11
Fine-Tuning Large Language Models

Up to this point, we've explored the features and applications of **large language models** (**LLMs**) in their "base" form, meaning that we consumed them with the parameters obtained from their base training. We experimented with many scenarios in which, even in their base form, LLMs have been able to adapt to a variety of scenarios. Nevertheless, there might be extremely domain-specific cases where a general-purpose LLM is not sufficient to fully embrace the taxonomy and knowledge of that domain. If this is the case, you might want to fine-tune your model on your domain-specific data.

Definition

In the context of fine-tuning language models, "taxonomy" refers to a structured classification or categorization system that organizes concepts, terms, and entities according to their relationships and hierarchies within a specific domain. This system is essential for making the model's understanding and generation of content more relevant and accurate for specialized applications.

A concrete example of taxonomy in a domain-specific sector is in the medical field. Here, taxonomy could categorize information into structured groups like diseases, symptoms, treatments, and patient demographics. For instance, in the "diseases" category, there might be subcategories for types of diseases like "cardiovascular diseases," which could be further divided into more specific conditions such as "hypertension" and "coronary artery disease." This detailed categorization helps in fine-tuning language models to understand and generate more precise and contextually appropriate responses in medical consultations or documentation.

In this chapter, we are going to cover the technical details of fine-tuning LLMs, from the theory behind it to the hands-on implementation with Python and Hugging Face. By the end of this chapter, you will be able to fine-tune an LLM on your own data, so that you can build domain-specific applications powered by those models.

We will delve into the following topics:

- Introduction to fine-tuning
- Understanding when you need fine-tuning
- Preparing your data to fine-tune the model
- Fine-tuning a base model on your data
- Hosting strategies for your fine-tuned model

Technical requirements

To complete the tasks in this chapter, you will need the following:

- A Hugging Face account and user access token.
- Python 3.7.1 or later version.
- Python packages: Make sure to have the following Python packages installed: `python-dotenv`, `huggingface_hub`, `accelerate>=0.16.0`, `<1 transformers[torch]`, `safetensors`, `tensorflow`, `datasets`, `evaluate`, and `accelerate`. Those can be easily installed via `pip install` in your terminal. If you want to install everything from the latest release, you can refer to the original GitHub by running `pip install git+https://github.com/huggingface/transformers.git` in your terminal.

You can find all the code and examples in the book's GitHub repository at https://github.com/PacktPublishing/Building-LLM-Powered-Applications.

What is fine-tuning?

Fine-tuning is a technique of **transfer learning** in which the weights of a pretrained neural network are used as the initial values for training a new neural network on a different task. This can improve the performance of the new network by leveraging the knowledge learned from the previous task, especially when the new task has limited data.

Definition

Transfer learning is a technique in machine learning that involves using the knowledge learned from one task to improve the performance on a related but different task. For example, if you have a model that can recognize cars, you can use some of its features to help you recognize trucks. Transfer learning can save you time and resources by reusing existing models instead of training new ones from scratch.

To better understand the concepts of transfer learning and fine-tuning, let's consider the following example.

Imagine you want to train a computer vision neural network to recognize different types of flowers, such as roses, sunflowers, and tulips. You have a lot of photos of flowers, but not enough to train a model from scratch.

Instead, you can use transfer learning, which means taking a model that was already trained on a different task and using some of its knowledge for your new task. For example, you can take a model that was trained to recognize many vehicles, such as cars, trucks, and bicycles. This model has learned how to extract features from images, such as edges, shapes, colors, and textures. These features are useful for any image recognition task, not just the original one.

You can use this model as a base for your flower recognition model. You only need to add a new layer on top of it, which will learn how to classify the features into flower types. This layer is called the classifier layer, and it is needed for the model to adapt to the new task. Training the classifier layer on top of the base model is a process called **feature extraction**. Once this step is done, you can further tailor your model with fine-tuning by unfreezing some of the base model layers and training them together with the classifier layer. This allows you to adjust the base model features to better suit your task.

