CHAPTER 9

Dealing with Few to No Labels

There is one question so deeply ingrained into every data scientist's mind that it's usually the first thing they ask at the start of a new project: is there any labeled data? More often than not, the answer is "no" or "a little bit," followed by an expectation from the client that your team's fancy machine learning models should still perform well. Since training models on very small datasets does not typically yield good results, one obvious solution is to annotate more data. However, this takes time and can be very expensive, especially if each annotation requires domain expertise to validate.

Fortunately, there are several methods that are well suited for dealing with few to no labels! You may already be familiar with some of them, such as *zero-shot* or *few-shot learning*, as witnessed by GPT-3's impressive ability to perform a diverse range of tasks with just a few dozen examples.

In general, the best-performing method will depend on the task, the amount of available data, and what fraction of that data is labeled. The decision tree shown in Figure 9-1 can help guide us through the process of picking the most appropriate method.

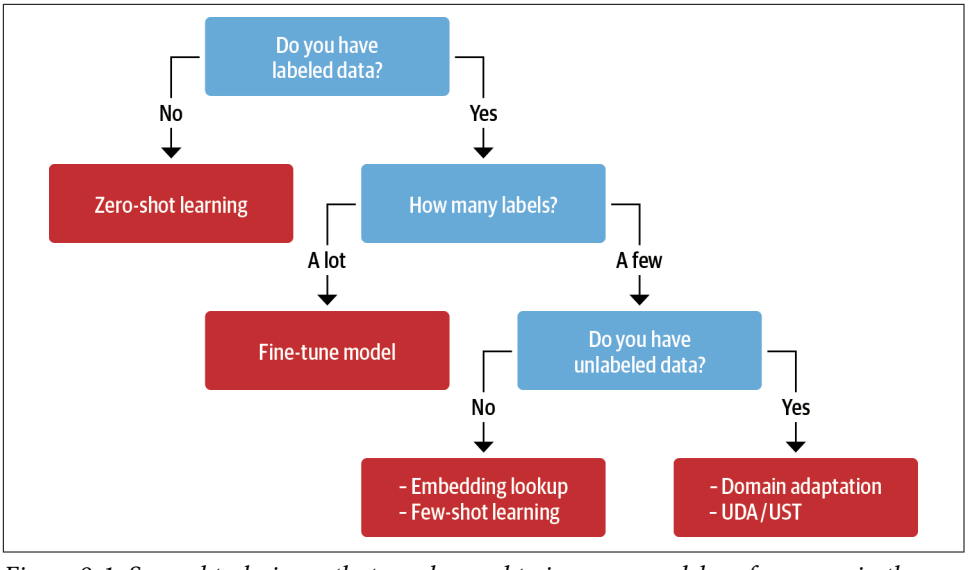


Figure 9-1. Several techniques that can be used to improve model performance in the absence of large amounts of labeled data

Let's walk through this decision tree step-by-step:

1. Do you have labeled data?
: Even a handful of labeled samples can make a difference with regard to which method works best. If you have no labeled data at all, you can start with the zero-shot learning approach, which often sets a strong baseline to work from.

2. How many labels?
: If labeled data is available, the deciding factor is how much. If you have a lot of training data available you can use the standard fine-tuning approach discussed in Chapter 2.

3. Do you have unlabeled data?
: If you only have a handful of labeled samples it can help immensely if you have access to large amounts of unlabeled data. If you have access to unlabeled data you can either use it to fine-tune the language model on the domain before training a classifier, or you can use more sophisticated methods such as unsupervised data augmentation (UDA) or uncertainty-aware self-training (UST).[1] If you don't have any unlabeled data available, you don't have the option of annotating more

1 Q. Xie et al., "Unsupervised Data Augmentation for Consistency Training", (2019); S. Mukherjee and A.H. Awadallah, "Uncertainty-Aware Self-Training for Few-Shot Text Classification", (2020).

data. In this case you can use few-shot learning or use the embeddings from a pretrained language model to perform lookups with a nearest neighbor search.

In this chapter we'll work our way through this decision tree by tackling a common problem facing many support teams that use issue trackers like Jira or GitHub to assist their users: tagging issues with metadata based on the issue's description. These tags might define the issue type, the product causing the problem, or which team is responsible for handling the reported issue. Automating this process can have a big impact on productivity and enables the support teams to focus on helping their users. As a running example, we'll use the GitHub issues associated with a popular open source project: 🤗 Transformers! Let's now take a look at what information is contained in these issues, how to frame the task, and how to get the data.

The methods presented in this chapter work well for text classification, but other techniques such as data augmentation may be necessary for tackling more complex tasks like named entity recognition, question answering, or summarization.

Building a GitHub Issues Tagger

If you navigate to the Issues tab of the 🤗 Transformers repository, you'll find issues like the one shown in Figure 9-2, which contains a title, a description, and a set of tags or labels that characterize the issue. This suggests a natural way to frame the supervised learning task: given a title and description of an issue, predict one or more labels. Since each issue can be assigned a variable number of labels, this means we are dealing with a *multilabel text classification* problem. This is usually more challenging than the multiclass problem that we encountered in Chapter 2, where each tweet was assigned to only one emotion.

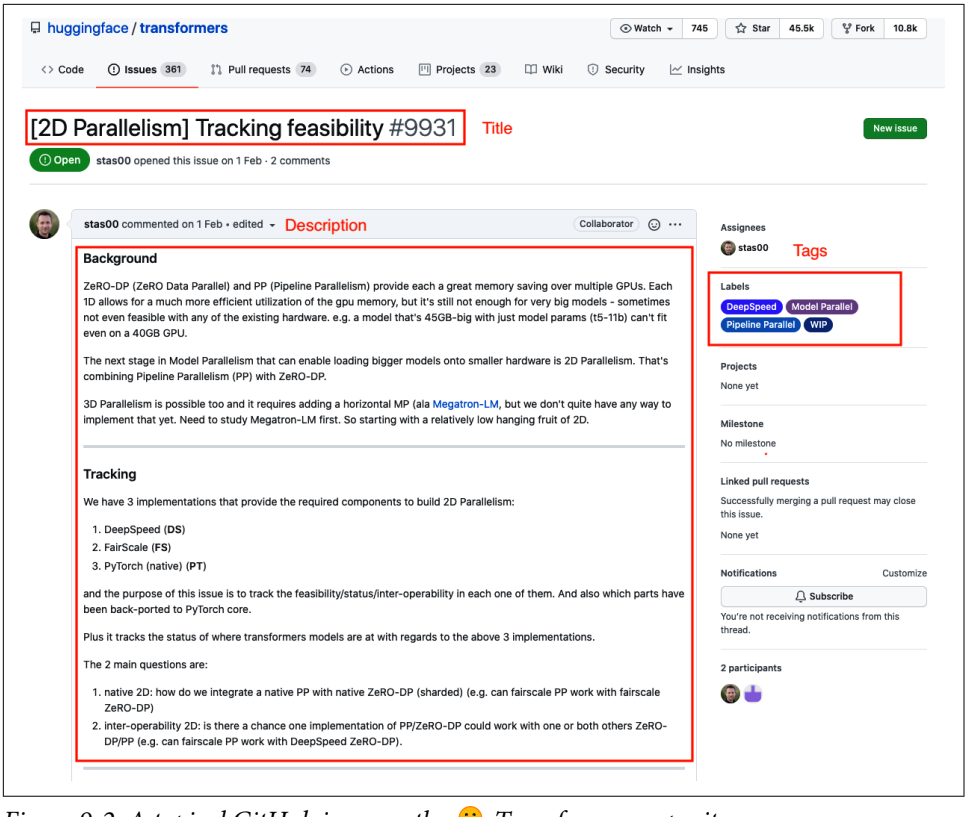


Figure 9-2. A typical GitHub issue on the 🤗 Transformers repository

Now that we've seen what the GitHub issues look like, let's see how we can download them to create our dataset.

Getting the Data

To grab all the repository's issues, we'll use the GitHub REST API to poll the `Issues` endpoint. This endpoint returns a list of JSON objects, with each containing a large number of fields about the issue at hand, including its state (open or closed), who opened the issue, as well as the title, body, and labels we saw in Figure 9-2.

Since it takes a while to fetch all the issues, we've included a *github-issues-transformers.jsonl* file in this book's GitHub repository, along with a `fetch_issues()` function that you can use to download them yourself.

The GitHub REST API treats pull requests as issues, so our dataset contains a mix of both. To keep things simple, we'll develop our classifier for both types of issue, although in practice you might consider building two separate classifiers to have more fine-grained control over the model's performance.

Now that we know how to grab the data, let's take a look at cleaning it up.

Preparing the Data

Once we've downloaded all the issues, we can load them using Pandas:

```
import pandas as pd

dataset_url = "https://git.io/nlp-with-transformers"
df_issues = pd.read_json(dataset_url, lines=True)
print(f"DataFrame shape: {df_issues.shape}")
```

```
DataFrame shape: (9930, 26)
```

There are almost 10,000 issues in our dataset, and by looking at a single row we can see that the information retrieved from the GitHub API contains many fields such as URLs, IDs, dates, users, title, body, as well as labels:

```
cols = ["url", "id", "title", "user", "labels", "state", "created_at", "body"]
df_issues.loc[2, cols].to_frame()
```

	2
url	https://api.github.com/repos/huggingface/trans...
id	849529761
title	[DeepSpeed] ZeRO stage 3 integration: getting ...
user	{'login': 'stas00', 'id': 10676103, 'node_id':...
labels	[{'id': 2659267025, 'node_id': 'MDU6TGFiZWwyNj...
state	open
created_at	2021-04-02 23:40:42
body	**[This is not yet alive, preparing for the re...

The `labels` column is the thing that we're interested in, and each row contains a list of JSON objects with metadata about each label:

```
[
   {
      "id":2659267025,
      "node_id":"MDU6TGFiZWwyNjU5MjY3MDI1",
      "url":"https://api.github.com/repos/huggingface...",
      "name":"DeepSpeed",
      "color":"4D34F7",
```

```
        "default":false,
        "description":""
    }
]
```

For our purposes, we're only interested in the `name` field of each label object, so let's overwrite the `labels` column with just the label names:

```
df_issues["labels"] = (df_issues["labels"]
                       .apply(lambda x: [meta["name"] for meta in x]))
df_issues[["labels"]].head()
```

	labels
0	[]
1	[]
2	[DeepSpeed]
3	[]
4	[]

Now each row in the `labels` column is a list of GitHub labels, so we can compute the length of each row to find the number of labels per issue:

```
df_issues["labels"].apply(lambda x : len(x)).value_counts().to_frame().T
```

	0	1	2	3	4	5
labels	6440	3057	305	100	25	3

This shows that the majority of issues have zero or one label, and much fewer have more than one. Next let's take a look at the top 10 most frequent labels in the dataset. In Pandas we can do this by "exploding" the `labels` column so that each label in the list becomes a row, and then simply counting the occurrences of each label:

```
df_counts = df_issues["labels"].explode().value_counts()
print(f"Number of labels: {len(df_counts)}")
# Display the top-8 label categories
df_counts.to_frame().head(8).T
```

```
Number of labels: 65
```

	wontfix	model card	Core: Tokenization	New model	Core: Modeling	Help wanted	Good First Issue	Usage
labels	2284	649	106	98	64	52	50	46

We can see that there are 65 unique labels in the dataset and that the classes are very imbalanced, with `wontfix` and `model card` being the most common labels. To make the classification problem more tractable, we'll focus on building a tagger for a subset

of the labels. For example, some labels, such as `Good First Issue` or `Help Wanted`, are potentially very difficult to predict from the issue's description, while others, such as `model card`, could be classified with a simple rule that detects when a model card is added on the Hugging Face Hub.

The following code filters the dataset for the subset of labels that we'll work with, along with a standardization of the names to make them easier to read:

```
label_map = {"Core: Tokenization": "tokenization",
             "New model": "new model",
             "Core: Modeling": "model training",
             "Usage": "usage",
             "Core: Pipeline": "pipeline",
             "TensorFlow": "tensorflow or tf",
             "PyTorch": "pytorch",
             "Examples": "examples",
             "Documentation": "documentation"}

def filter_labels(x):
    return [label_map[label] for label in x if label in label_map]

df_issues["labels"] = df_issues["labels"].apply(filter_labels)
all_labels = list(label_map.values())
```

Now let's look at the distribution of the new labels:

```
df_counts = df_issues["labels"].explode().value_counts()
df_counts.to_frame().T
```

	tokenization	new model	model training	usage	pipeline	tensorflow or tf	pytorch	documentation	examples
labels	106	98	64	46	42	41	37	28	24

Later in this chapter we'll find it useful to treat the unlabeled issues as a separate training split, so let's create a new column that indicates whether the issue is unlabeled or not:

```
df_issues["split"] = "unlabeled"
mask = df_issues["labels"].apply(lambda x: len(x)) > 0
df_issues.loc[mask, "split"] = "labeled"
df_issues["split"].value_counts().to_frame()
```

	split
unlabeled	9489
labeled	441

Let's now take a look at an example:

```
for column in ["title", "body", "labels"]:
    print(f"{column}: {df_issues[column].iloc[26][:500]}\n")
```

```
title: Add new CANINE model

body: # ⭐ New model addition

## Model description

Google recently proposed a new **C**haracter **A**rchitecture with **N**o
 tokenization **I**n **N**eural **E**ncoders architecture (CANINE). Not only
 the title is exciting:

Pipelined NLP systems have largely been superseded by end-to-end neural
 modeling, yet nearly all commonly-used models still require an explicit
 tokenization step. While recent tokenization approaches based on data-derived
 subword lexicons are less brittle than manually en

labels: ['new model']
```

In this example a new model architecture is proposed, so the `new model` tag makes sense. We can also see that the `title` contains information that will be useful for our classifier, so let's concatenate it with the issue's description in the `body` field:

```
df_issues["text"] = (df_issues
                     .apply(lambda x: x["title"] + "\n\n" + x["body"], axis=1))
```

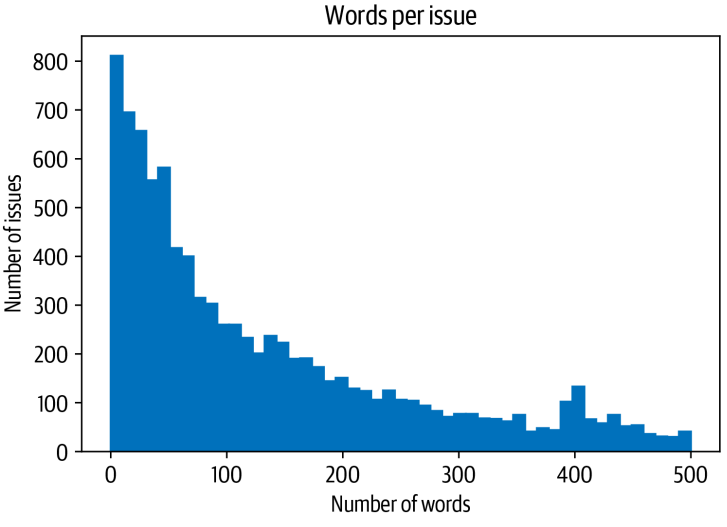
Before we look at the rest of the data, let's check for any duplicates in the data and drop them with the `drop_duplicates()` method:

```
len_before = len(df_issues)
df_issues = df_issues.drop_duplicates(subset="text")
print(f"Removed {(len_before-len(df_issues))/len_before:.2%} duplicates.")
```

```
Removed 1.88% duplicates.
```

We can see that there were a few duplicate issues in our dataset, but they only represented a small percentage. As we've done in other chapters, it's also a good idea to have a quick look at the number of words in our texts to see if we'll lose much information when we truncate to each model's context size:

```
import numpy as np
import matplotlib.pyplot as plt

(df_issues["text"].str.split().apply(len)
 .hist(bins=np.linspace(0, 500, 50), grid=False, edgecolor="C0"))
plt.title("Words per issue")
plt.xlabel("Number of words")
plt.ylabel("Number of issues")
plt.show()
```



The distribution has the long tail characteristic of many text datasets. Most of the texts are fairly short, but there are also issues with more than 500 words. It is common to have some very long issues, especially when error messages and code snippets are posted along with them. Given that most transformer models have a context size of 512 tokens or larger, truncating a handful of long issues is not likely to affect the overall performance. Now that we've explored and cleaned up our dataset, the final thing to do is define our training and validation sets to benchmark our classifiers. Let's take a look at how to do this.

Creating Training Sets

Creating training and validation sets is a bit trickier for multlilabel problems because there is no guaranteed balance for all labels. However, it can be approximated, and we can use the Scikit-multilearn library, which is specifically set up for this purpose. The first thing we need to do is transform our set of labels, like `pytorch` and `tokenization`, into a format that the model can process. Here we can use Scikit-learn's `MultiLabelBinarizer` class, which takes a list of label names and creates a vector with zeros for absent labels and ones for present labels. We can test this by fitting `MultiLabelBinarizer` on `all_labels` to learn the mapping from label name to ID as follows:

```
from sklearn.preprocessing import MultiLabelBinarizer

mlb = MultiLabelBinarizer()
mlb.fit([all_labels])
mlb.transform([["tokenization", "new model"], ["pytorch"]])
```

```
array([[0, 0, 0, 1, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0]])
```

In this simple example we can see the first row has two ones corresponding to the `tokenization` and `new model` labels, while the second row has just one hit with `pytorch`.

To create the splits we can use the `iterative_train_test_split()` function from Scikit-multilearn, which creates the train/test splits iteratively to achieve balanced labels. We wrap it in a function that we can apply to `DataFrames`. Since the function expects a two-dimensional feature matrix, we need to add a dimension to the possible indices before making the split:

```
from skmultilearn.model_selection import iterative_train_test_split

def balanced_split(df, test_size=0.5):
    ind = np.expand_dims(np.arange(len(df)), axis=1)
    labels = mlb.transform(df["labels"])
    ind_train, _, ind_test, _ = iterative_train_test_split(ind, labels,
                                                           test_size)
    return df.iloc[ind_train[:, 0]], df.iloc[ind_test[:,0]]
```

Armed with the `balanced_split()` function, we can split the data into supervised and unsupervised datasets, and then create balanced training, validation, and test sets for the supervised part:

```
from sklearn.model_selection import train_test_split

df_clean = df_issues[["text", "labels", "split"]].reset_index(drop=True).copy()
df_unsup = df_clean.loc[df_clean["split"] == "unlabeled", ["text", "labels"]]
df_sup = df_clean.loc[df_clean["split"] == "labeled", ["text", "labels"]]

np.random.seed(0)
df_train, df_tmp = balanced_split(df_sup, test_size=0.5)
df_valid, df_test = balanced_split(df_tmp, test_size=0.5)
```

Finally, let's create a `DatasetDict` with all the splits so that we can easily tokenize the dataset and integrate with the `Trainer`. Here we'll use the nifty `from_pandas()` method to load each split directly from the corresponding Pandas `DataFrame`:

```
from datasets import Dataset, DatasetDict

ds = DatasetDict({
    "train": Dataset.from_pandas(df_train.reset_index(drop=True)),
    "valid": Dataset.from_pandas(df_valid.reset_index(drop=True)),
    "test": Dataset.from_pandas(df_test.reset_index(drop=True)),
    "unsup": Dataset.from_pandas(df_unsup.reset_index(drop=True))})
```

This looks good, so the last thing to do is to create some training slices so that we can evaluate the performance of each classifier as a function of the training set size.

Creating Training Slices

The dataset has the two characteristics that we'd like to investigate in this chapter: sparse labeled data and multilabel classification. The training set consists of only 220 examples to train with, which is certainly a challenge even with transfer learning. To drill down into how each method in this chapter performs with little labeled data, we'll also create slices of the training data with even fewer samples. We can then plot the number of samples against the performance and investigate various regimes. We'll start with only eight samples per label and build up until the slice covers the full training set using the `iterative_train_test_split()` function:

```
np.random.seed(0)
all_indices = np.expand_dims(list(range(len(ds["train"]))), axis=1)
indices_pool = all_indices
labels = mlb.transform(ds["train"]["labels"])
train_samples = [8, 16, 32, 64, 128]
train_slices, last_k = [], 0

for i, k in enumerate(train_samples):
    # Split off samples necessary to fill the gap to the next split size
    indices_pool, labels, new_slice, _ = iterative_train_test_split(
        indices_pool, labels, (k-last_k)/len(labels))
    last_k = k
    if i==0: train_slices.append(new_slice)
    else: train_slices.append(np.concatenate((train_slices[-1], new_slice)))

# Add full dataset as last slice
train_slices.append(all_indices), train_samples.append(len(ds["train"]))
train_slices = [np.squeeze(train_slice) for train_slice in train_slices]
```

Note that this iterative approach only approximately splits the samples to the desired size, since it is not always possible to find a balanced split at a given split size:

```
print("Target split sizes:")
print(train_samples)
print("Actual split sizes:")
print([len(x) for x in train_slices])
```

```
Target split sizes:
[8, 16, 32, 64, 128, 223]
Actual split sizes:
[10, 19, 36, 68, 134, 223]
```

We'll use the specified split sizes as the labels for the following plots. Great, we've finally prepared our dataset into training splits—let's next take a look at training a strong baseline model!

