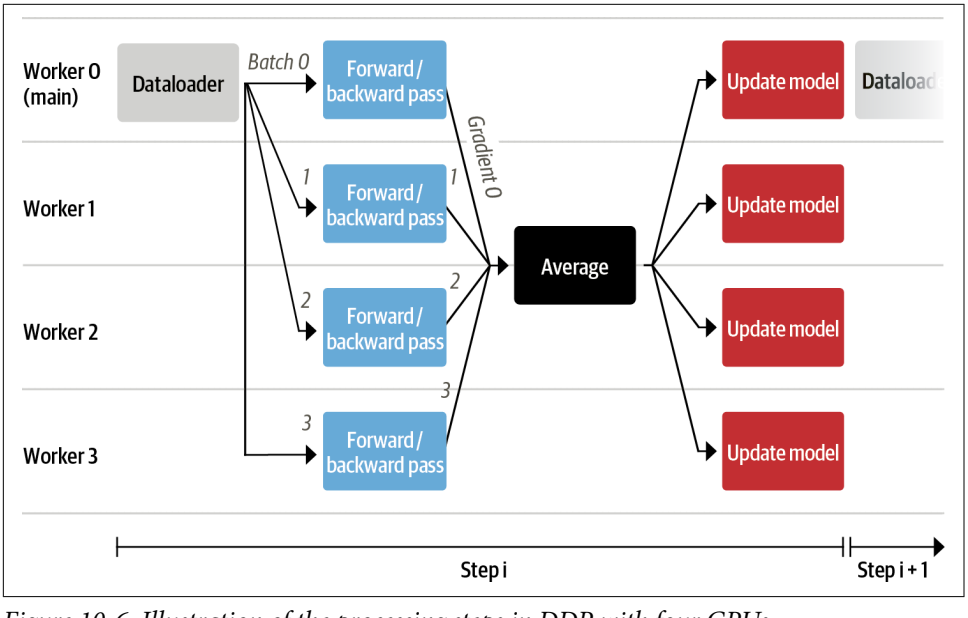

Figure 10-6. Illustration of the processing steps in DDP with four GPUs

Let's go through the pipeline step by step:

1. Each worker consists of a GPU. In 🤗 Accelerate, there is a dataloader running on the main process that prepares the batches of data and sends them to all the workers.
2. Each GPU receives a batch of data and calculates the loss and respective accumulated gradients from forward and backward passes with a local copy of the model.
3. The gradients from each node are averaged with a *reduce* pattern, and the averaged gradients are sent back to each worker.

9 You can read more about gradient checkpointing on OpenAI's release post.

4. The gradients are applied using the optimizer on each node individually. Although this might seem like redundant work, it avoids transferring copies of the large models between nodes. We'll need to update the model at least once, and without this approach the other nodes would each need to wait until they'd received the updated version.
5. Once all models are updated we start all over again, with the main worker preparing new batches.

This simple pattern allows us to train large models extremely fast by scaling up to the number of available GPUs without much additional logic. Sometimes, however, this is not enough. For example, if the model does not fit on a single GPU you might need more sophisticated parallelism strategies. Now that we have all the pieces needed for training, it's time to launch a job! As you'll see in the next section, this is quite simple to do.

The Training Run

We'll save the training script in a file called *codeparrot_training.py* so that we can execute it on our training server. To make life even easier, we'll add it along with a *requirements.txt* file containing all the required Python dependencies to the model repository on the Hub. Remember that the models on the Hub are essentially Git repositories so we can just clone the repository, add any files we want, and then push them back to the Hub. On the training server, we can then spin up training with the following handful of commands:

```
$ git clone https://huggingface.co/transformersbook/codeparrot
$ cd codeparrot
$ pip install -r requirements.txt
$ wandb login
$ accelerate config
$ accelerate launch codeparrot_training.py
```

And that's it—our model is now training! Note that `wandb login` will prompt you to authenticate with Weights & Biases for logging. The `accelerate config` command will guide you through setting up the infrastructure; you can see the settings used for this experiment in Table 10-2. We use an `a2-megagpu-16g` instance for all experiments, which is a workstation with 16 A100 GPUs with 40 GB of memory each.