The following picture illustrates the computer vision model example:

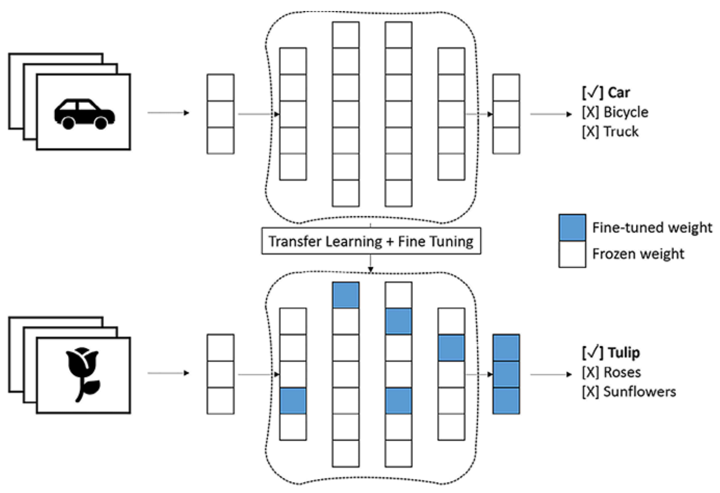


Figure 11.1: Example of transfer learning and fine-tuning

Fine-tuning is usually done after feature extraction, as a final step to improve the performance of the model. You can decide how many layers to unfreeze based on your data size and complexity. A common practice is to unfreeze the last few layers of the base model, which are more specific to the original task, and leave the first few layers frozen, which are more generic and reusable.

To summarize, transfer learning and fine-tuning are techniques that allow you to use a pretrained model for a new task. Transfer learning involves adding a new classifier layer on top of the base model and training only that layer. Fine-tuning involves unfreezing some or all of the base model layers and training them together with the classifier layer.

In the context of generative AI, fine-tuning is the process of adapting a pretrained language model to a specific task or domain by updating its parameters on a task-specific dataset. Fine-tuning can improve the performance and accuracy of the model for the target task. The steps involved in fine-tuning are:

1. **Load the pretrained language model and its tokenizer**: The tokenizer is used to convert text into numerical tokens that the model can process. Different models have unique architectures and requirements, often coming with their own specialized tokenizers designed to handle their specific input formats.

For instance, **BERT** (which stands for **Bidirectional Encoder Representations from Transformers**) uses WordPiece tokenization, while GPT-2 employs **byte-pair encoding** (**BPE**). Models also impose token limits due to memory constraints during training and inference.

These limits determine the maximum sequence length that a model can handle. For example, BERT has a maximum token limit of 512 tokens, while the GPT-2 can handle longer sequences (e.g., up to 1,024 tokens).

2. **Prepare the task-specific dataset**: The dataset should contain input-output pairs that are relevant to the task. For example, for sentiment analysis, the input could be a text review and the output could be a sentiment label (positive, negative, or neutral).
3. **Define the task-specific head**: The head is a layer or a set of layers that are added on top of the pretrained model to perform the task. The head should match the output format and size of the task. For example, for sentiment analysis, the head could be a linear layer with three output units corresponding to the three sentiment labels.

Note

When dealing with an LLM specifically designed for text generation, the architecture differs from models used for classification or other tasks. In fact, unlike classification tasks, where we predict labels, an LLM predicts the next word or token in a sequence. This layer is added on top of the pretrained transformer-based models with the purpose of transforming the contextualized hidden representations from the base model into probabilities over the vocabulary.

4. **Train the model on the task-specific dataset**: The training process involves feeding the input tokens to the model, computing the loss between the model output and the true output, and updating the model parameters using an optimizer. The training can be done for a fixed number of epochs or until a certain criterion is met.
5. **Evaluate the model on a test or validation set**: The evaluation process involves measuring the performance of the model on unseen data using appropriate metrics. For example, for sentiment analysis, the metric could be accuracy or F1-score (which will be discussed later in this chapter). The evaluation results can be used to compare different models or fine-tuning strategies.

Even though it is less computationally and time expensive than full training, fine-tuning an LLM is not a “light” activity. As LLMs are, by definition, large, their fine-tuning has hardware requirements as well as data collection and preprocessing.

So, the first question that you want to ask yourself while approaching a given scenario is: “Do I really need to finetune my LLM?”

When is fine-tuning necessary?