Implementing a Naive Bayesline

Whenever you start a new NLP project, it's always a good idea to implement a set of strong baselines. There are two main reasons for this:

1. A baseline based on regular expressions, handcrafted rules, or a very simple model might already work really well to solve the problem. In these cases, there is no reason to bring out big guns like transformers, which are generally more complex to deploy and maintain in production environments.
2. The baselines provide quick checks as you explore more complex models. For example, suppose you train BERT-large and get an accuracy of 80% on your validation set. You might write it off as a hard dataset and call it a day. But what if you knew that a simple classifier like logistic regression gets 95% accuracy? That would raise your suspicions and prompt you to debug your model.

So let's start our analysis by training a baseline model. For text classification, a great baseline is a *Naive Bayes classifier* as it is very simple, quick to train, and fairly robust to perturbations in the inputs. The Scikit-learn implementation of Naive Bayes does not support multilabel classification out of the box, but fortunately we can again use the Scikit-multilearn library to cast the problem as a one-versus-rest classification task where we train L binary classifiers for L labels. First, let's use a multilabel binarizer to create a new `label_ids` column in our training sets. We can use the `map()` function to take care of all the processing in one go:

```
def prepare_labels(batch):
    batch["label_ids"] = mlb.transform(batch["labels"])
    return batch

ds = ds.map(prepare_labels, batched=True)
```

To measure the performance of our classifiers, we'll use the micro and macro F_1-scores, where the former tracks performance on the frequent labels and the latter on all labels disregarding the frequency. Since we'll be evaluating each model across different-sized training splits, let's create a `defaultdict` with a list to store the scores per split:

```
from collections import defaultdict

macro_scores, micro_scores = defaultdict(list), defaultdict(list)
```

Now we're finally ready to train our baseline! Here's the code to train the model and evaluate our classifier across increasing training set sizes:

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report
from skmultilearn.problem_transform import BinaryRelevance
from sklearn.feature_extraction.text import CountVectorizer
```

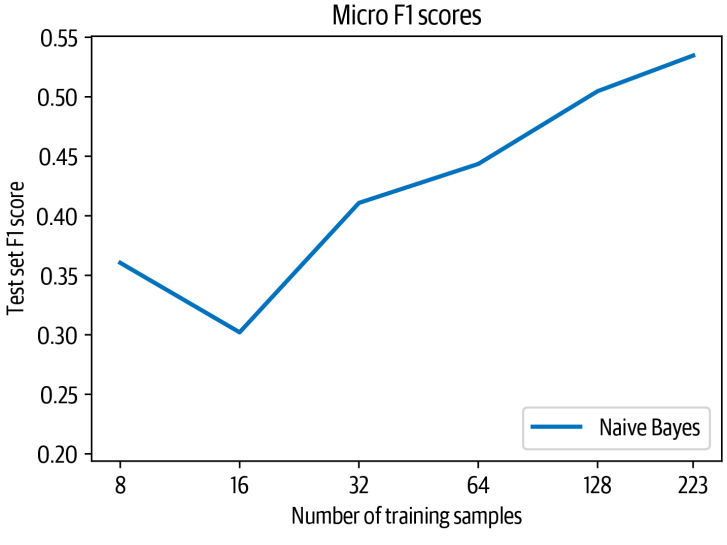
```
for train_slice in train_slices:
    # Get training slice and test data
    ds_train_sample = ds["train"].select(train_slice)
    y_train = np.array(ds_train_sample["label_ids"])
    y_test = np.array(ds["test"]["label_ids"])
    # Use a simple count vectorizer to encode our texts as token counts
    count_vect = CountVectorizer()
    X_train_counts = count_vect.fit_transform(ds_train_sample["text"])
    X_test_counts = count_vect.transform(ds["test"]["text"])
    # Create and train our model!
    classifier = BinaryRelevance(classifier=MultinomialNB())
    classifier.fit(X_train_counts, y_train)
    # Generate predictions and evaluate
    y_pred_test = classifier.predict(X_test_counts)
    clf_report = classification_report(
        y_test, y_pred_test, target_names=mlb.classes_, zero_division=0,
        output_dict=True)
    # Store metrics
    macro_scores["Naive Bayes"].append(clf_report["macro avg"]["f1-score"])
    micro_scores["Naive Bayes"].append(clf_report["micro avg"]["f1-score"])
```

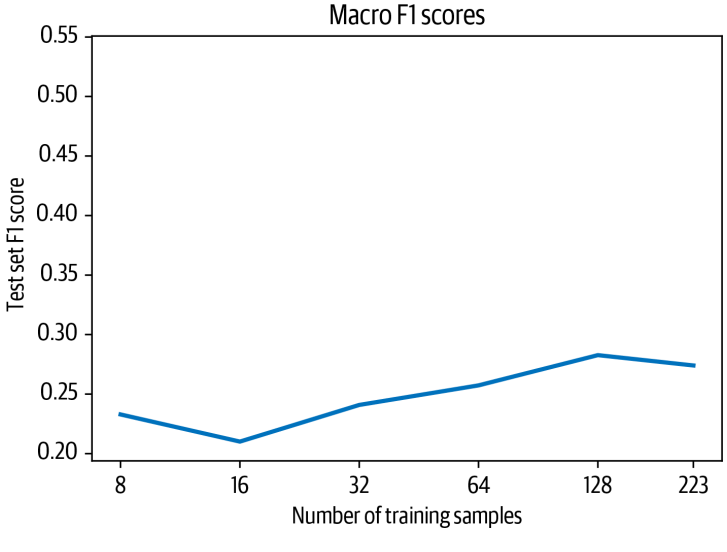
There's quite a lot going on in this block of code, so let's unpack it. First, we get the training slice and encode the labels. Then we use a count vectorizer to encode the texts by simply creating a vector of the size of the vocabulary where each entry corresponds to the frequency with which a token appeared in the text. This is called a *bag-of-words* approach, since all information on the order of the words is lost. Then we train the classifier and use the predictions on the test set to get the micro and macro F_1-scores via the classification report.

With the following helper function we can plot the results of this experiment:

```
import matplotlib.pyplot as plt

def plot_metrics(micro_scores, macro_scores, sample_sizes, current_model):
    fig, (ax0, ax1) = plt.subplots(1, 2, figsize=(10, 4), sharey=True)

    for run in micro_scores.keys():
        if run == current_model:
            ax0.plot(sample_sizes, micro_scores[run], label=run, linewidth=2)
            ax1.plot(sample_sizes, macro_scores[run], label=run, linewidth=2)
        else:
            ax0.plot(sample_sizes, micro_scores[run], label=run,
                     linestyle="dashed")
            ax1.plot(sample_sizes, macro_scores[run], label=run,
                     linestyle="dashed")

    ax0.set_title("Micro F1 scores")
    ax1.set_title("Macro F1 scores")
    ax0.set_ylabel("Test set F1 score")
    ax0.legend(loc="lower right")
    for ax in [ax0, ax1]:
```

```
            ax.set_xlabel("Number of training samples")
            ax.set_xscale("log")
            ax.set_xticks(sample_sizes)
            ax.set_xticklabels(sample_sizes)
            ax.minorticks_off()
        plt.tight_layout()
        plt.show()

plot_metrics(micro_scores, macro_scores, train_samples, "Naive Bayes")
```





Note that we plot the number of samples on a logarithmic scale. From the figure we can see that the micro and macro F_1-scores both improve as we increase the number of training samples. With so few samples to train on, the results are also slightly noisy since each slice can have a different class distribution. Nevertheless, what's important here is the trend, so let's now see how these results fare against transformer-based approaches!

Working with No Labeled Data

The first technique that we'll consider is *zero-shot classification*, which is suitable in settings where you have no labeled data at all. This is surprisingly common in industry, and might occur because there is no historic data with labels or because acquiring the labels for the data is difficult. We will cheat a bit in this section since we will still use the test data to measure the performance, but we will not use any data to train the model (otherwise the comparison to the following approaches would be difficult).

The goal of zero-shot classification is to make use of a pretrained model without any additional fine-tuning on your task-specific corpus. To get a better idea of how this could work, recall that language models like BERT are pretrained to predict masked tokens in text on thousands of books and large Wikipedia dumps. To successfully predict a missing token, the model needs to be aware of the topic in the context. We can try to trick the model into classifying a document for us by providing a sentence like:

> "This section was about the topic [MASK]."

The model should then give a reasonable suggestion for the document's topic, since this is a natural text to occur in the dataset.[2]

Let's illustrate this further with the following toy problem: suppose you have two children, and one of them likes movies with cars while the other enjoys movies with animals better. Unfortunately, they have already seen all the ones you know, so you want to build a function that tells you what topic a new movie is about. Naturally, you turn to transformers for this task. The first thing to try is to load BERT-base in the `fill-mask` pipeline, which uses the masked language model to predict the content of the masked tokens:

```
from transformers import pipeline

pipe = pipeline("fill-mask", model="bert-base-uncased")
```

2 We thank Joe Davison for suggesting this approach to us.

Next, let's construct a little movie description and add a prompt to it with a masked word. The goal of the prompt is to guide the model to help us make a classification. The `fill-mask` pipeline returns the most likely tokens to fill in the masked spot:

```
movie_desc = "The main characters of the movie madacascar \
are a lion, a zebra, a giraffe, and a hippo. "
prompt = "The movie is about [MASK]."

output = pipe(movie_desc + prompt)
for element in output:
    print(f"Token {element['token_str']}:\t{element['score']:.3f}%")
```

```
Token animals:  0.103%
Token lions:    0.066%
Token birds:    0.025%
Token love:     0.015%
Token hunting:  0.013%
```

Clearly, the model predicts only tokens that are related to animals. We can also turn this around, and instead of getting the most likely tokens we can query the pipeline for the probability of a few given tokens. For this task we might choose `cars` and `animals`, so we can pass them to the pipeline as targets:

```
output = pipe(movie_desc + prompt, targets=["animals", "cars"])
for element in output:
    print(f"Token {element['token_str']}:\t{element['score']:.3f}%")
```

```
Token animals:  0.103%
Token cars:     0.001%
```

Unsurprisingly, the predicted probability for the token `cars` is much smaller than for `animals`. Let's see if this also works for a description that is closer to cars:

```
movie_desc = "In the movie transformers aliens \
can morph into a wide range of vehicles."

output = pipe(movie_desc + prompt, targets=["animals", "cars"])
for element in output:
    print(f"Token {element['token_str']}:\t{element['score']:.3f}%")
```

```
Token cars:     0.139%
Token animals:  0.006%
```

It does! This is only a simple example, and if we want to make sure it works well we should test it thoroughly, but it illustrates the key idea of many approaches discussed in this chapter: find a way to adapt a pretrained model for another task without training it. In this case we set up a prompt with a mask in such a way that we can use a masked language model directly for classification. Let's see if we can do better by adapting a model that has been fine-tuned on a task that's closer to text classification: *natural language inference* (NLI).

Using the masked language model for classification is a nice trick, but we can do better still by using a model that has been trained on a task that is closer to classification. There is a neat proxy task called *text entailment* that fits the bill. In text entailment, the model needs to determine whether two text passages are likely to follow or contradict each other. Models are typically trained to detect entailments and contradictions with datasets such as Multi-Genre NLI Corpus (MNLI) or Cross-Lingual NLI Corpus (XNLI).[3]

Each sample in these datasets is composed of three parts: a premise, a hypothesis, and a label, which can be one of `entailment`, `neutral`, or `contradiction`. The `entailment` label is assigned when the hypothesis text is necessarily true under the premise. The `contradiction` label is used when the hypothesis is necessarily false or inappropriate under the premise. If neither of these cases applies, then the `neutral` label is assigned. See Table 9-1 for examples of each.

Table 9-1. The three classes in the MLNI dataset

Premise	Hypothesis	Label
His favourite color is blue.	He is into heavy metal music.	`neutral`
She finds the joke hilarious.	She thinks the joke is not funny at all.	`contradiction`
The house was recently built.	The house is new.	`entailment`

Now, it turns out that we can hijack a model trained on the MNLI dataset to build a classifier without needing any labels at all! The key idea is to treat the text we wish to classify as the premise, and then formulate the hypothesis as:

> "This example is about {label}."

where we insert the class name for the label. The entailment score then tells us how likely that premise is to be about that topic, and we can run this for any number of classes sequentially. The downside of this approach is that we need to execute a forward pass for each class, which makes it less efficient than a standard classifier. Another slightly tricky aspect is that the choice of label names can have a large impact on the accuracy, and choosing labels with semantic meaning is generally the best approach. For example, if the label is simply `Class 1`, the model has no hint what this might mean and whether this constitutes a contradiction or entailment.

🤗 Transformers has an MNLI model for zero-shot classification built in. We can initialize it via a pipeline as follows:

3 A. Williams, N. Nangia, and S.R. Bowman, "A Broad-Coverage Challenge Corpus for Sentence Understanding Through Inference", (2018); A. Conneau et al., "XNLI: Evaluating Cross-Lingual Sentence Representations", (2018).

```
from transformers import pipeline

pipe = pipeline("zero-shot-classification", device=0)
```

The setting `device=0` makes sure that the model runs on the GPU instead of the default CPU to speed up inference. To classify a text, we simply need to pass it to the pipeline along with the label names. In addition, we can set `multi_label=True` to ensure that all the scores are returned and not only the maximum for single-label classification:

```
sample = ds["train"][0]
print(f"Labels: {sample['labels']}")
output = pipe(sample["text"], all_labels, multi_label=True)
print(output["sequence"][:400])
print("\nPredictions:")

for label, score in zip(output["labels"], output["scores"]):
    print(f"{label}, {score:.2f}")
```

```
Labels: ['new model']
Add new CANINE model

# ⭐ New model addition

## Model description

Google recently proposed a new **C**haracter **A**rchitecture with **N**o
tokenization **I**n **N**eural **E**ncoders architecture (CANINE). Not only the
title is exciting:

> Pipelined NLP systems have largely been superseded by end-to-end neural
modeling, yet nearly all commonly-used models still require an explicit tokeni

Predictions:
new model, 0.98
tensorflow or tf, 0.37
examples, 0.34
usage, 0.30
pytorch, 0.25
documentation, 0.25
model training, 0.24
tokenization, 0.17
pipeline, 0.16
```

Since we are using a subword tokenizer, we can even pass code to the model! The tokenization might not be very efficient because only a small fraction of the pretraining dataset for the zero-shot pipeline consists of code snippets, but since code is also made up of a lot of natural words this is not a big issue. Also, the code block might contain important information, such as the framework (PyTorch or TensorFlow).

We can see that the model is very confident that this text is about a new model, but it also produces relatively high scores for the other labels. An important aspect for zero-shot classification is the domain we're operating in. The texts we are dealing with here are very technical and mostly about coding, which makes them quite different from the original text distribution in the MNLI dataset. Thus, it is not surprising that this is a challenging task for the model; it might work much better for some domains than others, depending on how close they are to the training data.

Let's write a function that feeds a single example through the zero-shot pipeline, and then scale it out to the whole validation set by running `map()`:

```
def zero_shot_pipeline(example):
    output = pipe(example["text"], all_labels, multi_label=True)
    example["predicted_labels"] = output["labels"]
    example["scores"] = output["scores"]
    return example

ds_zero_shot = ds["valid"].map(zero_shot_pipeline)
```

Now that we have our scores, the next step is to determine which set of labels should be assigned to each example. There are a few options we can experiment with:

- Define a threshold and select all labels above the threshold.
- Pick the top *k* labels with the *k* highest scores.

To help us determine which method is best, let's write a `get_preds()` function that applies one of the approaches to retrieve the predictions:

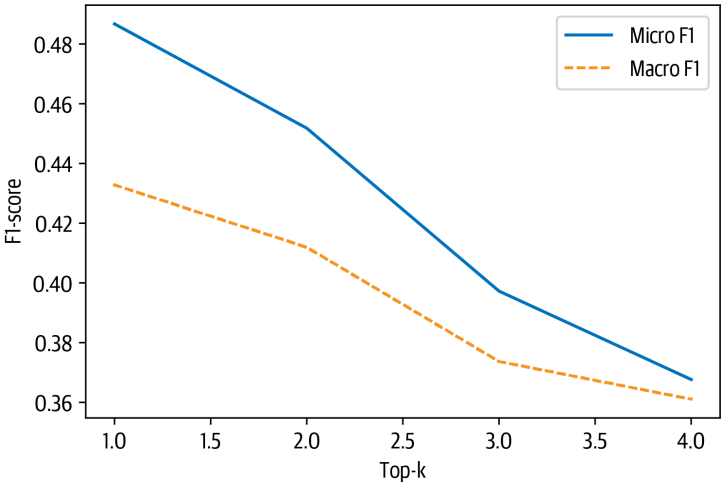
```
def get_preds(example, threshold=None, topk=None):
    preds = []
    if threshold:
        for label, score in zip(example["predicted_labels"], example["scores"]):
            if score >= threshold:
                preds.append(label)
    elif topk:
        for i in range(topk):
            preds.append(example["predicted_labels"][i])
    else:
        raise ValueError("Set either `threshold` or `topk`.")
    return {"pred_label_ids": list(np.squeeze(mlb.transform([preds])))}
```

Next, let's write a second function, `get_clf_report()`, that returns the Scikit-learn classification report from a dataset with the predicted labels:

```
def get_clf_report(ds):
    y_true = np.array(ds["label_ids"])
    y_pred = np.array(ds["pred_label_ids"])
    return classification_report(
        y_true, y_pred, target_names=mlb.classes_, zero_division=0,
        output_dict=True)
```

Armed with these two functions, let's start with the top-k method by increasing k for several values and then plotting the micro and macro F_1-scores across the validation set:

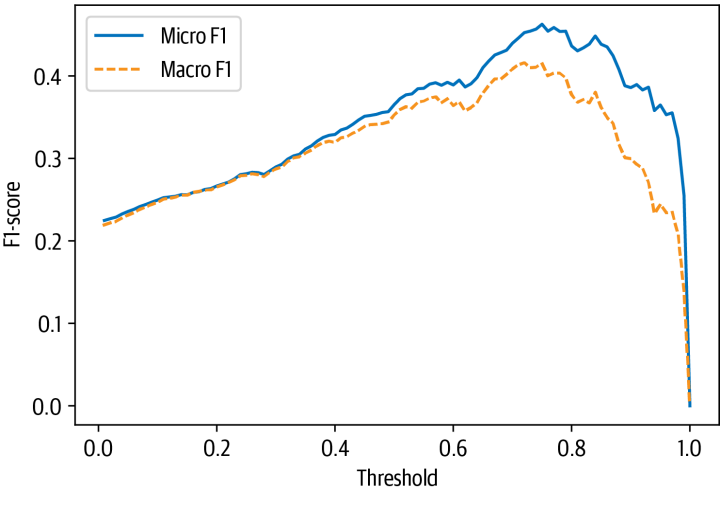
```
macros, micros = [], []
topks = [1, 2, 3, 4]
for topk in topks:
    ds_zero_shot = ds_zero_shot.map(get_preds, batched=False,
                                    fn_kwargs={'topk': topk})
    clf_report = get_clf_report(ds_zero_shot)
    micros.append(clf_report['micro avg']['f1-score'])
    macros.append(clf_report['macro avg']['f1-score'])

plt.plot(topks, micros, label='Micro F1')
plt.plot(topks, macros, label='Macro F1')
plt.xlabel("Top-k")
plt.ylabel("F1-score")
plt.legend(loc='best')
plt.show()
```



From the plot we can see that the best results are obtained by selecting the label with the highest score per example (top 1). This is perhaps not so surprising, given that most of the examples in our datasets have only one label. Let's now compare this against setting a threshold, so we can potentially predict more than one label per example:

```
macros, micros = [], []
thresholds = np.linspace(0.01, 1, 100)
for threshold in thresholds:
    ds_zero_shot = ds_zero_shot.map(get_preds,
                                    fn_kwargs={"threshold": threshold})
    clf_report = get_clf_report(ds_zero_shot)
```

```
        micros.append(clf_report["micro avg"]["f1-score"])
        macros.append(clf_report["macro avg"]["f1-score"])

plt.plot(thresholds, micros, label="Micro F1")
plt.plot(thresholds, macros, label="Macro F1")
plt.xlabel("Threshold")
plt.ylabel("F1-score")
plt.legend(loc="best")
plt.show()
```