Table 10-2. Configuration used to train the CodeParrot models

Setting	Value
Compute environment?	multi-GPU
How many machines?	1
DeepSpeed?	No
How many processes?	16
Use FP16?	Yes

Running the training script with these settings on that infrastructure takes about 24 hours and 7 days for the small and large models, respectively. If you train your own custom model, make sure your code runs smoothly on smaller infrastructure in order to make sure that expensive long run goes smoothly as well. After the full training run completes successfully, you can merge the experiment branch on the Hub back into the main branch with the following commands:

```
$ git checkout main
$ git merge <RUN_NAME>
$ git push
```

Naturally, *RUN_NAME* should be the name of the experiment branch on the Hub you would like to merge. Now that we have a trained model, let's have a look at how we can investigate its performance.

Results and Analysis

After anxiously monitoring the logs for a week, you will probably see loss and perplexity curves that look like those shown in Figure 10-7. The training loss and validation perplexity go down continuously, and the loss curve looks almost linear on the log-log scale. We also see that the large model converges faster in terms of processed tokens, although the overall training takes longer.

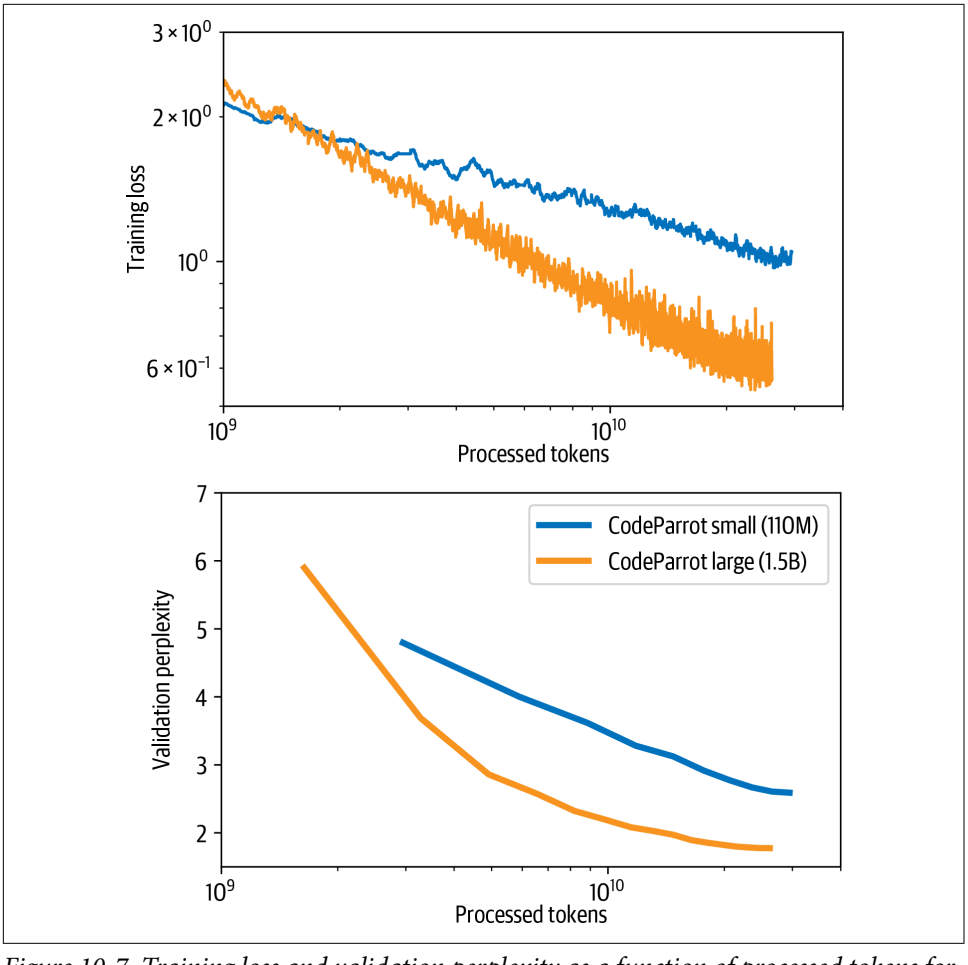

Figure 10-7. Training loss and validation perplexity as a function of processed tokens for the small and large CodeParrot models