As we saw in previous chapters, good prompt engineering combined with the non-parametric knowledge you can add to your model via embeddings are exceptional techniques to customize your LLM, and they can account for around 90% of use cases. However, the preceding affirmation tends to hold for the state-of-the-art models, such as GPT-4, Llama 2, and PaLM 2. As discussed, those models have a huge number of parameters that make them heavy, hence the need for computational power; plus, they might be proprietary and subject to a pay-per-use cost.

Henceforth, fine-tuning might also be useful when you want to leverage a light and free-of-charge LLM, such as the Falcon LLM 7B, yet you want it to perform as well as a SOTA model in your specific task.

Some examples of when fine-tuning might be necessary are:

- When you want to use an LLM for sentiment analysis on movie reviews, but the LLM was pretrained on Wikipedia articles and books. Fine-tuning can help the LLM learn the vocabulary, style, and tone of movie reviews, as well as the relevant features for sentiment classification.
- When you want to use an LLM for text summarization on news articles, but the LLM was pretrained on a language modeling objective. Fine-tuning can help the LLM learn the structure, content, and length of summaries, as well as the generation objective and evaluation metrics.
- When you want to use an LLM for machine translation between two languages, but the LLM was pretrained on a multilingual corpus that does not include those languages. Fine-tuning can help the LLM learn the vocabulary, grammar, and syntax of the target languages, as well as the translation objective and alignment methods.
- When you want to use an LLM to perform complex **named entity recognition** (**NER**) tasks. For example, financial and legal documents contain specialized terminology and entities that are not typically prioritized in general language models, henceforth a fine-tuning process might be extremely beneficial here.

In this chapter, we will be covering a full-code approach leveraging Hugging Face models and libraries. However, be aware that Hugging Face also offers a low-code platform called AutoTrain (you can read more about that at `https://huggingface.co/autotrain`), which might be a good alternative if your organization is more oriented towards low-code strategies.

Getting started with fine-tuning

In this section, we are going to cover all the steps needed to fine-tune an LLM with a full-code approach. We will be leveraging Hugging Face libraries, such as `datasets` (to load data from the Hugging Face datasets ecosystem) and `tokenizers` (to provide an implementation of the most popular tokenizers). The scenario we are going to address is a sentiment analysis task. Our goal is to fine-tune a model to make it an expert binary classifier of emotions, clustered into "positive" and "negative."

Obtaining the dataset

The first ingredient that we need is the training dataset. For this purpose, I will leverage the datasets library available in Hugging Face to load a binary classification dataset called IMDB (you can find the dataset card at `https://huggingface.co/datasets/imdb`).

The dataset contains movie reviews, which are classified as positive or negative. More specifically, the dataset contains two columns:

- Text: The raw text movie review.
- Label: The sentiment of that review. It is mapped as "0" for "Negative" and "1" for "Positive."

As it is a **supervised learning** problem, the dataset already comes with 25,000 rows for the training set and 25,000 rows for the validation set.

Definition

Supervised learning is a type of machine learning that uses labeled datasets to train algorithms to classify data or predict outcomes accurately. Labeled datasets are collections of examples that have both input features and desired output values, also known as labels or targets. For example, a labeled dataset for handwriting recognition might have images of handwritten digits as input features and the corresponding numerical values as labels.

Training and validation sets are subsets of the labeled dataset that are used for different purposes in the supervised learning process. The training set is used to fit the parameters of the model, such as the weights of the connections in a neural network. The validation set is used to tune the hyperparameters of the model, such as the number of hidden units in a neural network or the learning rate. Hyperparameters are settings that affect the overall behavior and performance of the model but are not directly learned from the data. The validation set helps to select the best model among different candidates by comparing their accuracy or other metrics on the validation set.

Supervised learning differs from another type of machine learning, which is **unsupervised learning**. With the latter, the algorithm is tasked with finding patterns, structures, or relationships in a dataset without the presence of labeled outputs or targets. In other words, in unsupervised learning, the algorithm is not provided with specific guidance or labels to direct its learning process. Instead, it explores the data and identifies inherent patterns or groupings on its own.