```
best_t, best_micro = thresholds[np.argmax(micros)], np.max(micros)
print(f'Best threshold (micro): {best_t} with F1-score {best_micro:.2f}.')
best_t, best_macro = thresholds[np.argmax(macros)], np.max(macros)
print(f'Best threshold (micro): {best_t} with F1-score {best_macro:.2f}.')
```

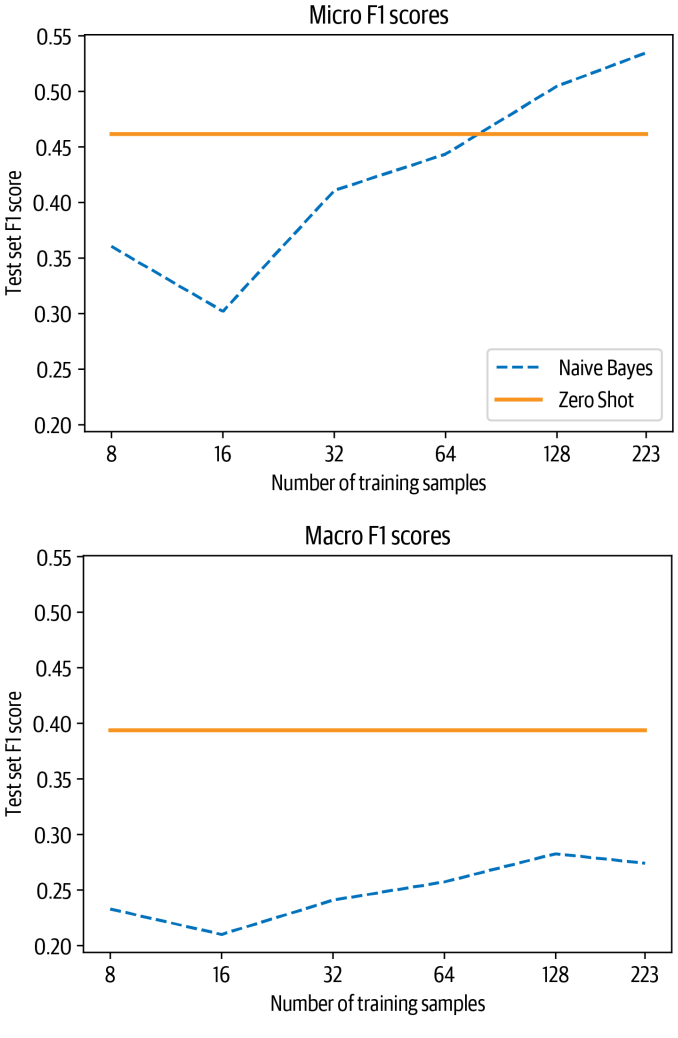
```
Best threshold (micro): 0.75 with F1-score 0.46.
Best threshold (micro): 0.72 with F1-score 0.42.
```

This approach fares somewhat worse than the top-1 results, but we can see the precision/recall trade-off clearly in this graph. If we set the threshold too low, then there are too many predictions, which leads to a low precision. If we set the threshold too high, then we will make hardly any predictions, which produces a low recall. From the plot we can see that a threshold value of around 0.8 is the sweet spot between the two.

Since the top-1 method performs best, let's use this to compare zero-shot classification against Naive Bayes on the test set:

```
ds_zero_shot = ds['test'].map(zero_shot_pipeline)
ds_zero_shot = ds_zero_shot.map(get_preds, fn_kwargs={'topk': 1})
clf_report = get_clf_report(ds_zero_shot)
for train_slice in train_slices:
```

```
        macro_scores['Zero Shot'].append(clf_report['macro avg']['f1-score'])
        micro_scores['Zero Shot'].append(clf_report['micro avg']['f1-score'])

plot_metrics(micro_scores, macro_scores, train_samples, "Zero Shot")
```



Comparing the zero-shot pipeline to the baseline, we observe two things:

1. If we have less than 50 labeled samples, the zero-shot pipeline handily outperforms the baseline.
2. Even above 50 samples, the performance of the zero-shot pipeline is superior when considering both the micro and macro F_1-scores. The results for the micro F_1-score tell us that the baseline performs well on the frequent classes, while the

zero-shot pipeline excels at those since it does not require any examples to learn from.

You might notice a slight paradox in this section: although we talk about dealing with no labels, we still use the validation and test sets. We use them to showcase different techniques and to make the results comparable between them. Even in a real use case, it makes sense to gather a handful of labeled examples to run some quick evaluations. The important point is that we did not adapt the parameters of the model with the data; instead, we just adapted some hyperparameters.

If you find it difficult to get good results on your own dataset, here are a few things you can do to improve the zero-shot pipeline:

- The way the pipeline works makes it very sensitive to the names of the labels. If the names don't make much sense or are not easily connected to the texts, the pipeline will likely perform poorly. Either try using different names or use several names in parallel and aggregate them in an extra step.
- Another thing you can improve is the form of the hypothesis. By default it is `hypothesis="This is example is about {}"`, but you can pass any other text to the pipeline. Depending on the use case, this might improve the performance.

Let's now turn to the regime where we have a few labeled examples we can use to train a model.

Working with a Few Labels

In most NLP projects, you'll have access to at least a few labeled examples. The labels might come directly from a client or cross-company team, or you might decide to just sit down and annotate a few examples yourself. Even for the previous approach, we needed a few labeled examples to evaluate how well the zero-shot approach worked. In this section, we'll have a look at how we can best leverage the few, precious labeled examples that we have. Let's start by looking at a technique known as data augmentation that can help us multiply the little labeled data that we have.

Data Augmentation

One simple but effective way to boost the performance of text classifiers on small datasets is to apply *data augmentation* techniques to generate new training examples from the existing ones. This is a common strategy in computer vision, where images are randomly perturbed without changing the meaning of the data (e.g., a slightly

rotated cat is still a cat). For text, data augmentation is somewhat trickier because perturbing the words or characters can completely change the meaning. For example, the two questions "Are elephants heavier than mice?" and "Are mice heavier than elephants?" differ by a single word swap, but have opposite answers. However, if the text consists of more than a few sentences (like our GitHub issues do), then the noise introduced by these types of transformations will generally not affect the label. In practice, there are two types of data augmentation techniques that are commonly used:

Back translation
: Take a text in the source language, translate it into one or more target languages using machine translation, and then translate it back to the source language. Back translation tends to works best for high-resource languages or corpora that don't contain too many domain-specific words.

Token perturbations
: Given a text from the training set, randomly choose and perform simple transformations like random synonym replacement, word insertion, swap, or deletion.[4]

Examples of these transformations are shown in Table 9-2. For a detailed list of other data augmentation techniques for NLP, we recommend reading Amit Chaudhary's blog post "A Visual Survey of Data Augmentation in NLP".

Table 9-2. Different types of data augmentation techniques for text

Augmentation	Sentence
None	Even if you defeat me Megatron, others will rise to defeat your tyranny
Synonym replace	Even if you kill me Megatron, others will prove to defeat your tyranny
Random insert	Even if you defeat me Megatron, others humanity will rise to defeat your tyranny
Random swap	You even if defeat me Megatron, others will rise defeat to tyranny your
Random delete	Even if you me Megatron, others to defeat tyranny
Back translate (German)	Even if you defeat me, others will rise up to defeat your tyranny

You can implement back translation using machine translation models like M2M100, while libraries like *NlpAug* and *TextAttack* provide various recipes for token perturbations. In this section, we'll focus on using synonym replacement as it's simple to implement and gets across the main idea behind data augmentation.

4 J. Wei and K. Zou, "EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks", (2019).

We'll use the `ContextualWordEmbsAug` augmenter from NlpAug to leverage the contextual word embeddings of DistilBERT for our synonym replacements. Let's start with a simple example:

```
from transformers import set_seed
import nlpaug.augmenter.word as naw

set_seed(3)
aug = naw.ContextualWordEmbsAug(model_path="distilbert-base-uncased",
                                device="cpu", action="substitute")

text = "Transformers are the most popular toys"
print(f"Original text: {text}")
print(f"Augmented text: {aug.augment(text)}")
```

```
Original text: Transformers are the most popular toys
Augmented text: transformers'the most popular toys
```

Here we can see how the word "are" has been replaced with an apostrophe to generate a new synthetic training example. We can wrap this augmentation in a simple function as follows:

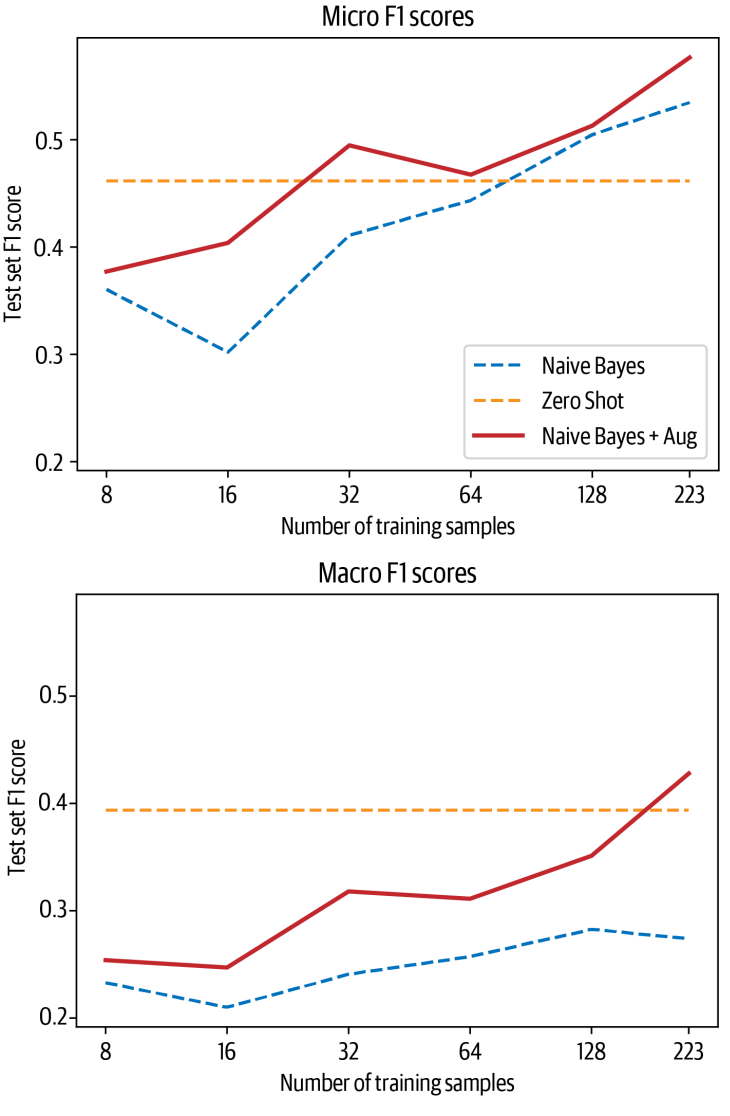
```
def augment_text(batch, transformations_per_example=1):
    text_aug, label_ids = [], []
    for text, labels in zip(batch["text"], batch["label_ids"]):
        text_aug += [text]
        label_ids += [labels]
        for _ in range(transformations_per_example):
            text_aug += [aug.augment(text)]
            label_ids += [labels]
    return {"text": text_aug, "label_ids": label_ids}
```

Now when we pass this function to the `map()` method, we can generate any number of new examples with the `transformations_per_example` argument. We can use this function in our code to train the Naive Bayes classifier by simply adding one line after we select the slice:

```
ds_train_sample = ds_train_sample.map(augment_text, batched=True,
    remove_columns=ds_train_sample.column_names).shuffle(seed=42)
```

Including this and rerunning the analysis produces the plot shown here:

```
plot_metrics(micro_scores, macro_scores, train_samples, "Naive Bayes + Aug")
```



From the figure, we can see that a small amount of data augmentation improves the F_1-score of the Naive Bayes classifier by around 5 points, and it overtakes the zero-shot pipeline for the macro scores once we have around 170 training samples. Let's now take a look at a method based on using the embeddings of large language models.

Using Embeddings as a Lookup Table

Large language models such as GPT-3 have been shown to be excellent at solving tasks with limited data. The reason is that these models learn useful representations of text that encode information across many dimensions, such as sentiment, topic, text structure, and more. For this reason, the embeddings of large language models can be used to develop a semantic search engine, find similar documents or comments, or even classify text.

In this section we'll create a text classifier that's modeled after the OpenAI API classification endpoint. The idea follows a three-step process:

1. Use the language model to embed all labeled texts.
2. Perform a nearest neighbor search over the stored embeddings.
3. Aggregate the labels of the nearest neighbors to get a prediction.

The process is illustrated in Figure 9-3, which shows how labeled data is embedded with a model and stored with the labels. When a new text needs to be classified it is embedded as well, and the label is given based on the labels of the nearest neighbors. It is important to calibrate the number of neighbors to be searched for, as too few might be noisy and too many might mix in neighboring groups.

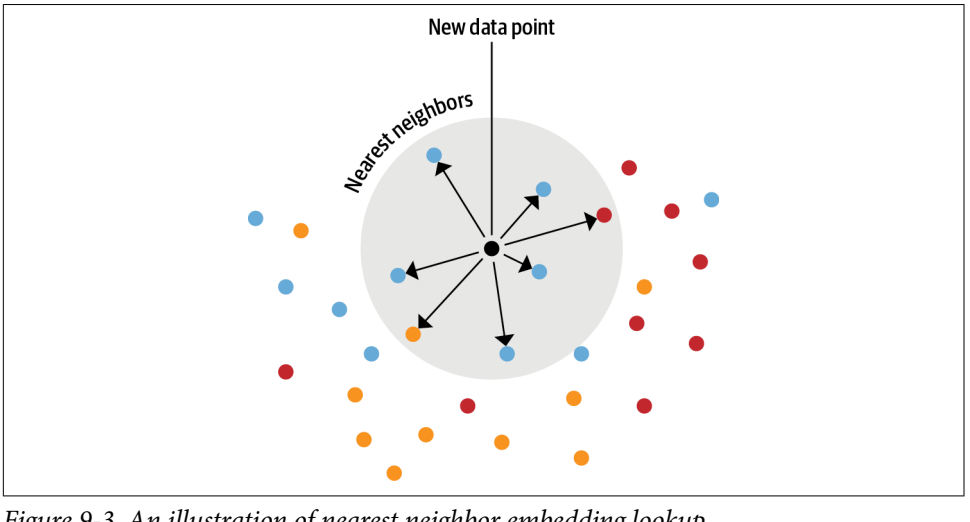


Figure 9-3. An illustration of nearest neighbor embedding lookup

The beauty of this approach is that no model fine-tuning is necessary to leverage the few available labeled data points. Instead, the main decision to make this approach work is to select an appropriate model that is ideally pretrained on a similar domain to your dataset.

Since GPT-3 is only available through the OpenAI API, we'll use GPT-2 to test the technique. Specifically, we'll use a variant of GPT-2 that was trained on Python code, which will hopefully capture some of the context contained in our GitHub issues.

Let's write a helper function that takes a list of texts and uses the model to create a single-vector representation for each text. One problem we have to deal with is that transformer models like GPT-2 will actually return one embedding vector per token. For example, given the sentence "I took my dog for a walk", we can expect several embedding vectors, one for each token. But what we really want is a single embedding vector for the whole sentence (or GitHub issue in our application). To deal with this, we can use a technique called *pooling*. One of the simplest pooling methods is to average the token embeddings, which is called *mean pooling*. With mean pooling, the only thing we need to watch out for is that we don't include padding tokens in the average, so we can use the attention mask to handle that.

To see how this works, let's load a GPT-2 tokenizer and model, define the mean pooling operation, and wrap the whole process in a simple `embed_text()` function:

```
import torch
from transformers import AutoTokenizer, AutoModel

model_ckpt = "miguelvictor/python-gpt2-large"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
model = AutoModel.from_pretrained(model_ckpt)

def mean_pooling(model_output, attention_mask):
    # Extract the token embeddings
    token_embeddings = model_output[0]
    # Compute the attention mask
    input_mask_expanded = (attention_mask
                           .unsqueeze(-1)
                           .expand(token_embeddings.size())
                           .float())
    # Sum the embeddings, but ignore masked tokens
    sum_embeddings = torch.sum(token_embeddings * input_mask_expanded, 1)
    sum_mask = torch.clamp(input_mask_expanded.sum(1), min=1e-9)
    # Return the average as a single vector
    return sum_embeddings / sum_mask

def embed_text(examples):
    inputs = tokenizer(examples["text"], padding=True, truncation=True,
                       max_length=128, return_tensors="pt")
    with torch.no_grad():
        model_output = model(**inputs)
    pooled_embeds = mean_pooling(model_output, inputs["attention_mask"])
    return {"embedding": pooled_embeds.cpu().numpy()}
```

Now we can get the embeddings for each split. Note that GPT-style models don't have a padding token, and therefore we need to add one before we can get the embeddings

in a batched fashion as implemented in the preceding code. We'll just recycle the end-of-string token for this purpose:

```
tokenizer.pad_token = tokenizer.eos_token
embs_train = ds["train"].map(embed_text, batched=True, batch_size=16)
embs_valid = ds["valid"].map(embed_text, batched=True, batch_size=16)
embs_test = ds["test"].map(embed_text, batched=True, batch_size=16)
```

Now that we have all the embeddings, we need to set up a system to search them. We could write a function that calculates, say, the cosine similarity between a new text embedding that we'll query and the existing embeddings in the training set. Alternatively, we can use a built-in structure of 🤗 Datasets called a *FAISS index*.[5] We already encountered FAISS in Chapter 7. You can think of this as a search engine for embeddings, and we'll have a closer look at how it works in a minute. We can use an existing field of the dataset to create a FAISS index with `add_faiss_index()`, or we can load new embeddings into the dataset with `add_faiss_index_from_external_arrays()`. Let's use the former function to add our training embeddings to the dataset as follows:

```
embs_train.add_faiss_index("embedding")
```

This created a new FAISS index called `embedding`. We can now perform a nearest neighbor lookup by calling the function `get_nearest_examples()`. It returns the closest neighbors as well as the matching score for each neighbor. We need to specify the query embedding as well as the number of nearest neighbors to retrieve. Let's give it a spin and have a look at the documents that are closest to an example:

```
i, k = 0, 3 # Select the first query and 3 nearest neighbors
rn, nl = "\r\n\r\n", "\n" # Used to remove newlines in text for compact display

query =  np.array(embs_valid[i]["embedding"], dtype=np.float32)
scores, samples = embs_train.get_nearest_examples("embedding", query, k=k)

print(f"QUERY LABELS: {embs_valid[i]['labels']}")
print(f"QUERY TEXT:\n{embs_valid[i]['text'][:200].replace(rn, nl)} [...]\n")
print("="*50)
print(f"Retrieved documents:")
for score, label, text in zip(scores, samples["labels"], samples["text"]):
    print("="*50)
    print(f"TEXT:\n{text[:200].replace(rn, nl)} [...]")
    print(f"SCORE: {score:.2f}")
    print(f"LABELS: {label}")
```

```
QUERY LABELS: ['new model']
QUERY TEXT:
Implementing efficient self attention in T5
```

5 J. Johnson, M. Douze, and H. Jégou, "Billion-Scale Similarity Search with GPUs", (2017).

```
# ⭐ New model addition
My teammates and I (including @ice-americano) would like to use efficient self
attention methods such as Linformer, Performer and [...]

==================================================
Retrieved documents:
==================================================
TEXT:
Add Linformer model

# ⭐ New model addition
## Model description
### Linformer: Self-Attention with Linear Complexity
Paper published June 9th on ArXiv: https://arxiv.org/abs/2006.04768
La [...]
SCORE: 54.92
LABELS: ['new model']
==================================================
TEXT:
Add FAVOR+ / Performer attention

# ⭐ FAVOR+ / Performer attention addition
Are there any plans to add this new attention approximation block to
Transformers library?
## Model description
The n [...]
SCORE: 57.90
LABELS: ['new model']
==================================================
TEXT:
Implement DeLighT: Very Deep and Light-weight Transformers

# ⭐ New model addition
## Model description
DeLight, that delivers similar or better performance than transformer-based
models with sign [...]
SCORE: 60.12
LABELS: ['new model']
```

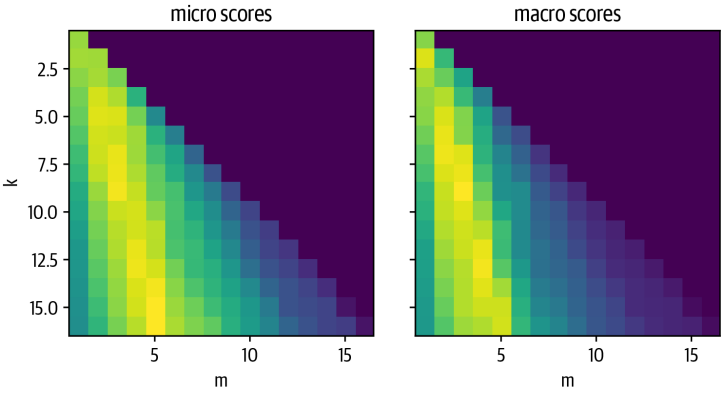
Nice! This is exactly what we hoped for: the three retrieved documents that we got via embedding lookup all have the same labels and we can already see from the titles that they are all very similar. The query as well as the retrieved documents revolve around adding new and efficient transformer models. The question remains, however, what is the best value for k? Similarly, how we should then aggregate the labels of the retrieved documents? Should we, for example, retrieve three documents and assign all labels that occurred at least twice? Or should we go for 20 and use all labels that appeared at least 5 times? Let's investigate this systematically: we'll try several values for k and then vary the threshold $m < k$ for label assignment with a helper function. We'll record the macro and micro performance for each setting so we can decide later which run performed best. Instead of looping over each sample in the validation set

we can make use of the function `get_nearest_examples_batch()`, which accepts a batch of queries:

```
def get_sample_preds(sample, m):
    return (np.sum(sample["label_ids"], axis=0) >= m).astype(int)

def find_best_k_m(ds_train, valid_queries, valid_labels, max_k=17):
    max_k = min(len(ds_train), max_k)
    perf_micro = np.zeros((max_k, max_k))
    perf_macro = np.zeros((max_k, max_k))
    for k in range(1, max_k):
        for m in range(1, k + 1):
            _, samples = ds_train.get_nearest_examples_batch("embedding",
                                                             valid_queries, k=k)
            y_pred = np.array([get_sample_preds(s, m) for s in samples])
            clf_report = classification_report(valid_labels, y_pred,
                target_names=mlb.classes_, zero_division=0, output_dict=True)
            perf_micro[k, m] = clf_report["micro avg"]["f1-score"]
            perf_macro[k, m] = clf_report["macro avg"]["f1-score"]
    return perf_micro, perf_macro
```

Let's check what the best values would be with all the training samples and visualize the scores for all *k* and *m* configurations:

```
valid_labels = np.array(embs_valid["label_ids"])
valid_queries = np.array(embs_valid["embedding"], dtype=np.float32)
perf_micro, perf_macro = find_best_k_m(embs_train, valid_queries, valid_labels)

fig, (ax0, ax1) = plt.subplots(1, 2, figsize=(10, 3.5), sharey=True)
ax0.imshow(perf_micro)
ax1.imshow(perf_macro)

ax0.set_title("micro scores")
ax0.set_ylabel("k")
ax1.set_title("macro scores")
for ax in [ax0, ax1]:
    ax.set_xlim([0.5, 17 - 0.5])
    ax.set_ylim([17 - 0.5, 0.5])
    ax.set_xlabel("m")
plt.show()
```



From the plots we can see that there is a pattern: choosing m too large or small for a given k yields suboptimal results. The best performance is achieved when choosing a ratio of approximately $m/k = 1/3$. Let's see which k and m give the best result overall:

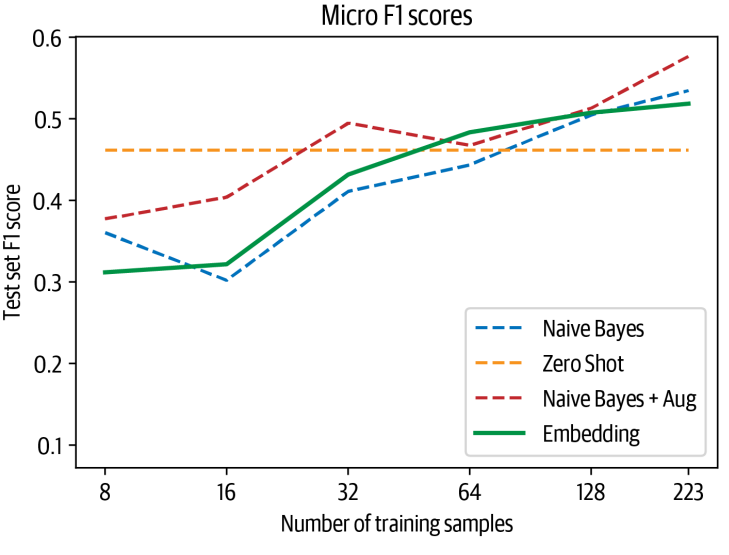
```
k, m = np.unravel_index(perf_micro.argmax(), perf_micro.shape)
print(f"Best k: {k}, best m: {m}")
```

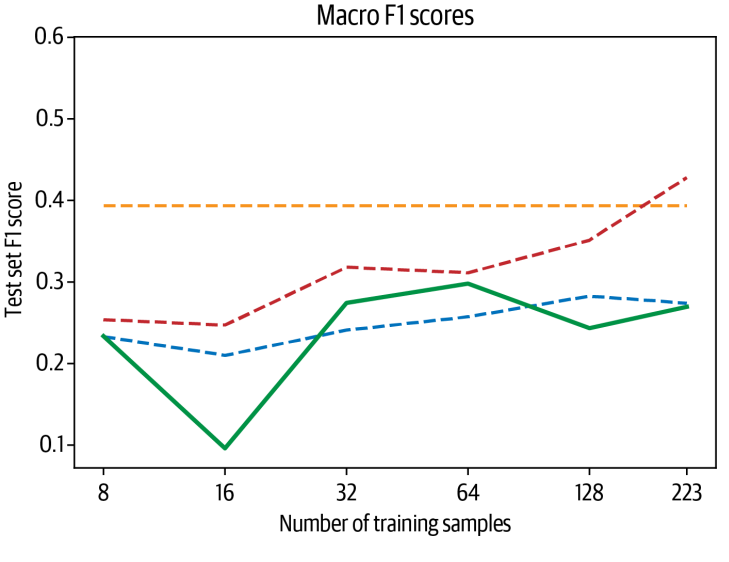
```
Best k: 15, best m: 5
```

The perfomance is best when we choose $k = 15$ and $m = 5$, or in other words when we retrieve the 15 nearest neighbors and then assign the labels that occurred at least 5 times. Now that we have a good method for finding the best values for the embedding lookup, we can play the same game as with the Naive Bayes classifier where we go through the slices of the training set and evaluate the performance. Before we can slice the dataset, we need to remove the index since we cannot slice a FAISS index like the dataset. The rest of the loops stay exactly the same, with the addition of using the validation set to get the best k and m values:

```
embs_train.drop_index("embedding")
test_labels = np.array(embs_test["label_ids"])
test_queries = np.array(embs_test["embedding"], dtype=np.float32)

for train_slice in train_slices:
    # Create a Faiss index from training slice
    embs_train_tmp = embs_train.select(train_slice)
    embs_train_tmp.add_faiss_index("embedding")
    # Get best k, m values with validation set
    perf_micro, _ = find_best_k_m(embs_train_tmp, valid_queries, valid_labels)
    k, m = np.unravel_index(perf_micro.argmax(), perf_micro.shape)
    # Get predictions on test set
    _, samples = embs_train_tmp.get_nearest_examples_batch("embedding",
                                                           test_queries,
                                                           k=int(k))
    y_pred = np.array([get_sample_preds(s, m) for s in samples])
    # Evaluate predictions
```

```
    clf_report = classification_report(test_labels, y_pred,
        target_names=mlb.classes_, zero_division=0, output_dict=True,)
    macro_scores["Embedding"].append(clf_report["macro avg"]["f1-score"])
    micro_scores["Embedding"].append(clf_report["micro avg"]["f1-score"])

plot_metrics(micro_scores, macro_scores, train_samples, "Embedding")
```





The embedding lookup is competitive on the micro scores with the previous approaches while just having two "learnable" parameters, k and m, but performs slightly worse on the macro scores.

Take these results with a grain of salt; which method works best strongly depends on the domain. The zero-shot pipeline's training data is quite different from the GitHub issues dataset we're using it on, which contains a lot of code that the model likely has not encountered much before. For a more common task such as sentiment analysis of reviews, the pipeline might work much better. Similarly, the embeddings' quality depends on the model and the data it was trained on. We tried half a dozen models, such as `sentence-transformers/stsb-roberta-large`, which was trained to give high-quality embeddings of sentences, and `microsoft/codebert-base` and `dbernsohn/roberta-python`, which were trained on code and documentation. For this specific use case, GPT-2 trained on Python code worked best.

Since you don't actually need to change anything in your code besides replacing the model checkpoint name to test another model, you can quickly try out a few models once you have the evaluation pipeline set up.

Let's now compare this simple embedding trick against simply fine-tuning a transformer on the limited data we have.

Efficient Similarity Search with FAISS

We first encountered FAISS in Chapter 7, where we used it to retrieve documents via the DPR embeddings. Here we'll explain briefly how the FAISS library works and why it is a powerful tool in the ML toolbox.

We are used to performing fast text queries on huge datasets such as Wikipedia or the web with search engines such as Google. When we move from text to embeddings, we would like to maintain that performance; however, the methods used to speed up text queries don't apply to embeddings.

To speed up text search we usually create an inverted index that maps terms to documents. An inverted index works like an index at the end of a book: each word is mapped to the pages (or in our case, document) it occurs in. When we later run a query we can quickly look up in which documents the search terms appear. This works well with discrete objects such as words, but does not work with continuous objects such as vectors. Each document likely has a unique vector, and therefore the index will never match with a new vector. Instead of looking for exact matches, we need to look for close or similar matches.

When we want to find the most similar vectors in a database to a query vector, in theory we need to compare the query vector to each of the n vectors in the database. For a small database such as we have in this chapter this is no problem, but if we

scaled this up to thousands or even million of entries we would need to wait a while for each query to be processed.

FAISS addresses this issue with several tricks. The main idea is to partition the dataset. If we only need to compare the query vector to a subset of the database, we can speed up the process significantly. But if we just randomly partition the dataset, how can we decide which partition to search, and what guarantees do we get for finding the most similar entries? Evidently, there must be a better solution: apply k-means clustering to the dataset! This clusters the embeddings into groups by similarity. Furthermore, for each group we get a centroid vector, which is the average of all members of the group (Figure 9-4).

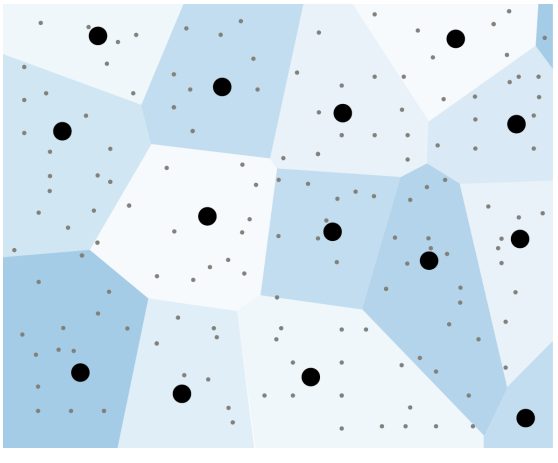
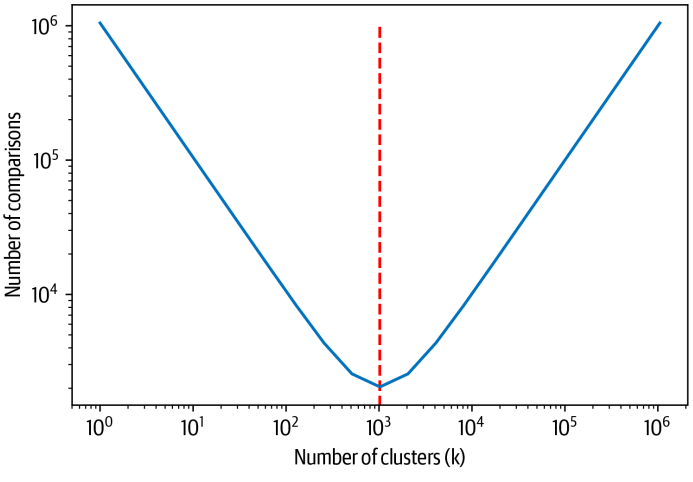


Figure 9-4. The structure of a FAISS index: the gray points represent data points added to the index, the bold black points are the cluster centers found via k-means clustering, and the colored areas represent the regions belonging to a cluster center

Given such a grouping, searching among n vectors is much easier: we first search across the k centroids for the one that is most similar to our query (k comparisons), and then we search within the group ($\frac{k}{n}$ elements to compare). This reduces the number of comparisons from n to $k + \frac{n}{k}$. So the question is, what is the best option for k? If it is too small, each group still contains many samples we need to compare against in the second step, and if k is too large there are many centroids we need to search through. Looking for the minimum of the function $f(k) = k + \frac{n}{k}$ with respect to k, we find $k = \sqrt{n}$. In fact, we can visualize this with the following graphic with $n = 2^{20}$.



In the plot you can see the number of comparisons as a function of the number of clusters. We are looking for the minimum of this function, where we need to do the least comparisons. We can see that the minimum is exactly where we expected to see it, at $\sqrt{2^{20}} = 2^{10} = 1,024$.

In addition to speeding up queries with partitioning, FAISS also allows you to utilize GPUs for a further speedup. If memory becomes a concern there are also several options to compress the vectors with advanced quantization schemes. If you want to use FAISS for your project, the repository has a simple guide for you to choose the right methods for your use case.

One of the largest projects to use FAISS was the creation of the CCMatrix corpus by Facebook. The authors used multilingual embeddings to find parallel sentences in different languages. This enormous corpus was subsequently used to train M2M100, a large machine translation model that is able to directly translate between any of 100 languages.

Fine-Tuning a Vanilla Transformer

If we have access to labeled data, we can also try to do the obvious thing: simply fine-tune a pretrained transformer model. In this section, we'll use the standard BERT checkpoint as a starting point. Later, we'll see the effect that fine-tuning the language model has on performance.

For many applications, starting with a pretrained BERT-like model is a good idea. However, if the domain of your corpus differs significantly from the pretraining corpus (which is usually Wikipedia), you should explore the many models that are available on the Hugging Face Hub. Chances are someone has already pretrained a model on your domain!

Let's start by loading the pretrained tokenizer, tokenizing our dataset, and getting rid of the columns we don't need for training and evaluation:

```
import torch
from transformers import (AutoTokenizer, AutoConfig,
                          AutoModelForSequenceClassification)

model_ckpt = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)

def tokenize(batch):
    return tokenizer(batch["text"], truncation=True, max_length=128)
ds_enc = ds.map(tokenize, batched=True)
ds_enc = ds_enc.remove_columns(['labels', 'text'])
```

The multilabel loss function expects the labels to be of type float, since it also allows for class probabilities instead of discrete labels. Therefore, we need to change the type of the column `label_ids`. Since changing the format of the column element-wise does not play well with Arrow's typed format, we'll do a little workaround. First, we create a new column with the labels. The format of that column is inferred from the first element. Then we delete the original column and rename the new one to take the place of the original one:

```
ds_enc.set_format("torch")
ds_enc = ds_enc.map(lambda x: {"label_ids_f": x["label_ids"].to(torch.float)},
                    remove_columns=["label_ids"])
ds_enc = ds_enc.rename_column("label_ids_f", "label_ids")
```

Since we are likely to quickly overfit the training data due to its limited size, we set `load_best_model_at_end=True` and choose the best model based on the micro F_1-score:

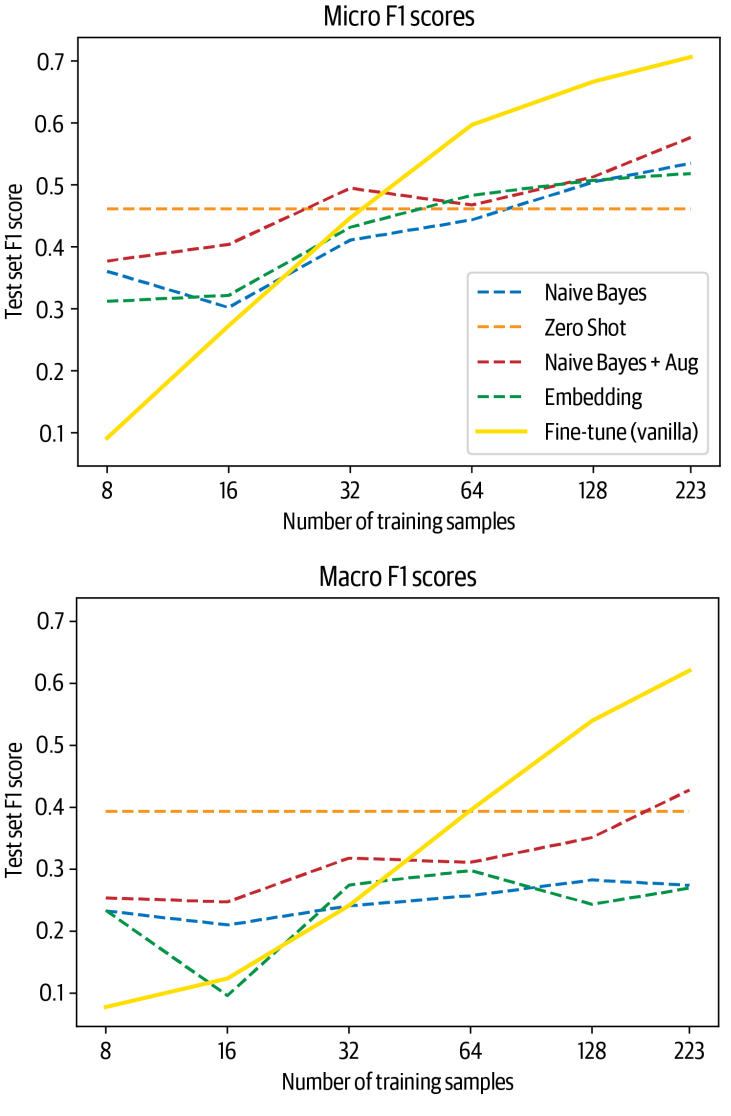
```
from transformers import Trainer, TrainingArguments

training_args_fine_tune = TrainingArguments(
    output_dir="./results", num_train_epochs=20, learning_rate=3e-5,
    lr_scheduler_type='constant', per_device_train_batch_size=4,
    per_device_eval_batch_size=32, weight_decay=0.0,
    evaluation_strategy="epoch", save_strategy="epoch",logging_strategy="epoch",
    load_best_model_at_end=True, metric_for_best_model='micro f1',
    save_total_limit=1, log_level='error')
```

We need the F_1-score to choose the best model, so we need to make sure it is calculated during the evaluation. Because the model returns the logits, we first need to normalize the predictions with a sigmoid function and can then binarize them with a simple threshold. Then we return the scores we are interested in from the classification report:

```
from scipy.special import expit as sigmoid

def compute_metrics(pred):
    y_true = pred.label_ids
    y_pred = sigmoid(pred.predictions)
    y_pred = (y_pred>0.5).astype(float)

    clf_dict = classification_report(y_true, y_pred, target_names=all_labels,
                                     zero_division=0, output_dict=True)
    return {"micro f1": clf_dict["micro avg"]["f1-score"],
            "macro f1": clf_dict["macro avg"]["f1-score"]}
```

Now we are ready to rumble! For each training set slice we train a classifier from scratch, load the best model at the end of the training loop, and store the results on the test set:

```
config = AutoConfig.from_pretrained(model_ckpt)
config.num_labels = len(all_labels)
config.problem_type = "multi_label_classification"

for train_slice in train_slices:
    model = AutoModelForSequenceClassification.from_pretrained(model_ckpt,
                                                               config=config)
    trainer = Trainer(
        model=model, tokenizer=tokenizer,
        args=training_args_fine_tune,
        compute_metrics=compute_metrics,
        train_dataset=ds_enc["train"].select(train_slice),
        eval_dataset=ds_enc["valid"],)

    trainer.train()
    pred = trainer.predict(ds_enc["test"])
    metrics = compute_metrics(pred)
    macro_scores["Fine-tune (vanilla)"].append(metrics["macro f1"])
    micro_scores["Fine-tune (vanilla)"].append(metrics["micro f1"])

plot_metrics(micro_scores, macro_scores, train_samples, "Fine-tune (vanilla)")
```



First of all we see that simply fine-tuning a vanilla BERT model on the dataset leads to competitive results when we have access to around 64 examples. We also see that before this the behavior is a bit erratic, which is again due to training a model on a small sample where some labels can be unfavorably unbalanced. Before we make use of the unlabeled part of our dataset, let's take a quick look at another promising approach for using language models in the few-shot domain.

In-Context and Few-Shot Learning with Prompts

We saw earlier in this chapter that we can use a language model like BERT or GPT-2 and adapt it to a supervised task by using prompts and parsing the model's token predictions. This is different from the classic approach of adding a task-specific head and tuning the model parameters for the task. On the plus side, this approach does not require any training data, but on the negative side it seems we can't leverage labeled data if we have access to it. There is a middle ground that we can sometimes take advantage of called *in-context* or *few-shot learning*.

To illustrate the concept, consider an English to French translation task. In the zero-shot paradigm, we would construct a prompt that might look as follows:

```
prompt = """\
Translate English to French:
thanks =>
"""
```

This hopefully prompts the model to predict the tokens of the word "merci". We already saw when using GPT-2 for summarization in Chapter 6 that adding "TL;DR" to a text prompted the model to generate a summary without explicitly being trained to do this. An interesting finding of the GPT-3 paper was the ability of large language models to effectively learn from examples presented in the prompt—so, the previous translation example could be augmented with several English to German examples, which would make the model perform much better on this task.[6]

Furthermore, the authors found that the larger the models are scaled, the better they are at using the in-context examples, leading to significant performance boosts. Although GPT-3-sized models are challenging to use in production, this is an exciting emerging research field and people have built cool applications, such as a natural language shell where commands are entered in natural language and parsed by GPT-3 to shell commands.

An alternative approach to using labeled data is to create examples of the prompts and desired predictions and continue training the language model on these examples. A novel method called ADAPET uses such an approach and beats GPT-3 on a wide variety of tasks,[7] tuning the model with generated prompts. Recent work by Hugging Face researchers suggests that such an approach can be more data-efficient than fine-tuning a custom head.[8]

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

7 D. Tam et al., "Improving and Simplifying Pattern Exploiting Training", (2021).

8 T. Le Scao and A.M. Rush, "How Many Data Points Is a Prompt Worth?", (2021).

In this section we briefly looked at various ways to make good use of the few labeled examples that we have. Very often we also have access to a lot of unlabeled data in addition to the labeled examples; in the next section we'll discuss how to make good use of that.

Leveraging Unlabeled Data

Although having access to large volumes of high-quality labeled data is the best-case scenario to train a classifier, this does not mean that unlabeled data is worthless. Just think about the pretraining of most models we have used: even though they are trained on mostly unrelated data from the internet, we can leverage the pretrained weights for other tasks on a wide variety of texts. This is the core idea of transfer learning in NLP. Naturally, if the downstream task has similar textual structure as the pretraining texts the transfer works better, so if we can bring the pretraining task closer to the downstream objective we could potentially improve the transfer.

Let's think about this in terms of our concrete use case: BERT is pretrained on the BookCorpus and English Wikipedia, and texts containing code and GitHub issues are definitely a small niche in these datasets. If we pretrained BERT from scratch we could do it on a crawl of all of the issues on GitHub, for example. However, this would be expensive, and a lot of aspects about language that BERT has learned are still valid for GitHub issues. So is there a middle ground between retraining from scratch and just using the model as is for classification? There is, and it is called domain adaptation (which we also saw for question answering in Chapter 7). Instead of retraining the language model from scratch, we can continue training it on data from our domain. In this step we use the classic language model objective of predicting masked words, which means we don't need any labeled data. After that we can load the adapted model as a classifier and fine-tune it, thus leveraging the unlabeled data.

The beauty of domain adaptation is that compared to labeled data, unlabeled data is often abundantly available. Furthermore, the adapted model can be reused for many use cases. Imagine you want to build an email classifier and apply domain adaptation on all your historic emails. You can later use the same model for named entity recognition or another classification task like sentiment analysis, since the approach is agnostic to the downstream task.

Let's now see the steps we need to take to fine-tune a pretrained language model.

Fine-Tuning a Language Model

In this section we'll fine-tune the pretrained BERT model with masked language modeling on the unlabeled portion of our dataset. To do this we only need two new

concepts: an extra step when tokenizing the data and a special data collator. Let's start with the tokenization.

In addition to the ordinary tokens from the text the tokenizer also adds special tokens to the sequence, such as the `[CLS]` and `[SEP]` tokens that are used for classification and next sentence prediction. When we do masked language modeling, we want to make sure we don't train the model to also predict these tokens. For this reason we mask them from the loss, and we can get a mask when tokenizing by setting `return_special_tokens_mask=True`. Let's retokenize the text with that setting:

```
def tokenize(batch):
    return tokenizer(batch["text"], truncation=True,
                     max_length=128, return_special_tokens_mask=True)

ds_mlm = ds.map(tokenize, batched=True)
ds_mlm = ds_mlm.remove_columns(["labels", "text", "label_ids"])
```

What's missing to start with masked language modeling is the mechanism to mask tokens in the input sequence and have the target tokens in the outputs. One way we could approach this is by setting up a function that masks random tokens and creates labels for these sequences. But this would double the size of the dataset, since we would also store the target sequence in the dataset, and it would mean we would use the same masking of a sequence every epoch.

A much more elegant solution is to use a data collator. Remember that the data collator is the function that builds the bridge between the dataset and the model calls. A batch is sampled from the dataset, and the data collator prepares the elements in the batch to feed them to the model. In the simplest case we have encountered, it simply concatenates the tensors of each element into a single tensor. In our case we can use it to do the masking and label generation on the fly. That way we don't need to store the labels and we get new masks every time we sample. The data collator for this task is called `DataCollatorForLanguageModeling`. We initialize it with the model's tokenizer and the fraction of tokens we want to mask via the `mlm_probability` argument. We'll use this collator to mask 15% of the tokens, which follows the procedure in the BERT paper:

```
from transformers import DataCollatorForLanguageModeling, set_seed

data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer,
                                                mlm_probability=0.15)
```

Let's have a quick look at the data collator in action to see what it actually does. To quickly show the results in a `DataFrame`, we switch the return formats of the tokenizer and the data collator to NumPy:

```
set_seed(3)
data_collator.return_tensors = "np"
inputs = tokenizer("Transformers are awesome!", return_tensors="np")
```

```
outputs = data_collator([{"input_ids": inputs["input_ids"][0]}])

pd.DataFrame({
    "Original tokens": tokenizer.convert_ids_to_tokens(inputs["input_ids"][0]),
    "Masked tokens": tokenizer.convert_ids_to_tokens(outputs["input_ids"][0]),
    "Original input_ids": original_input_ids,
    "Masked input_ids": masked_input_ids,
    "Labels": outputs["labels"][0]}).T
```

	0	1	2	3	4	5
Original tokens	[CLS]	transformers	are	awesome	!	[SEP]
Masked tokens	[CLS]	transformers	are	awesome	[MASK]	[SEP]
Original input_ids	101	19081	2024	12476	999	102
Masked input_ids	101	19081	2024	12476	103	102
Labels	-100	-100	-100	-100	999	-100

We see that the token corresponding to the exclamation mark has been replaced with a mask token. In addition, the data collator returned a label array, which is –100 for the original tokens and the token ID for the masked tokens. As we have seen previously, the entries containing –100 are ignored when calculating the loss. Let's switch the format of the data collator back to PyTorch:

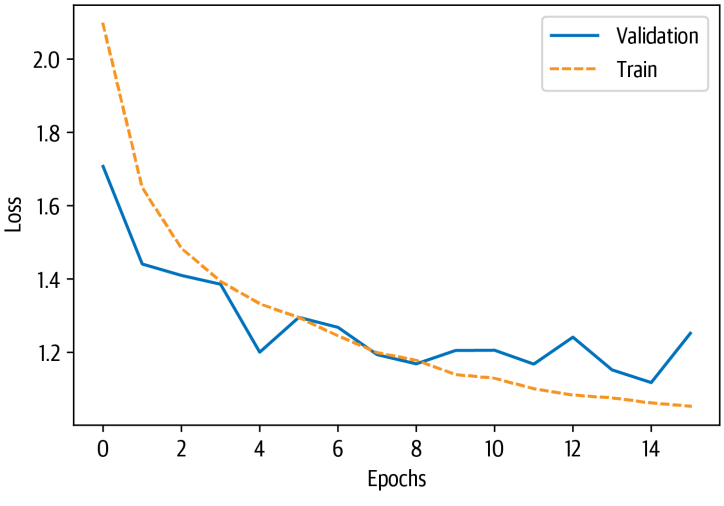
```
data_collator.return_tensors = "pt"
```

With the tokenizer and data collator in place, we are ready to fine-tune the masked language model. We set up the `TrainingArguments` and `Trainer` as usual:

```
from transformers import AutoModelForMaskedLM

training_args = TrainingArguments(
    output_dir = f"{model_ckpt}-issues-128", per_device_train_batch_size=32,
    logging_strategy="epoch", evaluation_strategy="epoch", save_strategy="no",
    num_train_epochs=16, push_to_hub=True, log_level="error", report_to="none")

trainer = Trainer(
        model=AutoModelForMaskedLM.from_pretrained("bert-base-uncased"),
        tokenizer=tokenizer, args=training_args, data_collator=data_collator,
        train_dataset=ds_mlm["unsup"], eval_dataset=ds_mlm["train"])

trainer.train()

trainer.push_to_hub("Training complete!")
```

We can access the trainer's log history to look at the training and validation losses of the model. All logs are stored in `trainer.state.log_history` as a list of dictionaries that we can easily load into a Pandas `DataFrame`. Since the training and validation loss are recorded at different steps, there are missing values in the dataframe. For this reason we drop the missing values before plotting the metrics:

```
df_log = pd.DataFrame(trainer.state.log_history)

(df_log.dropna(subset=["eval_loss"]).reset_index()["eval_loss"]
 .plot(label="Validation"))
df_log.dropna(subset=["loss"]).reset_index()["loss"].plot(label="Train")

plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend(loc="upper right")
plt.show()
```



It seems that both the training and validation loss went down considerably. So let's check if we can also see an improvement when we fine-tune a classifier based on this model.

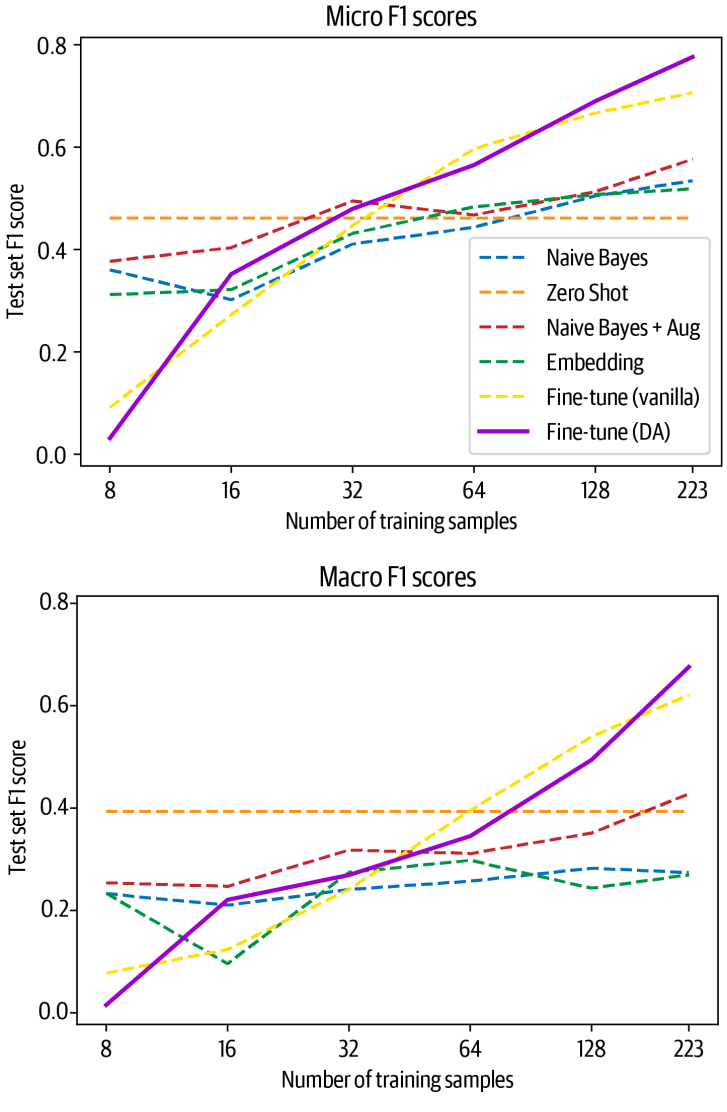
Fine-Tuning a Classifier

Now we'll repeat the fine-tuning procedure, but with the slight difference that we load our own custom checkpoint:

```
model_ckpt = f'{model_ckpt}-issues-128'
config = AutoConfig.from_pretrained(model_ckpt)
config.num_labels = len(all_labels)
config.problem_type = "multi_label_classification"

for train_slice in train_slices:
    model = AutoModelForSequenceClassification.from_pretrained(model_ckpt,
                                                               config=config)
    trainer = Trainer(
        model=model,
        tokenizer=tokenizer,
        args=training_args_fine_tune,
        compute_metrics=compute_metrics,
        train_dataset=ds_enc["train"].select(train_slice),
        eval_dataset=ds_enc["valid"],
    )

    trainer.train()
    pred = trainer.predict(ds_enc['test'])
    metrics = compute_metrics(pred)
    # DA refers to domain adaptation
    macro_scores['Fine-tune (DA)'].append(metrics['macro f1'])
    micro_scores['Fine-tune (DA)'].append(metrics['micro f1'])
```

Comparing the results to the fine-tuning based on vanilla BERT, we see that we get an advantage especially in the low-data domain. We also gain a few percentage points in the regime where more labeled data is available:

```
plot_metrics(micro_scores, macro_scores, train_samples, "Fine-tune (DA)")
```



This highlights that domain adaptation can provide a slight boost to the model's performance with unlabeled data and little effort. Naturally, the more unlabeled data and the less labeled data you have, the more impact you will get with this method. Before we conclude this chapter, we'll show you a few more tricks for taking advantage of unlabeled data.

Advanced Methods

Fine-tuning the language model before tuning the classification head is a simple yet reliable method to boost performance. However, there are sophisticated methods than can leverage unlabeled data even further. We summarize a few of these methods here, which should provide a good starting point if you need more performance.

Unsupervised data augmentation

The key idea behind unsupervised data augmentation (UDA) is that a model's predictions should be consistent for an unlabeled example and a slightly distorted one. Such distortions are introduced with standard data augmentation strategies such as token replacement and back translation. Consistency is then enforced by minimizing the KL divergence between the predictions of the original and distorted examples. This process is illustrated in Figure 9-5, where the consistency requirement is incorporated by augmenting the cross-entropy loss with an additional term from the unlabeled examples. This means that one trains a model on the labeled data with the standard supervised approach, but constrains the model to make consistent predictions on the unlabeled data.

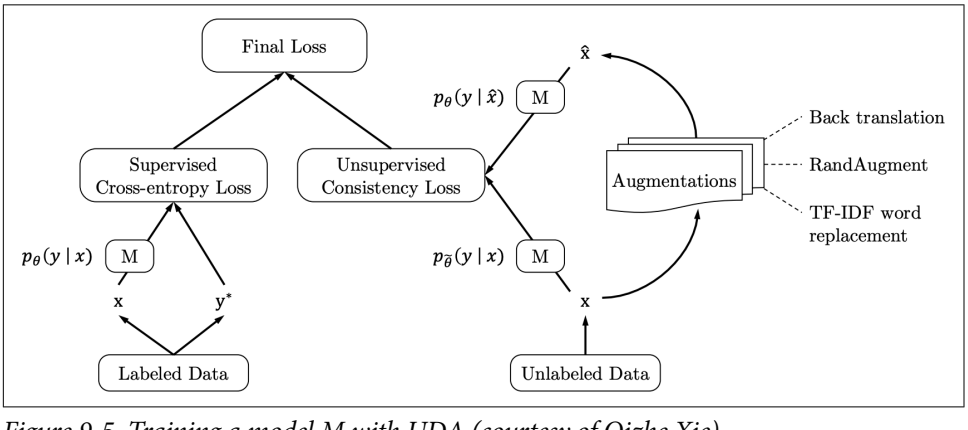


Figure 9-5. Training a model M with UDA (courtesy of Qizhe Xie)

The performance of this approach is quite impressive: with a handful of labeled examples, BERT models trained with UDA get similar performance to models trained on thousands of examples. The downside is that you need a data augmentation pipeline, and training takes much longer since you need multiple forward passes to generate the predicted distributions on the unlabeled and augmented examples.

Uncertainty-aware self-training

Another promising method to leverage unlabeled data is uncertainty-aware self-training (UST). The idea here is to train a teacher model on the labeled data and then

use that model to create pseudo-labels on the unlabeled data. Then a student is trained on the pseudo-labeled data, and after training it becomes the teacher for the next iteration.

One interesting aspect of this method is how the pseudo-labels are generated: to get an uncertainty measure of the model's predictions the same input is fed several times through the model with dropout turned on. Then the variance in the predictions gives a proxy for the certainty of the model on a specific sample. With that uncertainty measure the pseudo-labels are then sampled using a method called Bayesian Active Learning by Disagreement (BALD). The full training pipeline is illustrated in Figure 9-6.

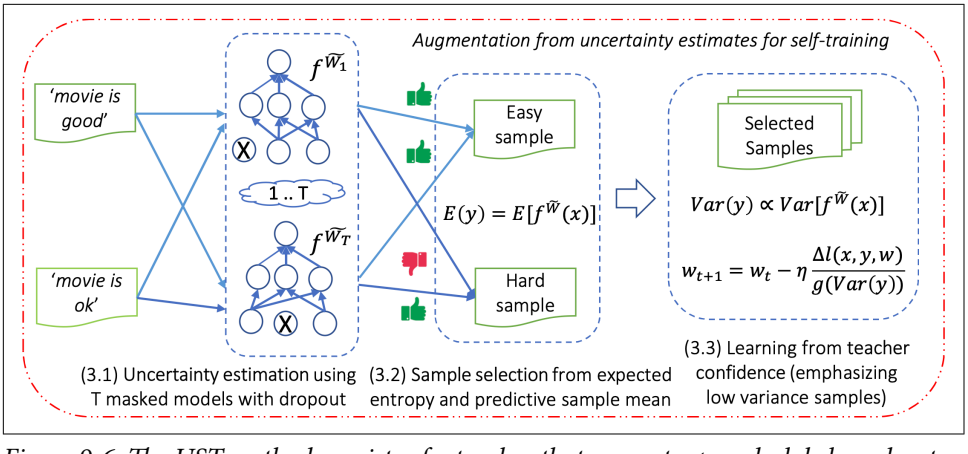


Figure 9-6. The UST method consists of a teacher that generates pseudo-labels and a student that is subsequently trained on those labels; after the student is trained it becomes the teacher and the step is repeated (courtesy of Subhabrata Mukherjee)[9]

With this iteration scheme the teacher continuously gets better at creating pseudo-labels, and thus the model's performance improves. In the end this approach gets within a few percent of models trained on the full training data with thousands of samples and even beats UDA on several datasets.

Now that we've seen a few advanced methods, let's take a step back and summarize what we've learned in this chapter.

9 S. Mukherjee and A.H. Awadallah, "Uncertainty-Aware Self-Training for Few-Shot Text Classification", (2020).

Conclusion

In this chapter we've seen that even if we have only a few or even no labels, not all hope is lost. We can utilize models that have been pretrained on other tasks, such as the BERT language model or GPT-2 trained on Python code, to make predictions on the new task of GitHub issue classification. Furthermore, we can use domain adaptation to get an additional boost when training the model with a normal classification head.

Which of the presented approaches will work best on a specific use case depends on a variety of aspects: how much labeled data you have, how noisy is it, how close the data is to the pretraining corpus, and so on. To find out what works best, it is a good idea to set up an evaluation pipeline and then iterate quickly. The flexible API of 🤗 Transformers allows you to quickly load a handful of models and compare them without the need for any code changes. There are over 10,000 models on the Hugging Face Hub, and chances are somebody has worked on a similar problem in the past and you can build on top of this.

One aspect that is beyond the scope of this book is the trade-off between a more complex approach like UDA or UST and getting more data. To evaluate your approach, it makes sense to at least build a validation and test set early on. At every step of the way you can also gather more labeled data. Usually annotating a few hundred examples is a matter of a couple of hours' or a few days' work, and there are many tools that can assist you in doing so. Depending on what you are trying to achieve, it can make sense to invest some time in creating a small, high-quality dataset rather than engineering a very complex method to compensate for the lack thereof. With the methods we've presented in this chapter you can ensure that you get the most value out of your precious labeled data.

Here, we have ventured into the low-data regime and seen that transformer models are still powerful even with just a hundred examples. In the next chapter we'll look at the complete opposite case: we'll see what we can do when we have hundreds of gigabytes of data and a lot of compute. We'll train a large transformer model from scratch to autocomplete code for us.

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.

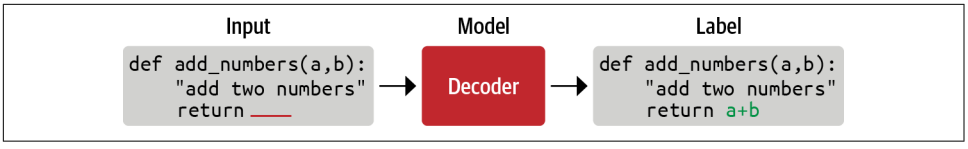


Figure 10-2. In causal language modeling, the future tokens are masked and the model has to predict them; typically a decoder model such as GPT is used for such a task

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.

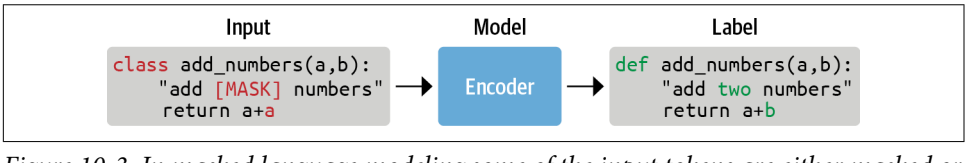


Figure 10-3. In masked language modeling some of the input tokens are either masked or replaced, and the model's task is to predict the original tokens; this is the architecture underlying the encoder branch of transformer models

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.

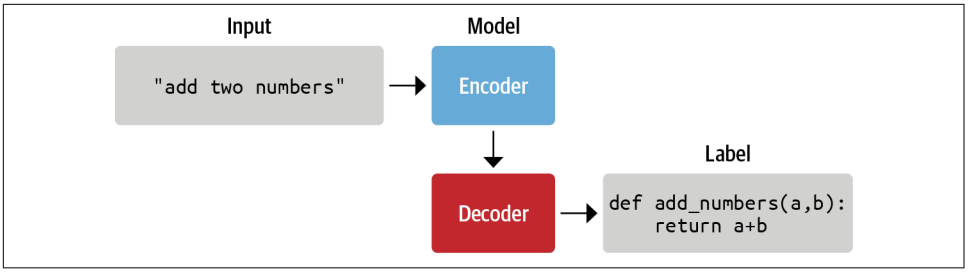


Figure 10-4. Using an encoder-decoder architecture for a sequence-to-sequence task where the inputs are split into comment/code pairs using heuristics: the model gets one element as input and needs to generate the other one

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.

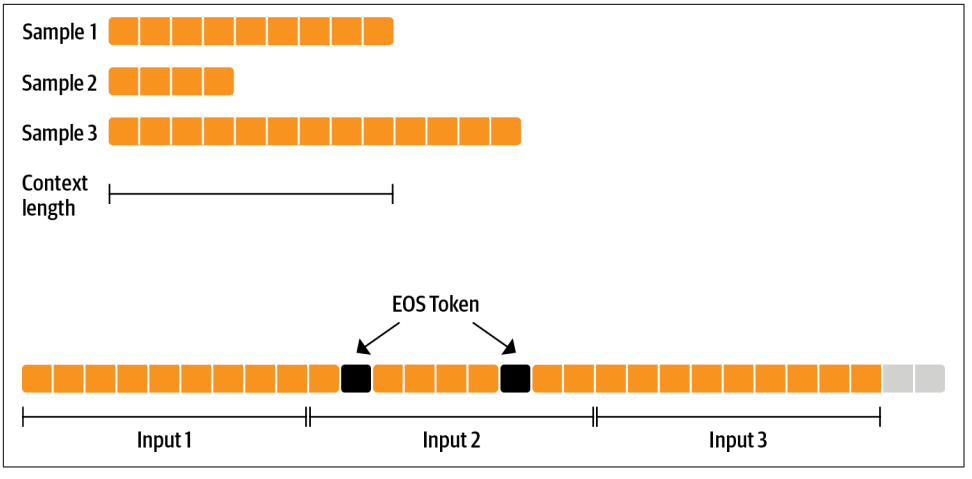


Figure 10-5. Preparing sequences of varying length for causal language modeling by concatenating several tokenized examples with an EOS token before chunking them

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.

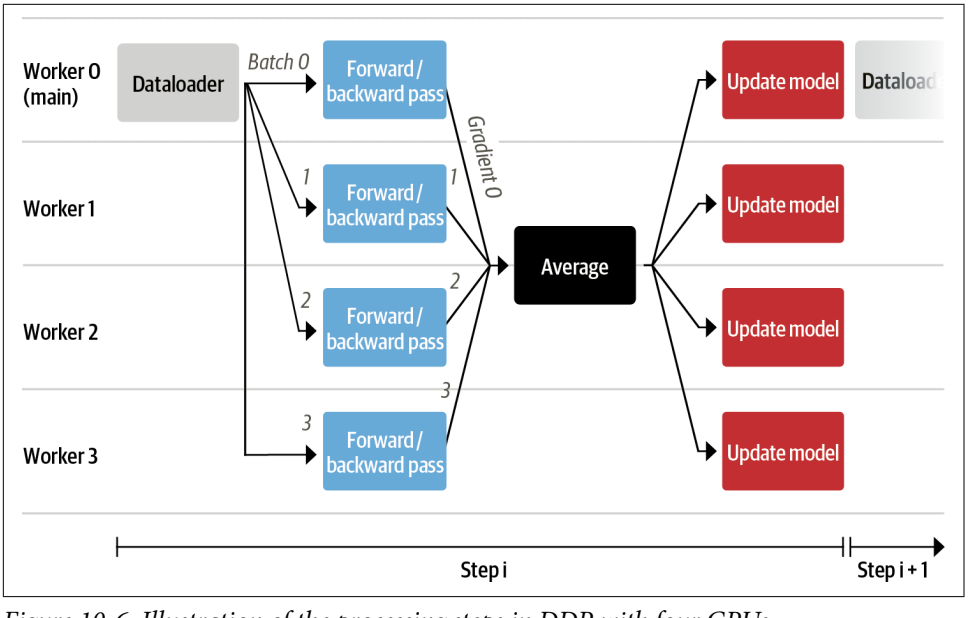


Figure 10-6. Illustration of the processing steps in DDP with four GPUs

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.

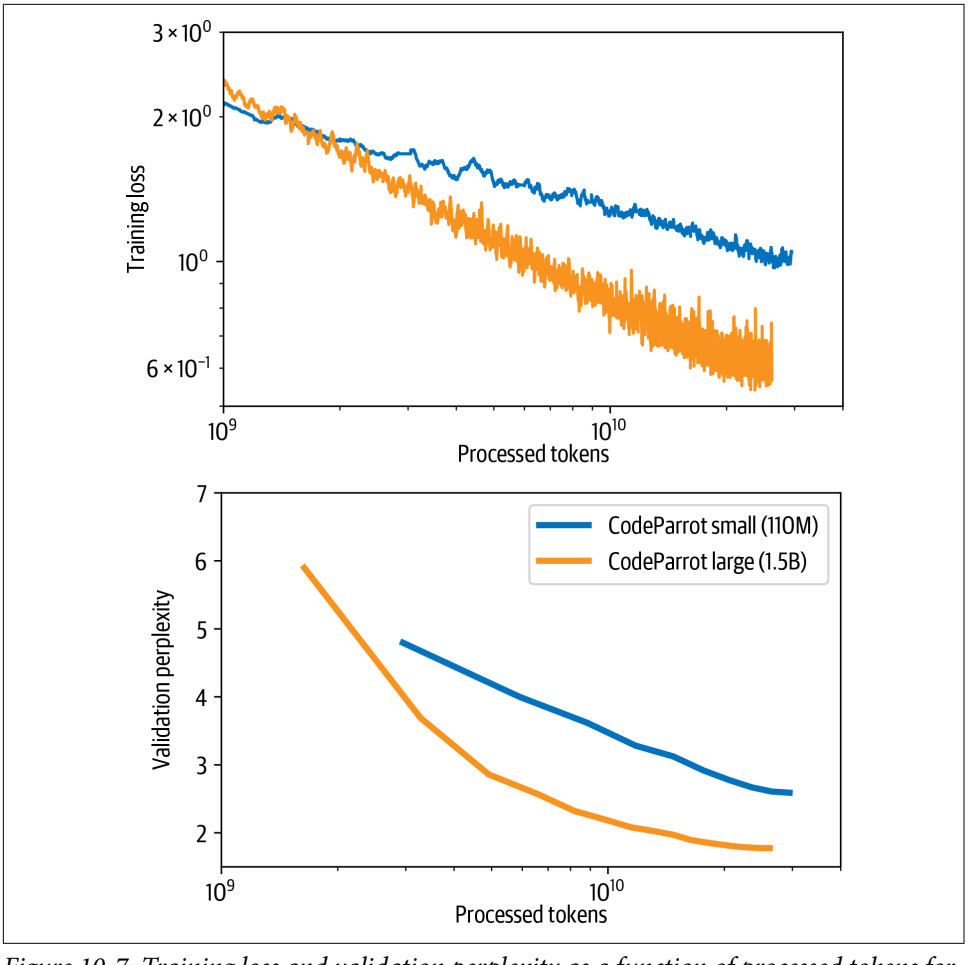


Figure 10-7. Training loss and validation perplexity as a function of processed tokens for the small and large CodeParrot models

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).

CHAPTER 11

Future Directions

Throughout this book we've explored the powerful capabilities of transformers across a wide range of NLP tasks. In this final chapter, we'll shift our perspective and look at some of the current challenges with these models and the research trends that are trying to overcome them. In the first part we explore the topic of scaling up transformers, both in terms of model and corpus size. Then we turn our attention toward various techniques that have been proposed to make the self-attention mechanism more efficient. Finally, we explore the emerging and exciting field of *multimodal transformers*, which can model inputs across multiple domains like text, images, and audio.

Scaling Transformers

In 2019, the researcher Richard Sutton wrote a provocative essay entitled "The Bitter Lesson" in which he argued that:

> The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin…. Seeking an improvement that makes a difference in the shorter term, researchers seek to leverage their human knowledge of the domain, but the only thing that matters in the long run is the leveraging of computation. These two need not run counter to each other, but in practice they tend to…. And the human-knowledge approach tends to complicate methods in ways that make them less suited to taking advantage of general methods leveraging computation.

The essay provides several historical examples, such as playing chess or Go, where the approach of encoding human knowledge within AI systems was ultimately outdone by increased computation. Sutton calls this the "bitter lesson" for the AI research field:

> We have to learn the bitter lesson that building in how we think we think does not work in the long run…. One thing that should be learned from the bitter lesson is the great power of general purpose methods, of methods that continue to scale with increased computation even as the available computation becomes very great. The two methods that seem to scale arbitrarily in this way are *search* and *learning*.

There are now signs that a similar lesson is at play with transformers; while many of the early BERT and GPT descendants focused on tweaking the architecture or pretraining objectives, the best-performing models in mid-2021, like GPT-3, are essentially basic scaled-up versions of the original models without many architectural modifications. In Figure 11-1 you can see a timeline of the development of the largest models since the release of the original Transformer architecture in 2017, which shows that model size has increased by over four orders of magnitude in just a few years!

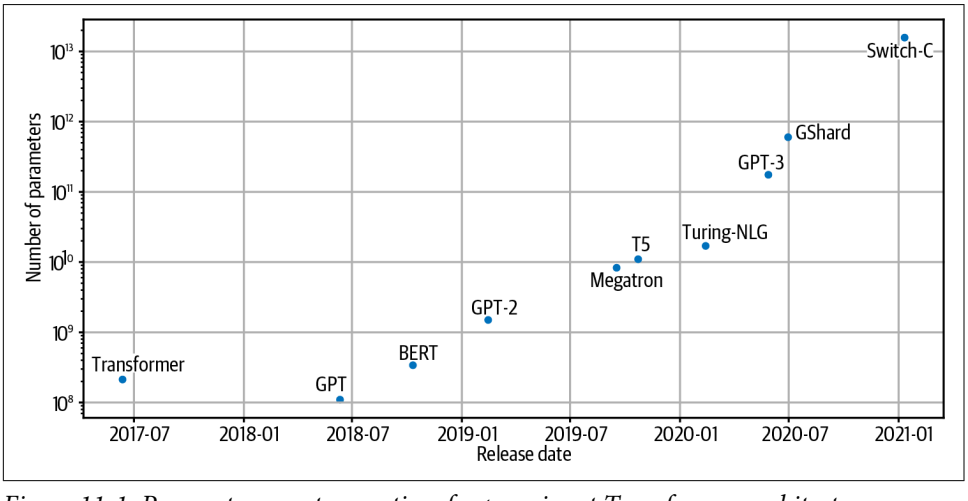


Figure 11-1. Parameter counts over time for prominent Transformer architectures

This dramatic growth is motivated by empirical evidence that large language models perform better on downstream tasks and that interesting capabilities such as zero-shot and few-shot learning emerge in the 10- to 100-billion parameter range. However, the number of parameters is not the only factor that affects model performance; the amount of compute and training data must also be scaled in tandem to train these monsters. Given that large language models like GPT-3 are estimated to cost $4.6 million to train, it is clearly desirable to be able to estimate the model's performance in advance. Somewhat surprisingly, the performance of language models appears to obey a *power law relationship* with model size and other factors that is codified in a set of scaling laws.[1] Let's take a look at this exciting area of research.

1 J. Kaplan et al., "Scaling Laws for Neural Language Models", (2020).

Scaling Laws

Scaling laws allow one to empirically quantify the "bigger is better" paradigm for language models by studying their behavior with varying compute budget C, dataset size D, and model size N.[2] The basic idea is to chart the dependence of the cross-entropy loss L on these three factors and determine if a relationship emerges. For autoregressive models like those in the GPT family, the resulting loss curves are shown in Figure 11-2, where each blue curve represents the training run of a single model.

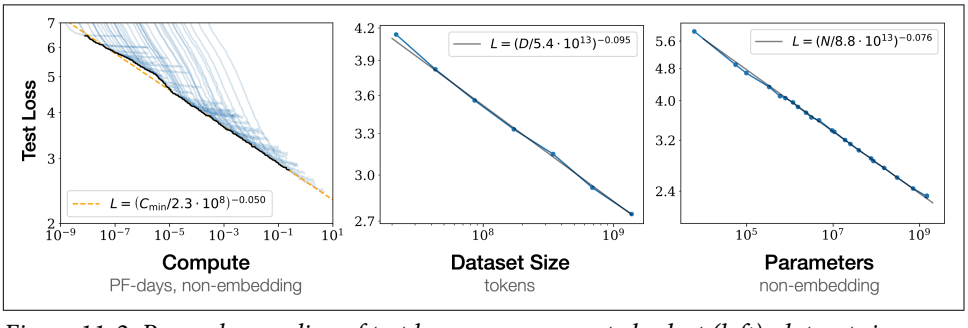


Figure 11-2. Power-law scaling of test loss versus compute budget (left), dataset size (middle), and model size (right) (courtesy of Jared Kaplan)

From these loss curves we can draw a few conclusions about:

The relationship of performance and scale
: Although many NLP researchers focus on architectural tweaks or hyperparameter optimization (like tuning the number of layers or attention heads) to improve performance on a fixed set of datasets, the implication of scaling laws is that a more productive path toward better models is to focus on increasing N, C, and D in tandem.

Smooth power laws
: The test loss L has a power law relationship with each of N, C, and D across several orders of magnitude (power law relationships are linear on a log-log scale). For $X = N, C, D$ we can express these power law relationships as $L(X) \sim 1/X^{\alpha}$, where α is a scaling exponent that is determined by a fit to the loss curves shown in Figure 11-2.[3] Typical values for α_X lie in the 0.05–0.095 range, and one attractive feature of these power laws is that the early part of a loss curve can be extrapolated to predict what the approximate loss would be if training was conducted for much longer.

2 The dataset size is measured in the number of tokens, while the model size excludes parameters from the embedding layers.

3 T. Henighan et al., "Scaling Laws for Autoregressive Generative Modeling", (2020).

Sample efficiency
: Large models are able to reach the same performance as smaller models with a smaller number of training steps. This can be seen by comparing the regions where a loss curve plateaus over some number of training steps, which indicates one gets diminishing returns in performance compared to simply scaling up the model.

Somewhat surprisingly, scaling laws have also been observed for other modalities, like images, videos, and mathematical problem solving, as illustrated in Figure 11-3.

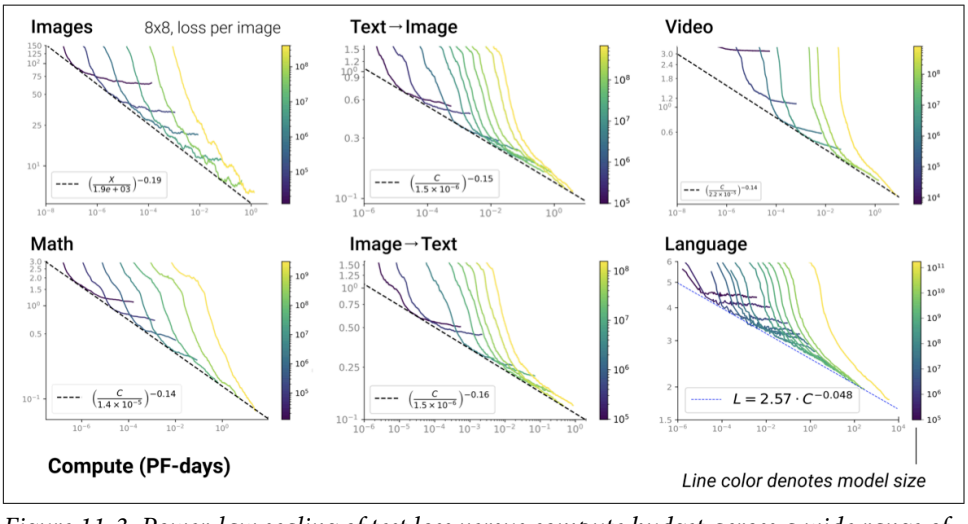


Figure 11-3. Power-law scaling of test loss versus compute budget across a wide range of modalities (courtesy of Tom Henighan)

Whether power-law scaling is a universal property of transformer language models is currently unknown. For now, we can use scaling laws as a tool to extrapolate large, expensive models without having to explicitly train them. However, scaling isn't quite as easy as it sounds. Let's now look at a few challenges that crop up when charting this frontier.

Challenges with Scaling

While scaling up sounds simple in theory ("just add more layers!"), in practice there are many difficulties. Here are a few of the biggest challenges you're likely to encounter when scaling language models:

Infrastructure
: Provisioning and managing infrastructure that potentially spans hundreds or thousands of nodes with as many GPUs is not for the faint-hearted. Are the required number of nodes available? Is communication between nodes a bottle-

neck? Tackling these issues requires a very different skill set than that found in most data science teams, and typically involves specialized engineers familiar with running large-scale, distributed experiments.

Cost

Most ML practitioners have experienced the feeling of waking up in the middle of the night in a cold sweat, remembering they forgot to shut down that fancy GPU on the cloud. This feeling intensifies when running large-scale experiments, and most companies cannot afford the teams and resources necessary to train models at the largest scales. Training a single GPT-3-sized model can cost several million dollars, which is not the kind of pocket change that many companies have lying around.[4]

Dataset curation

A model is only as good as the data it is trained on. Training large models requires large, high-quality datasets. When using terabytes of text data it becomes harder to make sure the dataset contains high-quality text, and even preprocessing becomes challenging. Furthermore, one needs to ensure that there is a way to control biases like sexism and racism that these language models can acquire when trained on large-scale webtext corpora. Another type of consideration revolves around licensing issues with the training data and personal information that can be embedded in large text datasets.

Model evaluation

Once the model is trained, the challenges don't stop. Evaluating the model on downstream tasks again requires time and resources. In addition, you'll want to probe the model for biased and toxic generations, even if you are confident that you created a clean dataset. These steps take time and need to be carried out thoroughly to minimize the risks of adverse effects later on.

Deployment

Finally, serving large language models also poses a significant challenge. In Chapter 8 we looked at a few approaches, such as distillation, pruning, and quantization, to help with these issues. However, this may not be enough if you are starting with a model that is hundreds of gigabytes in size. Hosted services such as the OpenAI API or Hugging Face's Accelerated Inference API are designed to help companies that cannot or do not want to deal with these deployment challenges.

4 However, recently a distributed deep learning framework has been proposed that enables smaller groups to pool their computational resources and pretrain models in a collaborative fashion. See M. Diskin et al., "Distributed Deep Learning in Open Collaborations", (2021).

This is by no means an exhaustive list, but it should give you an idea of the kinds of considerations and challenges that go hand in hand with scaling language models to ever larger sizes. While most of these efforts are centralized around a few institutions that have the resources and know-how to push the boundaries, there are currently two community-led projects that aim to produce and probe large language models in the open:

BigScience
: This is a one-year-long research workshop that runs from 2021 to 2022 and is focused on large language models. The workshop aims to foster discussions and reflections around the research questions surrounding these models (capabilities, limitations, potential improvements, bias, ethics, environmental impact, role in the general AI/cognitive research landscape) as well as the challenges around creating and sharing such models and datasets for research purposes and among the research community. The collaborative tasks involve creating, sharing, and evaluating a large multilingual dataset and a large language model. An unusually large compute budget was allocated for these collaborative tasks (several million GPU hours on several thousands GPUs). If successful, this workshop will run again in the future, focusing on involving an updated or different set of collaborative tasks. If you want to join the effort, you can find more information at the project's website.

EleutherAI
: This is a decentralized collective of volunteer researchers, engineers, and developers focused on AI alignment, scaling, and open source AI research. One of its aims is to train and open-source a GPT-3-sized model, and the group has already released some impressive models like GPT-Neo and GPT-J, which is a 6-billion-parameter model and currently the best-performing publicly available transformer in terms of zero-shot performance. You can find more information at EleutherAI's website.

Now that we've explored how to scale transformers across compute, model size, and dataset size, let's examine another active area of research: making self-attention more efficient.

Attention Please!

We've seen throughout this book that the self-attention mechanism plays a central role in the architecture of transformers; after all, the original Transformer paper is called "Attention Is All You Need"! However, there is a key challenge associated with self-attention: since the weights are generated from pairwise comparisons of all the tokens in a sequence, this layer becomes a computational bottleneck when trying to process long documents or apply transformers to domains like speech processing or computer vision. In terms of time and memory complexity, the self-attention layer of

the Transformer architecture naively scales like $\mathcal{O}\left(n^2\right)$, where n is the length of the sequence.[5]

As a result, much of the recent research on transformers has focused on making self-attention more efficient. The research directions are broadly clustered in Figure 11-4.

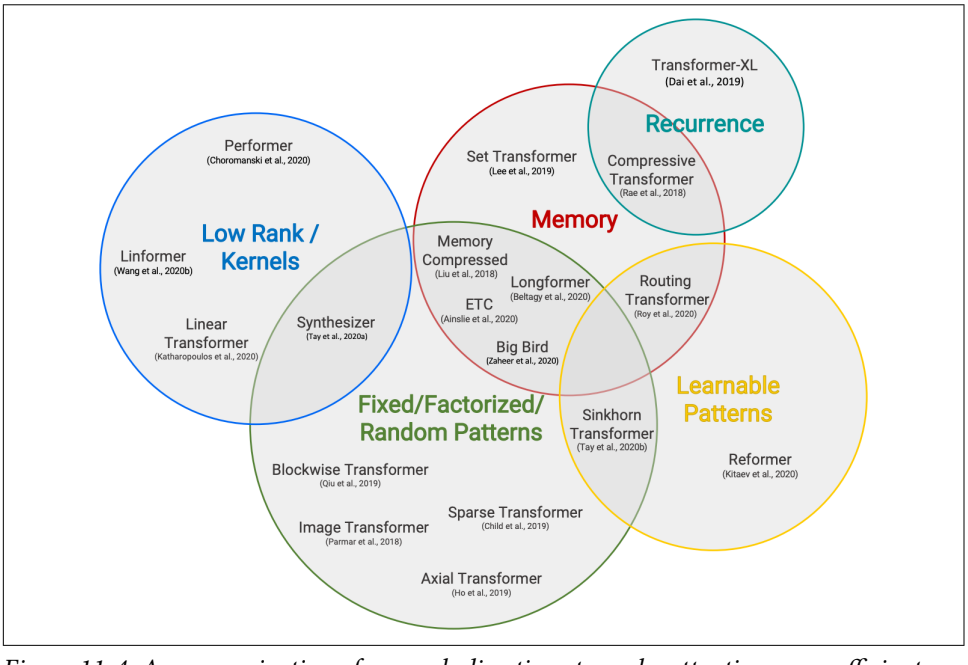


Figure 11-4. A summarization of research directions to make attention more efficient (courtesy of Yi Tay et al.)[6]

A common pattern is to make attention more efficient by introducing sparsity into the attention mechanism or by applying kernels to the attention matrix. Let's take a quick look at some of the most popular approaches to make self-attention more efficient, starting with sparsity.

Sparse Attention

One way to reduce the number of computations that are performed in the self-attention layer is to simply limit the number of query-key pairs that are generated

5 Although standard implementations of self-attention have $O\left(n^2\right)$ time and memory complexity, a recent paper by Google researchers shows that the memory complexity can be reduced to $O(\log n)$ via a simple reordering of the operations.

6 Yi Tay et al., "Efficient Transformers: A Survey", (2020).

according to some predefined pattern. There have been many sparsity patterns explored in the literature, but most of them can be decomposed into a handful of "atomic" patterns illustrated in Figure 11-5.

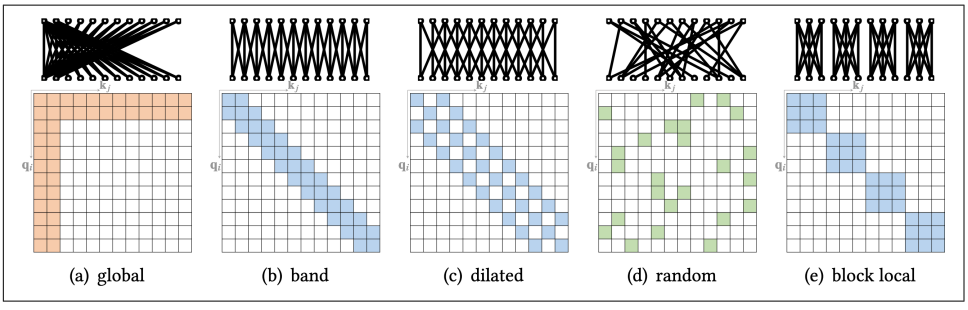


Figure 11-5. Common atomic sparse attention patterns for self-attention: a colored square means the attention score is calculated, while a blank square means the score is discarded (courtesy of Tianyang Lin)

We can describe these patterns as follows:[7]

Global attention
: Defines a few special tokens in the sequence that are allowed to attend to all other tokens

Band attention
: Computes attention over a diagonal band

Dilated attention
: Skips some query-key pairs by using a dilated window with gaps

Random attention
: Randomly samples a few keys for each query to compute attention scores

Block local attention
: Divides the sequence into blocks and restricts attention within these blocks

In practice, most transformer models with sparse attention use a mix of the atomic sparsity patterns shown in Figure 11-5 to generate the final attention matrix. As illustrated in Figure 11-6, models like Longformer use a mix of global and band attention, while BigBird adds random attention to the mix. Introducing sparsity into the attention matrix enables these models to process much longer sequences; in the case of Longformer and BigBird the maximum sequence length is 4,096 tokens, which is 8 times larger than BERT!

7 T. Lin et al., "A Survey of Transformers", (2021).

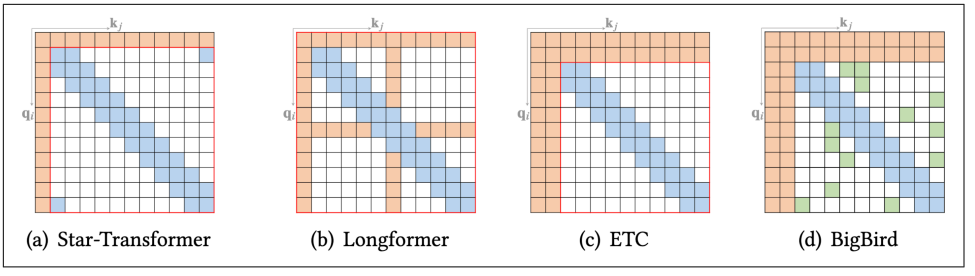


Figure 11-6. Sparse attention patterns for recent transformer models (courtesy of Tianyang Lin)

It is also possible to *learn* the sparsity pattern in a data-driven manner. The basic idea behind such approaches is to cluster the tokens into chunks. For example, Reformer uses a hash function to cluster similar tokens together.

Now that we've seen how sparsity can reduce the complexity of self-attention, let's take a look at another popular approach based on changing the operations directly.

Linearized Attention

An alternative way to make self-attention more efficient is to change the order of operations that are involved in computing the attention scores. Recall that to compute the self-attention scores of the queries and keys we need a similarity function, which for the transformer is just a simple dot product. However, for a general similarity function $\text{sim}(q_i, k_j)$ we can express the attention outputs as the following equation:

$$y_i = \sum_j \frac{\text{sim}(Q_i, K_j)}{\Sigma_k \text{sim}(Q_i, K_k)} V_j$$

The trick behind linearized attention mechanisms is to express the similarity function as a *kernel function* that decomposes the operation into two pieces:

$$\text{sim}(Q_j, K_j) = \phi(Q_i)^T \phi(K_j)$$

where ϕ is typically a high-dimensional feature map. Since $\phi(Q_i)$ is independent of j and k, we can pull it under the sums to write the attention outputs as follows:

$$y_i = \frac{\phi(Q_i)^T \Sigma_j \phi(K_j) V_j^T}{\phi(Q_i)^T \Sigma_k \phi(K_k)}$$

By first computing $\Sigma_j \phi(K_j) V_j^T$ and $\Sigma_k \phi(K_k)$, we can effectively linearize the space and time complexity of self-attention! The comparison between the two approaches is illustrated in Figure 11-7. Popular models that implement linearized self-attention include Linear Transformer and Performer.[8]

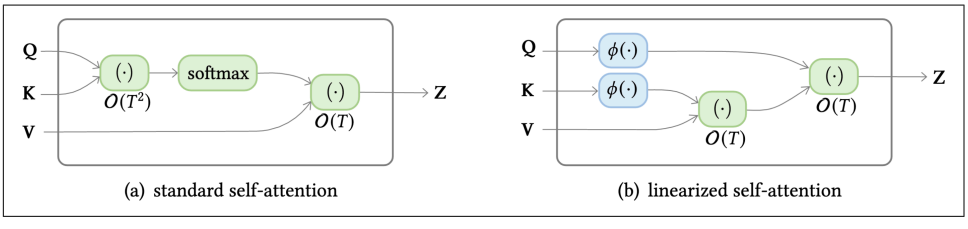


Figure 11-7. Complexity difference between standard self-attention and linearized self-attention (courtesy of Tianyang Lin)

In this section we've seen how Transformer architectures in general and attention in particular can be scaled up to achieve even better performance on a wide range of tasks. In the next section we'll have a look at how transformers are branching out of NLP into other domains such as audio and computer vision.

Going Beyond Text

Using text to train language models has been the driving force behind the success of transformer language models, in combination with transfer learning. On the one hand, text is abundant and enables self-supervised training of large models. On the other hand, textual tasks such as classification and question answering are common, and developing effective strategies for them allows us to address a wide range of real-world problems.

However, there are limits to this approach, including:

Human reporting bias
: The frequencies of events in text may not represent their true frequencies.[9] A model solely trained on text from the internet might have a very distorted image of the world.

8 A. Katharopoulos et al., "Transformers Are RNNs: Fast Autoregressive Transformers with Linear Attention", (2020); K. Choromanski et al., "Rethinking Attention with Performers", (2020).

9 J. Gordon and B. Van Durme, "Reporting Bias and Knowledge Extraction", (2013).

Common sense
: Common sense is a fundamental quality of human reasoning, but is rarely written down. As such, language models trained on text might know many facts about the world, but lack basic common-sense reasoning.

Facts
: A probabilistic language model cannot store facts in a reliable way and can produce text that is factually wrong. Similarly, such models can detect named entities, but have no direct way to access information about them.

Modality
: Language models have no way to connect to other modalities that could address the previous points, such as audio or visual signals or tabular data.

So, if we could solve the modality limitations we could potentially address some of the others as well. Recently there has been a lot of progress in pushing transformers to new modalities, and even building multimodal models. In this section we'll highlight a few of these advances.

Vision

Vision has been the stronghold of convolutional neural networks (CNNs) since they kickstarted the deep learning revolution. More recently, transformers have begun to be applied to this domain and to achieve efficiency similar to or better than CNNs. Let's have a look at a few examples.

iGPT

Inspired by the success of the GPT family of models with text, iGPT (short for image GPT) applies the same methods to images.[10] By viewing images as sequences of pixels, iGPT uses the GPT architecture and autoregressive pretraining objective to predict the next pixel values. Pretraining on large image datasets enables iGPT to "autocomplete" partial images, as displayed in Figure 11-8. It also achieves performant results on classification tasks when a classification head is added to the model.

10 M. Chen et al., "Generative Pretraining from Pixels," *Proceedings of the 37th International Conference on Machine Learning* 119 (2020):1691–1703, *https://proceedings.mlr.press/v119/chen20s.html*.

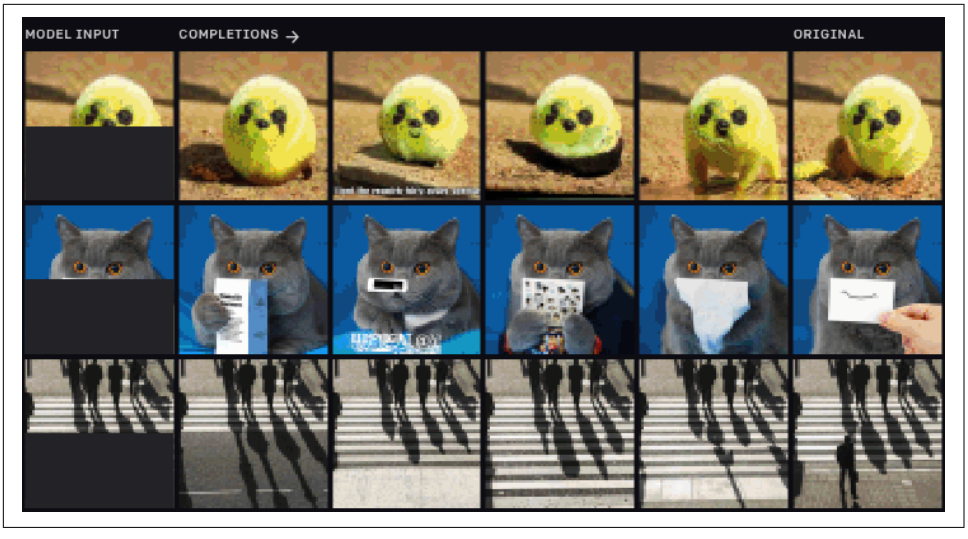


Figure 11-8. Examples of image completions with iGPT (courtesy of Mark Chen)

ViT

We saw that iGPT follows closely the GPT-style architecture and pretraining procedure. Vision Transformer (ViT)[11] is a BERT-style take on transformers for vision, as illustrated in Figure 11-9. First the image is split into smaller patches, and each of these patches is embedded with a linear projection. The results strongly resemble the token embeddings in BERT, and what follows is virtually identical. The patch embeddings are combined with position embeddings and then fed through an ordinary transformer encoder. During pretraining some of the patches are masked or distorted, and the objective is to predict the average color of the masked patch.

11 A. Dosovitskiy et al., "An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale", (2020).

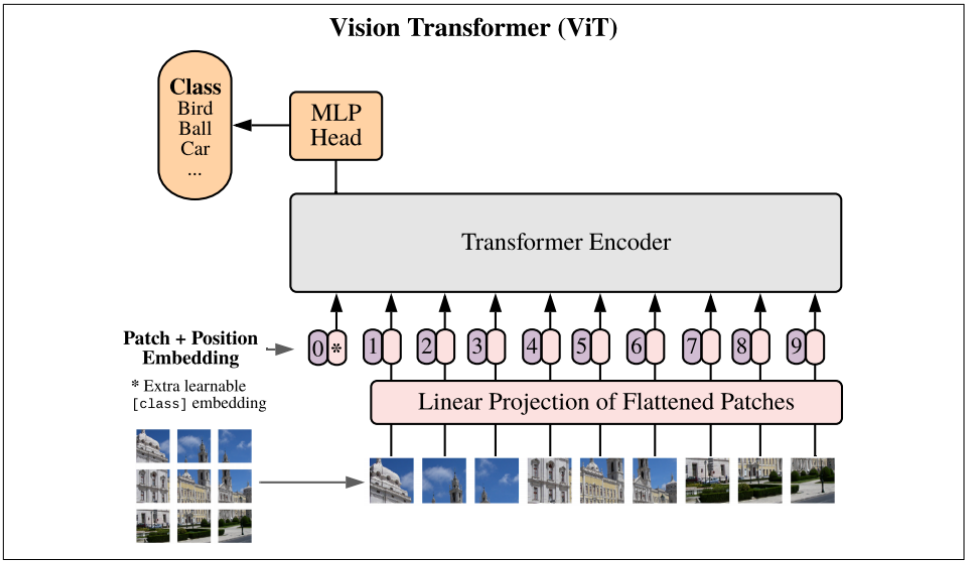
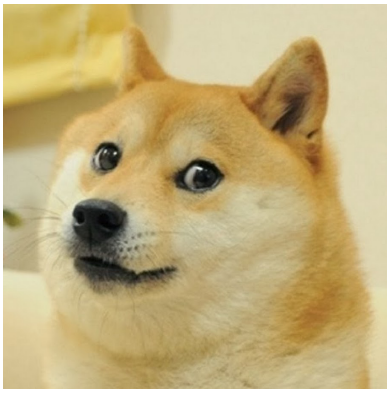


Figure 11-9. The ViT architecture (courtesy of Alexey Dosovitskiy et al.)

Although this approach did not produce better results when pretrained on the standard ImageNet dataset, it scaled significantly better than CNNs on larger datasets.

ViT is integrated in 🤗 Transformers, and using it is very similar to the NLP pipelines that we've used throughout this book. Let's start by loading the image of a rather famous dog:

```
from PIL import Image
import matplotlib.pyplot as plt

image = Image.open("images/doge.jpg")
plt.imshow(image)
plt.axis("off")
plt.show()
```



To load a ViT model, we just need to specify the `image-classification` pipeline, and then we feed in the image to extract the predicted classes:

```
import pandas as pd
from transformers import pipeline

image_classifier = pipeline("image-classification")
preds = image_classifier(image)
preds_df = pd.DataFrame(preds)
preds_df
```

	score	label
0	0.643599	Eskimo dog, husky
1	0.207407	Siberian husky
2	0.060160	dingo, warrigal, warragal, Canis dingo
3	0.035359	Norwegian elkhound, elkhound
4	0.012927	malamute, malemute, Alaskan malamute

Great, the predicted class seems to match the image!

A natural extension of image models is video models. In addition to the spatial dimensions, videos come with a temporal dimension. This makes the task more challenging as the volume of data gets much bigger and one needs to deal with the extra dimension. Models such as TimeSformer introduce a spatial and temporal attention mechanism to account for both.[12] In the future, such models can help build tools for a wide range of tasks such as classification or annotation of video sequences.

12 G. Bertasius, H. Wang, and L. Torresani, "Is Space-Time Attention All You Need for Video Understanding?", (2021).

Tables

A lot of data, such as customer data within a company, is stored in structured databases instead of as raw text. We saw in Chapter 7 that with question answering models we can query text with a question in natural text. Wouldn't it be nice if we could do the same with tables, as shown in Figure 11-10?

Table

Rank	Name	No. of reigns	Combined days
1	Lou Thesz	3	3,749
2	Ric Flair	8	3,103
3	Harley Race	7	1,799
4	Dory Funk Jr.	1	1,563
5	Dan Severn	2	1,559
6	Gene Kiniski	1	1,131

Example questions

#	Question	Answer	Example Type
1	*Which wrestler had the most number of reigns?*	Ric Flair	Cell selection
2	*Average time as champion for top 2 wrestlers?*	AVG(3749,3103)=3426	Scalar answer
3	*How many world champions are there with only one reign?*	COUNT(Dory Funk Jr., Gene Kiniski)=2	Ambiguous answer
4	*What is the number of reigns for* Harley Race?	7	Ambiguous answer
5	*Which of the following wrestlers were ranked in the bottom 3?*	{Dory Funk Jr., Dan Severn, Gene Kiniski}	Cell selection
	Out of these, who had more than one reign?	Dan Severn	Cell selection

Figure 11-10. Question answering over a table (courtesy of Jonathan Herzig)

TAPAS (short for Table Parser)[13] to the rescue! This model applies the Transformer architecture to tables by combining the tabular information with the query, as illustrated in Figure 11-11.

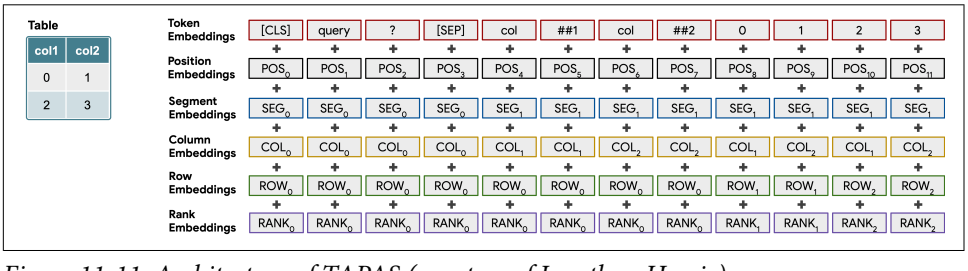


Figure 11-11. Architecture of TAPAS (courtesy of Jonathan Herzig)

Let's look at an example of how TAPAS works in practice. We have created a fictitious version of this book's table of contents. It contains the chapter number, the name of the chapter, as well as the starting and ending pages of the chapters:

```
book_data = [
    {"chapter": 0, "name": "Introduction", "start_page": 1, "end_page": 11},
    {"chapter": 1, "name": "Text classification", "start_page": 12,
     "end_page": 48},
    {"chapter": 2, "name": "Named Entity Recognition", "start_page": 49,
     "end_page": 73},
    {"chapter": 3, "name": "Question Answering", "start_page": 74,
```

13 J. Herzig et al., "TAPAS: Weakly Supervised Table Parsing via Pre-Training", (2020).