So what can we do with our freshly baked language model, straight out of the GPU oven? Well, we can use it to write some code for us. There are two types of analyses we can conduct: qualitative and quantitative. In the former, we look at concrete examples and try to better understand in which cases the model succeeds and where it fails. In the latter case, we evaluate the model's performance statistically on a large set of test cases. In this section we'll explore how we can use our model. First we'll have a look at a few examples, and then we'll briefly discuss how we could evaluate the model systematically and more robustly. First, let's wrap the small model in a pipeline and use it to continue some code inputs:

```
from transformers import pipeline, set_seed
```

```
model_ckpt = 'transformersbook/codeparrot-small'
generation = pipeline('text-generation', model=model_ckpt, device=0)
```

Now we can use the generation pipeline to generate candidate completions from a given prompt. By default, the pipeline will generate code until a predefined maximum length, and the output could contain multiple functions or classes. So, to keep the outputs concise, we'll implement a `first_block()` function that uses regular expressions to extract the first occurrence of a function or class. The `complete_code()` function below applies this logic to print out the completions generated by CodeParrot:

```
import re
from transformers import set_seed

def first_block(string):
    return re.split('\nclass|\ndef|\n#|\n@|\nprint|\nif', string)[0].rstrip()

def complete_code(pipe, prompt, max_length=64, num_completions=4, seed=1):
    set_seed(seed)
    gen_kwargs = {"temperature":0.4, "top_p":0.95, "top_k":0, "num_beams":1,
                  "do_sample":True,}
    code_gens = generation(prompt, num_return_sequences=num_completions,
                           max_length=max_length, **gen_kwargs)
    code_strings = []
    for code_gen in code_gens:
        generated_code = first_block(code_gen['generated_text'][len(prompt):])
        code_strings.append(generated_code)
    print(('\n'+'='*80 + '\n').join(code_strings))
```

Let's start with a simple example and have the model write a function for us that calculates the area of a rectangle:

```
prompt = '''def area_of_rectangle(a: float, b: float):
    """Return the area of the rectangle."""'''
complete_code(generation, prompt)
```

```
    return math.sqrt(a * b)
================================================================================

    return a * b / 2.0
================================================================================

    return a * b
================================================================================

    return a * b / a
```

That looks pretty good! Although not all the generations are correct, the right solution is in there. Now, can the model also solve a more complex task of extracting URLs from an HTML string? Let's see:

```
prompt = '''def get_urls_from_html(html):
    """Get all embedded URLs in a HTML string."""'''
complete_code(generation, prompt)
```

```
    if not html:
        return []
    return [url for url in re.findall(r'<a href="(/[^/]+/[^"]+?)">', html)]
================================================================================

    return [url for url in re.findall(r'<a href="(.*?)"', html)
            if url]
================================================================================

    return [url for url in re.findall(r'<a href="(/.*)",', html)]
================================================================================

    return re.findall(r'<a href="(.*?)" class="url"[^>]*>', html)
```

Although it didn't quite get it right in the second attempt, the other three generations are correct. We can test the function on the Hugging Face home page:

```
import requests

def get_urls_from_html(html):
    return [url for url in re.findall(r'<a href="(.*?)"', html) if url]

print(" | ".join(get_urls_from_html(requests.get('https://hf.co/').text)))
```

```
https://github.com/huggingface/transformers | /allenai | /facebook |
/asteroid-team | /google | /amazon | /speechbrain | /microsoft | /grammarly |
/models | /inference-api | /distilbert-base-uncased |
/dbmdz/bert-large-cased-finetuned-conll03-english |
https://huggingface.co/transformers | https://arxiv.org/abs/1811.06031 |
https://arxiv.org/abs/1803.10631 | https://transformer.huggingface.co/ | /coref
| https://medium.com/huggingface/distilbert-8cf3380435b5
```