You can download the IMDB dataset by running the following code:

```
from datasets import load_dataset
dataset = load_dataset("imdb")
dataset
```

Hugging Face datasets come with a dictionary schema, which is as follows:

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 25000
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 25000
    })
    unsupervised: Dataset({
        features: ['text', 'label'],
        num_rows: 50000
    })
})
```

To access one observation of a particular Dataset object (for example, `train`), you can use slicers, as follows:

```
dataset["train"][100]
```

This gives us the following output:

```
{'text': "Terrible movie. Nuff Said.[…]
 'label': 0}
```

So, the 101st observation of the training set contains a review labeled as negative.

Now that we have the dataset, we need to preprocess it so that can be used to train our LLM. To do so, we need to tokenize the provided text, and we will discuss this in the next section.

Tokenizing the data

A tokenizer is a component that is responsible for splitting a text into smaller units, such as words or subwords, that can be used as inputs for an LLM. Tokenizers can be used to encode text efficiently and consistently, as well as to add special tokens, such as mask or separator tokens, that are required by some models.

Hugging Face provides a powerful utility called AutoTokenizer, available in the Hugging Face Transformers library, that offers tokenizers for various models, such as BERT and GPT-2. It serves as a generic tokenizer class that dynamically selects and instantiates the appropriate tokenizer based on the pretrained model you specify.

The following code snippet shows how we can initialize our tokenizer:

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
```

Note that we picked a specific tokenizer called `bert-base-cased`. In fact, there is a link between a tokenizer and an LLM, in the sense that the the tokenizer prepares the inputs for the model by converting the text into numerical IDs that the model can understand.

Definition

The input IDs are the numerical IDs that correspond to the tokens in the vocabulary of the tokenizer. They are returned by the tokenizer function when encoding a text input. The input IDs are used as inputs for the model, which expects numerical tensors rather than strings. Different tokenizers may have different input IDs for the same tokens, depending on their vocabulary and tokenization algorithm.

Different models may use different tokenization algorithms, such as word-based, character-based, or subword-based. Therefore, it is important to use the correct tokenizer for each model, otherwise the model may not perform well or even produce errors. Let's look at potential scenarios for each:

- A character-based approach might fit scenarios that deal with rare words or languages with complex morphological structures, or when dealing with spelling correction tasks
- The word-based approach might be a good fit for scenarios like NER, sentiment analysis, and text classification
- The sub-word approach interpolates between the previous two, and it is useful when we want to balance the granularity of text representation with efficiency.

As we will see in the next section, we will leverage the **BERT** model for this scenario, hence we loaded its pretrained tokenizer (which is a word-based tokenizer powered by an algorithm called WordPiece).

We now need to initialize `tokenize_function`, which will be used to format the dataset:

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

As you can see, we also configured the **padding** and **truncation** of `tokenize_function` to ensure an output with the right sizing for our BERT model.

Definition

Padding and truncation are two techniques that are used to make the input sequences of text have the same length. This is often required for some **natural language processing** (**NLP**) models, such as the BERT model, that expect fixed-length inputs.

Padding means adding some special tokens, usually zeros, at the end or the beginning of a sequence to make it reach the desired length. For example, if we have a sequence of length 5 and we want to pad it to a length of 8, we can add 3 zeros at the end, like this: [1, 2, 3, 4, 5, 0, 0, 0]. This is called post-padding. Alternatively, we can add 3 zeros at the beginning, like this: [0, 0, 0, 1, 2, 3, 4, 5]. This is called pre-padding. The choice of padding strategy depends on the model and the task.

Truncation means removing some tokens from a sequence to make it fit the desired length. For example, if we have a sequence of length 10 and we want to truncate it to a length of 8, we can remove 2 tokens from the end or the beginning of the sequence. For example, we can remove the last 2 tokens, like this: [1, 2, 3, 4, 5, 6, 7, 8]. This is called post-truncation. Alternatively, we can remove the first 2 tokens, like this: [3, 4, 5, 6, 7, 8, 9, 10]. This is called pre-truncation. The choice of truncation strategy also depends on the model and the task.

Now, we can apply the function to our dataset and inspect the numerical IDs of one entry:

```
tokenized_datasets = dataset.map(tokenize_function, batched=True)
tokenized_datasets['train'][100]['input_ids']
```

Here is our output:

```
[101,
 12008,
 27788,
...
 0,
 0,
 0,
 0,
 0]
```

As you can see, the last elements of the vector are zeroes, due to the `padding='max_lenght'` parameter passed to the function.