```
     "end_page": 120},
    {"chapter": 4, "name": "Summarization", "start_page": 121,
     "end_page": 140},
    {"chapter": 5, "name": "Conclusion", "start_page": 141,
     "end_page": 144}
]
```

We can also easily add the number of pages each chapter has with the existing fields. In order to play nicely with the TAPAS model, we need to make sure that all columns are of type `str`:

```
table = pd.DataFrame(book_data)
table['number_of_pages'] = table['end_page']-table['start_page']
table = table.astype(str)
table
```

	chapter	name	start_page	end_page	number_of_pages
0	0	Introduction	1	11	10
1	1	Text classification	12	48	36
2	2	Named Entity Recognition	49	73	24
3	3	Question Answering	74	120	46
4	4	Summarization	121	140	19
5	5	Conclusion	141	144	3

By now you should know the drill. We first load the `table-question-answering` pipeline:

```
table_qa = pipeline("table-question-answering")
```

and then pass some queries to extract the answers:

```
table_qa = pipeline("table-question-answering")
queries = ["What's the topic in chapter 4?",
           "What is the total number of pages?",
           "On which page does the chapter about question-answering start?",
           "How many chapters have more than 20 pages?"]
preds = table_qa(table, queries)
```

These predictions store the type of table operation in an `aggregator` field, along with the answer. Let's see how well TAPAS fared on our questions:

```
for query, pred in zip(queries, preds):
    print(query)
    if pred["aggregator"] == "NONE":
        print("Predicted answer: " + pred["answer"])
    else:
        print("Predicted answer: " + pred["answer"])
    print('='*50)
```

```
What's the topic in chapter 4?
Predicted answer: Summarization
==================================================
What is the total number of pages?
Predicted answer: SUM > 10, 36, 24, 46, 19, 3
==================================================
On which page does the chapter about question-answering start?
Predicted answer: AVERAGE > 74
==================================================
How many chapters have more than 20 pages?
Predicted answer: COUNT > 1, 2, 3
==================================================
```

For the first chapter, the model predicted exactly one cell with no aggregation. If we look at the table, we see that the answer is in fact correct. In the next example the model predicted all the cells containing the number of pages in combination with the sum aggregator, which again is the correct way of calculating the total number of pages. The answer to question three is also correct; the average aggregation is not necessary in that case, but it doesn't make a difference. Finally, we have a question that is a little bit more complex. To determine how many chapters have more than 20 pages we first need to find out which chapters satisfy that criterion and then count them. It seem that TAPAS again got it right and correctly determined that chapters 1, 2, and 3 have more than 20 pages, and added a count aggregator to the cells.

The kinds of questions we asked can also be solved with a few simple Pandas commands; however, the ability to ask questions in natural language instead of Python code allows a much wider audience to query the data to answer specific questions. Imagine such tools in the hands of business analysts or managers who are able verify their own hypotheses about the data!

Multimodal Transformers

So far we've looked at extending transformers to a single new modality. TAPAS is arguably multimodal since it combines text and tables, but the table is also treated as text. In this section we examine transformers that combine two modalities at once: audio plus text and vision plus text.

Speech-to-Text

Although being able to use text to interface with a computer is a huge step forward, using spoken language is an even more natural way for us to communicate. You can see this trend in industry, where applications such as Siri and Alexa are on the rise and becoming progressively more useful. Also, for a large fraction of the population, writing and reading are more challenging than speaking. So, being able to process and understand audio is not only convenient, but can help many people access more information. A common task in this domain is *automatic speech recognition* (ASR),

which converts spoken words to text and enables voice technologies like Siri to answer questions like "What is the weather like today?"

The wav2vec 2.0 family of models are one of the most recent developments in ASR: they use a transformer layer in combination with a CNN, as illustrated in Figure 11-12.[14] By leveraging unlabeled data during pretraining, these models achieve competitive results with only a few minutes of labeled data.

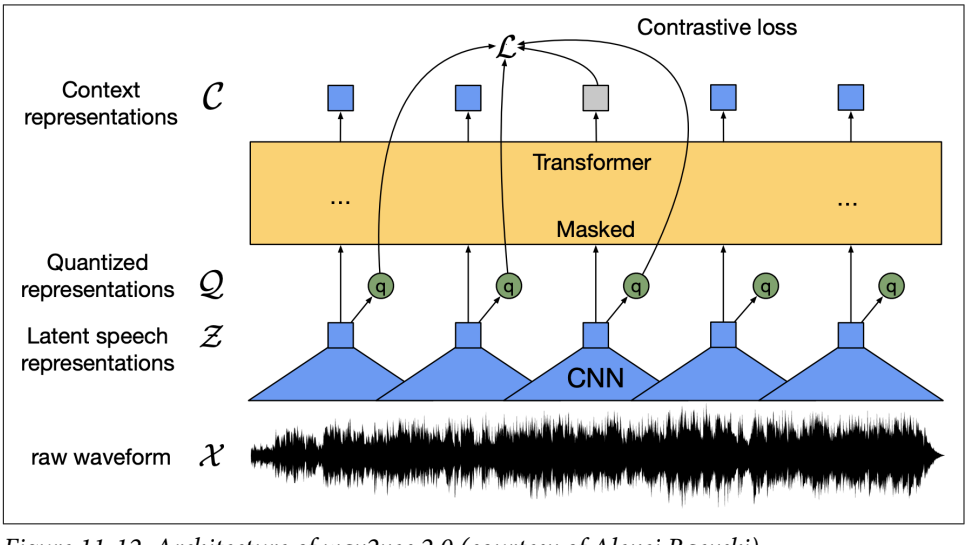


Figure 11-12. Architecture of wav2vec 2.0 (courtesy of Alexei Baevski)

The wav2vec 2.0 models are integrated in 🤗 Transformers, and you won't be surprised to learn that loading and using them follows the familiar steps that we have seen throughout this book. Let's load a pretrained model that was trained on 960 hours of speech audio:

```
asr = pipeline("automatic-speech-recognition")
```

To apply this model to some audio files we'll use the ASR subset of the SUPERB dataset, which is the same dataset the model was pretrained on. Since the dataset is quite large, we'll just load one example for our demo purposes:

```
from datasets import load_dataset

ds = load_dataset("superb", "asr", split="validation[:1]")
print(ds[0])
```

```
{'chapter_id': 128104, 'speaker_id': 1272, 'file': '~/.cache/huggingf
ace/datasets/downloads/extracted/e4e70a454363bec1c1a8ce336139866a39442114d86a433
```

14 A. Baevski et al., "wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations", (2020).

```
6014acd4b1ed55e55/LibriSpeech/dev-clean/1272/128104/1272-128104-0000.flac',
'id': '1272-128104-0000', 'text': 'MISTER QUILTER IS THE APOSTLE OF THE MIDDLE
CLASSES AND WE ARE GLAD TO WELCOME HIS GOSPEL'}
```

Here we can see that the audio in the `file` column is stored in the FLAC coding format, while the expected transcription is given by the `text` column. To convert the audio to an array of floats, we can use the *SoundFile* library to read each file in our dataset with `map()`:

```
import soundfile as sf

def map_to_array(batch):
    speech, _ = sf.read(batch["file"])
    batch["speech"] = speech
    return batch

ds = ds.map(map_to_array)
```

If you are using a Jupyter notebook you can easily play the sound files with the following `IPython` widgets:

```
from IPython.display import Audio

display(Audio(ds[0]['speech'], rate=16000))
```

Finally, we can pass the inputs to the pipeline and inspect the prediction:

```
pred = asr(ds[0]["speech"])
print(pred)
```

```
{'text': 'MISTER QUILTER IS THE APOSTLE OF THE MIDDLE CLASSES AND WE ARE GLAD TO
WELCOME HIS GOSPEL'}
```

This transcription seems to be correct. We can see that some punctuation is missing, but this is hard to get from audio alone and could be added in a postprocessing step. With only a handful of lines of code we can build ourselves a state-of-the-art speech-to-text application!

Building a model for a new language still requires a minimum amount of labeled data, which can be challenging to obtain, especially for low-resource languages. Soon after the release of wav2vec 2.0, a paper describing a method named wav2vec-U was published.[15] In this work, a combination of clever clustering and GAN training is used to build a speech-to-text model using only independent unlabeled speech and unlabeled text data. This process is visualized in detail in Figure 11-13. No aligned speech and text data is required at all, which enables the training of highly performant speech-to-text models for a much larger spectrum of languages.

15 A. Baevski et al., "Unsupervised Speech Recognition", (2021).

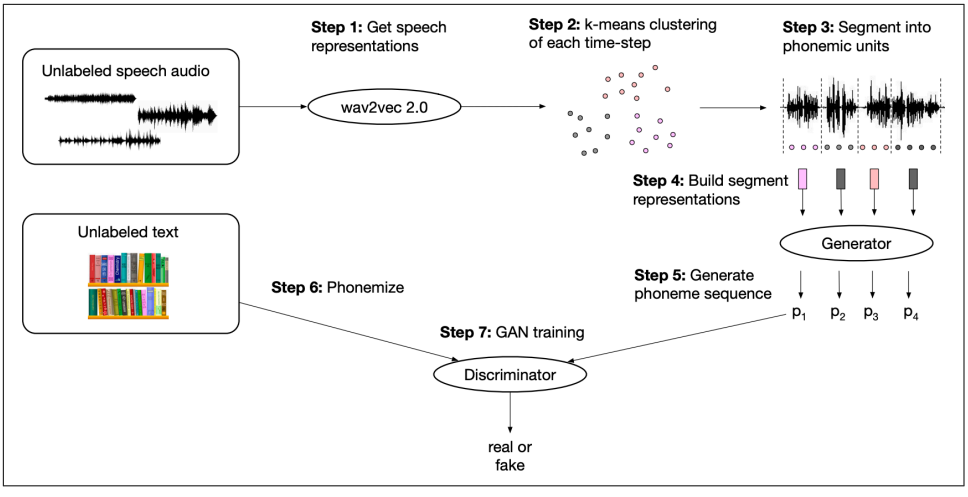


Figure 11-13. Training scheme for wav2vec-U (courtesy of Alexsei Baevski)

Great, so transformers can now "read" text and "hear" audio—can they also "see"? The answer is yes, and this is one of the current hot research frontiers in the field.

Vision and Text

Vision and text are another natural pair of modalities to combine since we frequently use language to communicate and reason about the contents of images and videos. In addition to the vision transformers, there have been several developments in the direction of combining visual and textual information. In this section we will look at four examples of models combining vision and text: VisualQA, LayoutLM, DALL·E, and CLIP.

VQA

In Chapter 7 we explored how we can use transformer models to extract answers to text-based questions. This can be done ad hoc to extract information from texts or offline, where the question answering model is used to extract structured information from a set of documents. There have been several efforts to expand this approach to vision with datasets such as VQA,[16] shown in Figure 11-14.

16 Y. Goyal et al., "Making the V in VQA Matter: Elevating the Role of Image Understanding in Visual Question Answering", (2016).

Figure 11-14. Example of a visual question answering task from the VQA dataset (courtesy of Yash Goyal)

Models such as LXMERT and VisualBERT use vision models like ResNets to extract features from the pictures and then use transformer encoders to combine them with the natural questions and predict an answer.[17]

LayoutLM

Analyzing scanned business documents like receipts, invoices, or reports is another area where extracting visual and layout information can be a useful way to recognize text fields of interest. Here the LayoutLM family of models are the current state of the art. They use an enhanced Transformer architecture that receives three modalities as input: text, image, and layout. Accordingly, as shown in Figure 11-15, there are embedding layers associated with each modality, a spatially aware self-attention mechanism, and a mix of image and text/image pretraining objectives to align the different modalities. By pretraining on millions of scanned documents, LayoutLM models are able to transfer to various downstream tasks in a manner similar to BERT for NLP.

17 H. Tan and M. Bansal, "LXMERT: Learning Cross-Modality Encoder Representations from Transformers", (2019); L.H. Li et al., "VisualBERT: A Simple and Performant Baseline for Vision and Language", (2019).

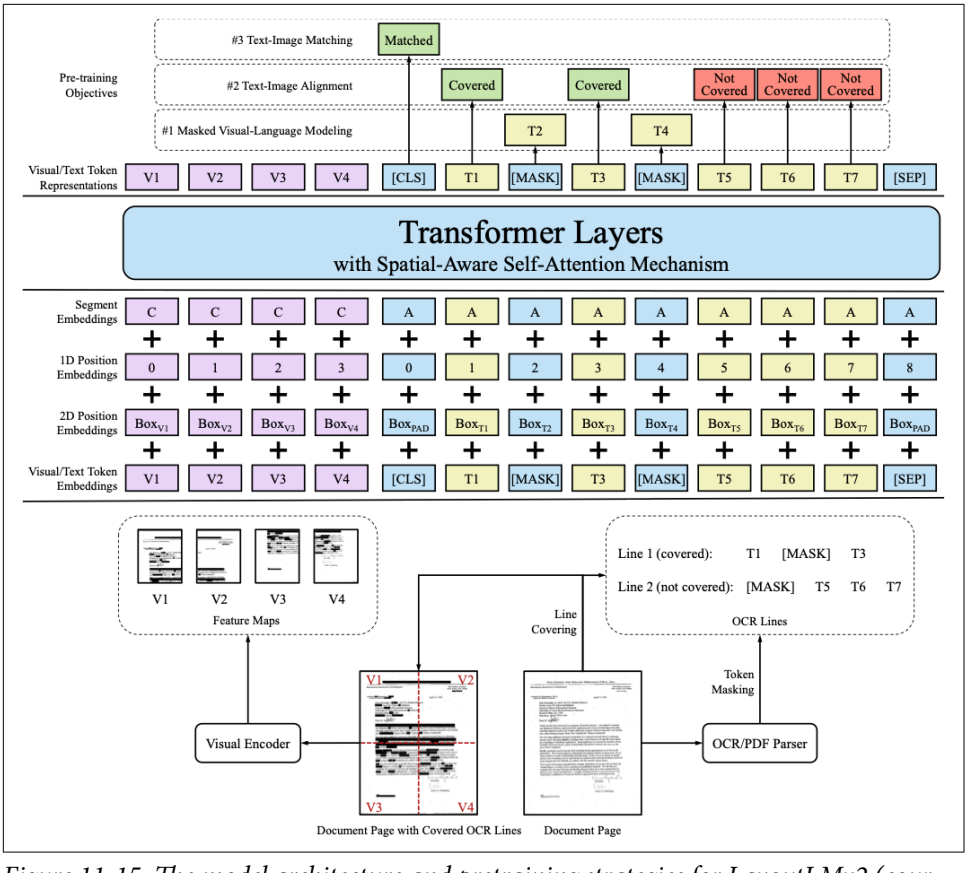


Figure 11-15. The model architecture and pretraining strategies for LayoutLMv2 (courtesy of Yang Xu)

DALL·E

A model that combines vision and text for *generative* tasks is DALL·E.[18] It uses the GPT architecture and autoregressive modeling to generate images from text. Inspired by iGPT, it regards the words and pixels as one sequence of tokens and is thus able to continue generating an image from a text prompt, as shown in Figure 11-16.

18 A. Ramesh et al., "Zero-Shot Text-to-Image Generation", (2021).

Figure 11-16. Generation examples with DALL·E (courtesy of Aditya Ramesh)

CLIP

Finally, let's have a look at CLIP,[19] which also combines text and vision but is designed for supervised tasks. Its creators constructed a dataset with 400 million image/caption pairs and used contrastive learning to pretrain the model. The CLIP architecture consists of a text and an image encoder (both transformers) that create embeddings of the captions and images. A batch of images with captions is sampled, and the contrastive objective is to maximize the similarity of the embeddings (as measured by the dot product) of the corresponding pair while minimizing the similarity of the rest, as illustrated in Figure 11-17.

In order to use the pretrained model for classification the possible classes are embedded with the text encoder, similar to how we used the zero-shot pipeline. Then the embeddings of all the classes are compared to the image embedding that we want to classify, and the class with the highest similarity is chosen.

19 A. Radford et al., "Learning Transferable Visual Models from Natural Language Supervision", (2021).

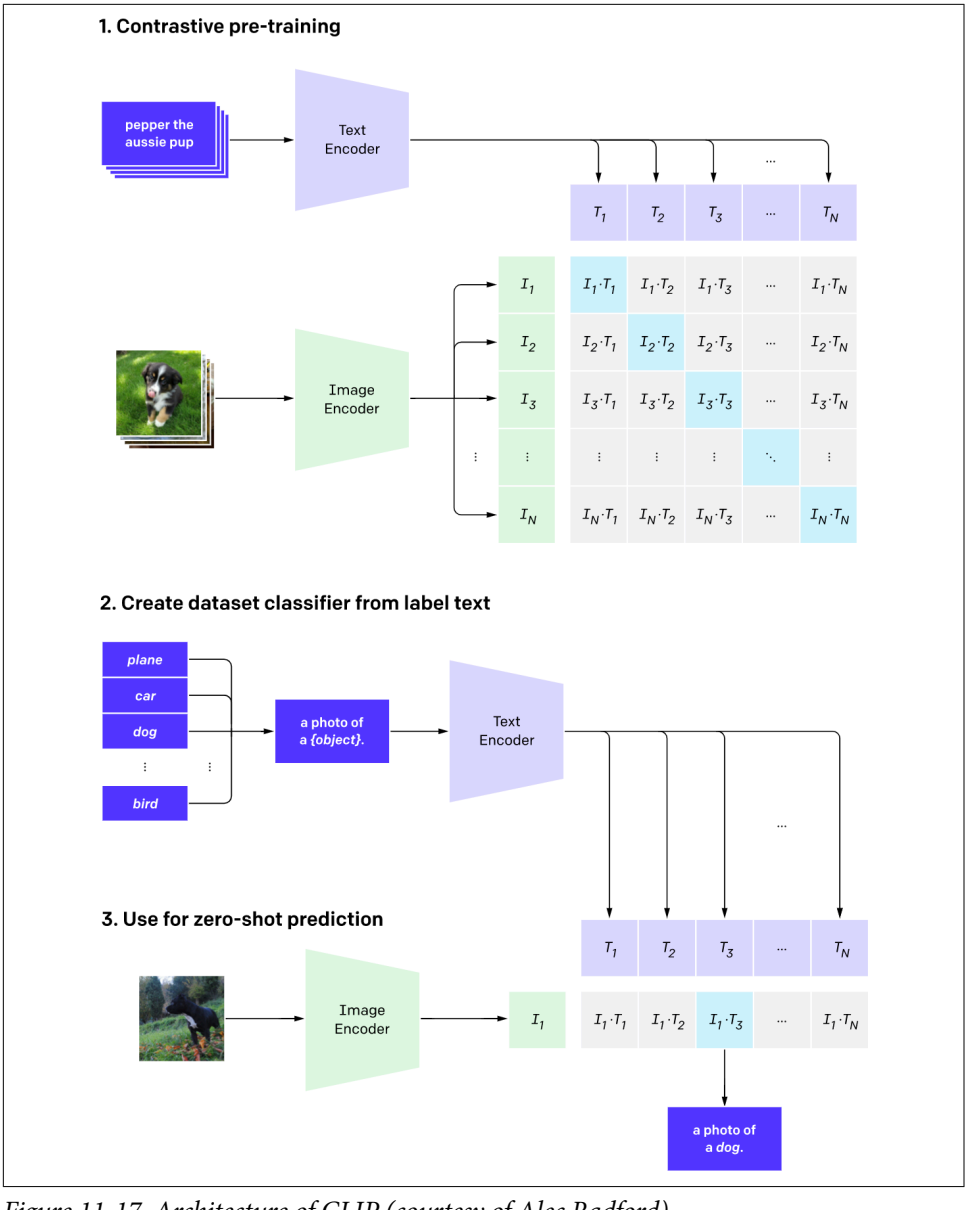


Figure 11-17. Architecture of CLIP (courtesy of Alec Radford)

The zero-shot image classification performance of CLIP is remarkable and competitive with fully supervised trained vision models, while being more flexible with regard to new classes. CLIP is also fully integrated in 🤗 Transformers, so we can try it out. For image-to-text tasks, we instantiate a *processor* that consists of a *feature extractor* and a tokenizer. The role of the feature extractor is to convert the image into a

form suitable for the model, while the tokenizer is responsible for decoding the model's predictions into text:

```
from transformers import CLIPProcessor, CLIPModel

clip_ckpt = "openai/clip-vit-base-patch32"
model = CLIPModel.from_pretrained(clip_ckpt)
processor = CLIPProcessor.from_pretrained(clip_ckpt)
```

Then we need a fitting image to try it out. What would be better suited than a picture of Optimus Prime?

```
image = Image.open("images/optimusprime.jpg")
plt.imshow(image)
plt.axis("off")
plt.show()
```

Next, we set up the texts to compare the image against and pass it through the model:

```
import torch

texts = ["a photo of a transformer", "a photo of a robot", "a photo of agi"]
inputs = processor(text=texts, images=image, return_tensors="pt", padding=True)
with torch.no_grad():
    outputs = model(**inputs)
logits_per_image = outputs.logits_per_image
probs = logits_per_image.softmax(dim=1)
probs
```

```
tensor([[0.9557, 0.0413, 0.0031]])
```

Well, it almost got the right answer (a photo of AGI of course). Jokes aside, CLIP makes image classification very flexible by allowing us to define classes through text instead of having the classes hardcoded in the model architecture. This concludes our tour of multimodal transformer models, but we hope we've whetted your appetite.

Where to from Here?

Well that's the end of the ride; thanks for joining us on this journey through the transformers landscape! Throughout this book we've explored how transformers can address a wide range of tasks and achieve state-of-the-art results. In this chapter we've seen how the current generation of models are being pushed to their limits with scaling and how they are also branching out into new domains and modalities.

If you want to reinforce the concepts and skills that you've learned in this book, here are a few ideas for where to go from here:

Join a Hugging Face community event
: Hugging Face hosts short sprints focused on improving the libraries in the ecosystem, and these events are a great way to meet the community and get a taste for open source software development. So far there have been sprints on adding 600+ datasets to 🤗 Datasets, fine-tuning 300+ ASR models in various languages, and implementing hundreds of projects in JAX/Flax.

Build your own project
: One very effective way to test your knowledge in machine learning is to build a project to solve a problem that interests you. You could reimplement a transformer paper, or apply transformers to a novel domain.

Contribute a model to 🤗 Transformers
: If you're looking for something more advanced, then contributing a newly published architecture to 🤗 Transformers is a great way to dive into the nuts and bolts of the library. There is a detailed guide to help you get started in the 🤗 Transformers documentation.

Blog about what you've learned
: Teaching others what you've learned is a powerful test of your own knowledge, and in a sense this was one of the driving motivations behind us writing this book! There are great tools to help you get started with technical blogging; we recommend *fastpages* as you can easily use Jupyter notebooks for everything.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

About the Authors

Lewis Tunstall is a machine learning engineer at Hugging Face. He has built machine learning applications for startups and enterprises in the domains of NLP, topological data analysis, and time series. Lewis has a PhD in theoretical physics and has held research positions in Australia, the USA, and Switzerland. His current work focuses on developing tools for the NLP community and teaching people to use them effectively.

Leandro von Werra is a machine learning engineer in the open source team at Hugging Face. He has several years of industry experience bringing NLP projects to production by working across the whole machine learning stack, and is the creator of a popular Python library called TRL, which combines transformers with reinforcement learning.

Thomas Wolf is chief science officer at and cofounder of Hugging Face. His team is on a mission to catalyze and democratize NLP research. Prior to cofounding Hugging Face, Thomas earned a PhD in physics and later a law degree. He has worked as a physics researcher and a European patent attorney.

Colophon

The bird on the cover of *Natural Language Processing with Transformers* is a coconut lorikeet (*Trichoglossus haematodus*), a relative of parakeets and parrots. It is also known as the green-naped lorikeet and is native to Oceania.

The plumage of coconut lorikeets blends into their colorful tropical and subtropical surroundings; their green nape meets a yellow collar beneath a deep dark blue head, which ends in an orange-red bill. Their eyes are orange and the breast feathers are red. Coconut lorikeets have one of the longest, pointed tails of the seven species of lorikeet, which is green from above and yellow underneath. These birds measure 10 to 12 inches long and weigh 3.8 to 4.8 ounces.

Coconut lorikeets have one monogamous partner and lay two matte white eggs at a time. They build nests over 80 feet high in eucalyptus trees and live 15 to 20 years in the wild. This species suffers from habitat loss and capture for the pet trade. Many of the animals on O'Reilly's covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *English Cyclopedia*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.