We can see that all the URLs starting with `https` are external pages, whereas the others are subpages of the main website. That's exactly what we wanted. Finally, let's load the large model and see if we can use it to translate a function from pure Python to NumPy:

```
model_ckpt = 'transformersbook/codeparrot'
generation = pipeline('text-generation', model=model_ckpt, device=0)

prompt = '''# a function in native python:
def mean(a):
    return sum(a)/len(a)

# the same function using numpy:
import numpy as np
def mean(a):'''
complete_code(generation, prompt, max_length=64)
```

```
Setting `pad_token_id` to `eos_token_id`:0 for open-end generation.


    return np.mean(a)
================================================================================

    return np.mean(a)
================================================================================

    return np.mean(a)
================================================================================

    return np.mean(a)
```

That worked! Let's see if we can also use the CodeParrot model to help us build a Scikit-learn model:

```
prompt = '''X = np.random.randn(100, 100)
y = np.random.randint(0, 1, 100)

# fit random forest classifier with 20 estimators'''
complete_code(generation, prompt, max_length=96)
```

```
Setting `pad_token_id` to `eos_token_id`:0 for open-end generation.


reg = DummyRegressor()

forest = RandomForestClassifier(n_estimators=20)

forest.fit(X, y)
================================================================================

clf = ExtraTreesClassifier(n_estimators=100, max_features='sqrt')
clf.fit(X, y)
================================================================================

clf = RandomForestClassifier(n_estimators=20, n_jobs=n_jobs, random_state=1)
clf.fit(X, y)
================================================================================

clf = RandomForestClassifier(n_estimators=20)
clf.fit(X, y)
```

Although in the second attempt it tried to train an extra-trees classifier, it generated what we asked in the other cases.

In Chapter 5 we explored a few metrics to measure the quality of generated text. Among these was the BLEU score, which is frequently used for that purpose. While this metric has limitations in general, it is particularly badly suited for our use case. The BLEU score measures the overlap of *n*-grams between the reference texts and the generated texts. When writing code we have a lot of freedom in terms of variables

and classes, and the success of a program does not depend on the naming scheme as long as it is consistent. However, the BLEU score would punish a generation that deviates from the reference naming, which might in fact be almost impossible to predict (even for a human coder).

In software development there are much better and more reliable ways to measure the quality of code, such as unit tests. This is how all the OpenAI Codex models were evaluated: by running several code generations for coding tasks through a set of unit tests and calculating the fraction of generations that pass the tests.[10] For a proper performance measure we should apply the same evaluation regimen to our models but this is beyond the scope of this chapter. You can find details on how CodeParrot performs on the HumanEval benchmark in the model's accompanying blog post.

Conclusion

Let's take a step back for a moment and contemplate what we have achieved in this chapter. We set out to create a code autocomplete function for Python. First we built a custom, large-scale dataset suitable for pretraining a large language model. Then we created a custom tokenizer that is able to efficiently encode Python code with that dataset. Finally, with the help of 🤗 Accelerate we put everything together and wrote a training script to train small and large versions of a GPT-2 model from scratch on a multi-GPU infrastructure, in under two hundred lines of code. Investigating the model outputs, we saw that it can generate reasonable code continuations, and we discussed how the model could be systematically evaluated.

You now not only know how to fine-tune any of the many pretrained models on the Hub, but also how to pretrain a custom model from scratch when you have enough data and compute resources available. You are now prepared to tackle almost any NLP use case with transformers. So the question is: where to next? In the next and last chapter, we'll have a look at where the field is currently moving and what new exciting applications and domains beyond NLP transformer models can tackle.

10 M. Chen et al., "Evaluating Large Language Models Trained on Code", (2021).