Optionally, you can decide to reduce the size of your dataset if you want to make the training time shorter. In my case, I've shrunk the dataset as follows:

```
small_train_dataset = tokenized_datasets["train"].shuffle(seed=42).select(range(500))
small_eval_dataset = tokenized_datasets["test"].shuffle(seed=42).select(range(500))
```

So, I have two sets – one for training, one for testing – of 500 observations each. Now that we have our dataset preprocessed and ready, we need the model to be fine-tuned.

Fine-tuning the model

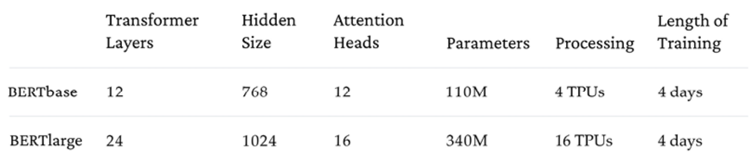
As anticipated in the previous section, the LLM we are going to leverage for fine-tuning is the base version of BERT. The BERT model is a transformer-based, encoder-only model for natural language understanding introduced by Google researchers in 2018. BERT was the first example of a general-purpose LLM, meaning that it was the first model to be able to tackle multiple NLP tasks at once, which was different from the task-specific models existing up to that moment.

Now, even though it might sound a bit "old fashioned" (in fact, compared to today's model like the GPT-4, it is not even "large," with only 340 million parameters in its large version), given all the new LLMs that have emerged in the market in the last few months, BERT and its fine-tuned variants are still a widely adopted architecture. In fact, it was thanks to BERT that the standard for language models has greatly improved.

The BERT model has two main components:

- Encoder: The encoder consists of multiple layers of transformer blocks, each with a self-attention layer and a feedforward layer. The encoder takes as input a sequence of tokens, which are the basic units of text, and outputs a sequence of hidden states, which are high-dimensional vectors that represent the semantic information of each token.

- Output layer: The output layer is task-specific and can be different depending on the type of task that BERT is used for. For example, for text classification, the output layer can be a linear layer that predicts the class label of the input text. For question answering, the output layer can be two linear layers that predict the start and end positions of the answer span in the input text.
- The number of layers and parameters of the model depends on the model version. In fact, BERT comes in two sizes: BERTbase and BERTlarge. The following illustration shows the difference between the two versions:



	Transformer Layers	Hidden Size	Attention Heads	Parameters	Processing	Length of Training
BERTbase	12	768	12	110M	4 TPUs	4 days
BERTlarge	24	1024	16	340M	16 TPUs	4 days

Figure 11.2: A comparison between BERTbase and BERTlarge (source: https://huggingface.co/blog/bert-101)

Later, other versions such as BERT-tiny, BERT-mini, BERT-small, and BERT-medium were introduced to reduce the computational cost and memory usage of BERT.

The model has been trained on a heterogeneous corpus of around 3.3 billion words, belonging to Wikipedia and Google's BooksCorpus. The training phase involved two objectives:

- **Masked language modeling** (**MLM**): MLM aims to teach the model to predict the original words that are randomly masked (replaced with a special token) in the input text. For example, given the sentence "He bought a new [MASK] yesterday," the model should predict the word "car" or "bike" or something else that makes sense. This objective helps the model learn the vocabulary and the syntax of the language, as well as the semantic and contextual relations between words.
- **Next sentence prediction** (**NSP**): NSP aims to teach the model to predict whether two sentences are consecutive or not in the original text. For example, given the sentences "She loves reading books" and "Her favorite genre is fantasy," the model should predict that they are consecutive because they are likely to appear together in a text. However, given the sentences "She loves reading books" and "He plays soccer every weekend," the model should predict that they are not consecutive because they are unlikely to be related. This objective helps the model learn the coherence and logic of the text, as well as the discourse and pragmatic relations between sentences.

By using these two objectives (on which the model is trained at the same time), the BERT model can learn general language knowledge that can be transferred to specific tasks, such as text classification, question answering, and NER. The BERT model can achieve better performance on these tasks than previous models that only use one direction of context or do not use pre-training at all. In fact, it has achieved state-of-the-art results on many benchmarks and tasks, such as **General Language Understanding Evaluation** (**GLUE**), **Stanford Question Answering**

Dataset (**SQuAD**), and **Multi-Genre Natural Language Inference** (**MultiNLI**).

The BERT model is available – along with many fine-tuned versions –in the Hugging Face Hub. You can instantiate the model as follows:

```
import torch
from transformers import AutoModelForSequenceClassification
model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", num_labels=2
```

Note that `AutoModelForSequenceClassification` is a subclass of `AutoModel`, which can instantiate a model architecture suitable for sequence classification, such as text classification or sentiment analysis. It can be used for any task that requires a single label or a list of labels for each input sequence. In my case, I set the number of output labels equal to two since we are dealing with a binary classification problem.

On the other hand, `AutoModel` is a generic class that can instantiate any model architecture from the library based on the pretrained model name or path. It can be used for any task that does not require a specific output format, such as feature extraction or language modeling.

The final step before starting the training is to define the evaluation metrics we will need to understand how well our model will perform once fine-tuned.

Using evaluation metrics

As we saw in *Chapter 1*, evaluating an LLM in its general-purpose application might be cumbersome. As those models are trained on unlabeled text and are not task-specific, but rather generic and adaptable given a user's prompt, traditional evaluation metrics were not suitable anymore. Evaluating an LLM means, among other things, measuring its language fluency, its coherence, and its ability to emulate different styles depending on a user's request.

However, we also saw how an LLM can be used for very specific scenarios, as in our binary classification task. If this is the case, evaluation metrics boil down to those commonly used for that scenario.

Note

When it comes to more conversational tasks like summarization, Q&A, and retrieval-augmented generation, a new set of evaluation metrics needs to be introduced, often powered in turn by LLMs. Some of the most popular metrics are the following:

- Fluency: This assesses how naturally and smoothly the generated text reads.
- Coherence: This evaluates the logical flow and connectivity of ideas within a text.
- Relevance: This measures how well the generated content aligns with the given prompt or context.
- GPT-similarity: This quantifies how closely the generated text resembles human-written content.

> > - Groundedness: This assesses whether the generated text is based on factual information or context.
> >
> > These evaluation metrics help us understand the quality, naturalness, and relevance of LLM-generated text, guiding improvements and ensuring reliable AI assistance.

When it comes to binary classification, one of the most basic ways to evaluate a binary classifier is to use a confusion matrix. A confusion matrix is a table that shows how many of the predicted labels match the true labels. It has four cells:

- **True positive (TP)**: The number of cases where the classifier correctly predicted 1 when the true label was 1.
- **False positive (FP)**: The number of cases where the classifier incorrectly predicted 1 when the true label was 0.
- **True negative (TN)**: The number of cases where the classifier correctly predicted 0 when the true label was 0.
- **False negative (FN)**: The number of cases where the classifier incorrectly predicted 0 when the true label was 1.

Here is an example of a confusion matrix for the sentiment classifier we are going to build, knowing that the label 0 is associated with "Negative" and the label 1 with "Positive":

	Predicted Positive	**Predicted Negative**
Positive	20 (TP)	5 (FN)
Negative	3 (FP)	72 (TN)

The confusion matrix can be used to calculate various metrics that measure different aspects of the classifier's performance. Some of the most common metrics are:

- **Accuracy**: The proportion of correct predictions among all predictions. It is calculated as `(TP + TN) / (TP + FP + TN + FN)`. For example, the accuracy of the sentiment classifier is `(20 + 72) / (20 + 3 + 72 + 5) = 0.92`.
- **Precision**: The proportion of correct positive predictions among all positive predictions. It is calculated as `TP / (TP + FP)`. For example, the precision of the sentiment classifier is `20 / (20 + 3) = 0.87`.
- **Recall**: The proportion of correct positive predictions among all positive cases. It is also known as sensitivity or true positive rate. It is calculated as `TP / (TP + FN)`. For example, the recall of the sentiment classifier is `20 / (20 + 5) = 0.8`.
- **Specificity**: The proportion of correct negative predictions among all negative cases. It is also known as the true negative rate. It is calculated as `TN / (TN + FP)`. For example, the specificity of the sentiment classifier is `72 / (72 + 3) = 0.96`.
- **F1-score**: The harmonic mean of precision and recall. It is a measure of balance between precision and recall. It is calculated as `2 * (precision * recall) / (precision + recall)`. For example, the F1-score of the sentiment classifier is `2 * (0.87 * 0.8) / (0.87 + 0.8) = 0.83`.

There are many other metrics that can be derived from the confusion matrix or other sources, such as the decision score or the probability output of the classifier. Some examples are:

- **Receiver operating characteristic (ROC) curve**: A plot of recall versus false positive rate (`FP / (FP + TN)`), which shows how well the classifier can distinguish between positive and negative cases at different thresholds.
- **Area under the ROC curve (AUC)**: The AUC, which measures how well the classifier can rank positive cases higher than negative cases. It can be illustrated in the following diagram, where the ROC curve and the area under the curve are displayed:

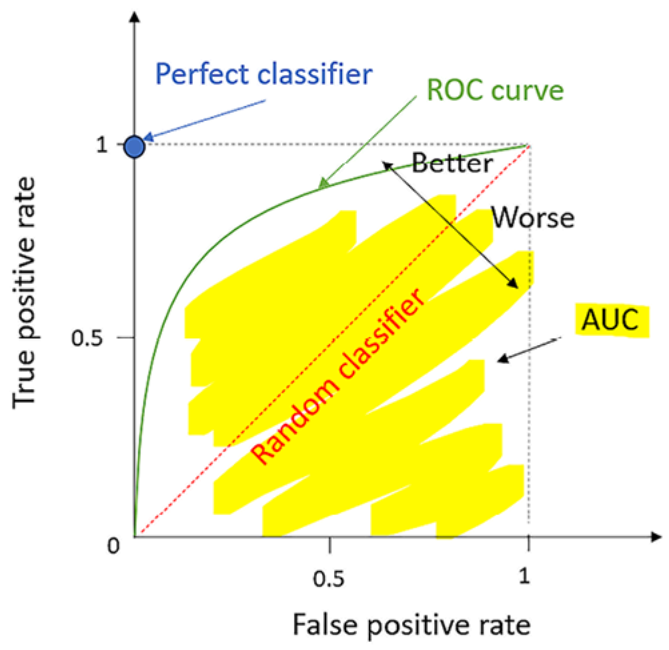


Figure 11.3: Illustration of a ROC curve, hightlighting a perfect classifier and the Area Under the Curve (AUC)

In our case, we will simply use the accuracy metric by following these steps:

1. You can import this metric from the `evaluate` library as follows:

```
import numpy as np
import evaluate
metric = evaluate.load("accuracy")
```

2. We also need to define a function that computes the accuracy given the output of the training phase:

```
def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    return metric.compute(predictions=predictions, references=labels)
```

3. Finally, we need to set our evaluation strategy, which means how often we want our model to be tested against the test set while training:

```
from transformers import TrainingArguments, Trainer
training_args = TrainingArguments(output_dir="test_trainer", num_train_epochs = 2
evaluation_strategy="epoch")
```

In our case, we will set `epoch` as the evaluation strategy, meaning that the evaluation is done at the end of each epoch.

Definition

An epoch is a term used in machine learning to describe one complete pass through the entire training dataset. It is a hyperparameter that can be tuned to improve the performance of a machine-learning model. During an epoch, the model's weights are updated based on the training data and the loss function. An epoch can consist of one or more batches, which are smaller subsets of the training data. The number of batches in an epoch depends on the batch size, which is another hyperparameter that can be adjusted.

Now we have all the ingredients needed to start our fine-tuning, which will be covered in the next section.

Training and saving

The last component we need to fine-tune our model is a `Trainer` object. The `Trainer` object is a class that provides an API for feature-complete training and evaluation of models in PyTorch, optimized for Hugging Face Transformers. You can follow these steps:

1. Let's first initialize our `Trainer` by specifying the parameters we've already configured in the previous steps. More specifically, the `Trainer` will need a model, some configuration args (such as the number of epochs), a training dataset, an evaluation dataset, and the type of evaluation metric to compute:

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=small_train_dataset,
    eval_dataset=small_eval_dataset,
    compute_metrics=compute_metrics,
)
```

2. You can then initiate the process of fine-tuning by calling the `trainer` as follows:

```
trainer.train()
```

Depending on your hardware, the training process might take some time. In my case, given the reduced size of the dataset and the low number of epochs (only 2), I don't expect exceptional results. Nevertheless, the training results for only two epochs in terms of accuracy are the following:

```
{'eval_loss': 0.6720085144042969, 'eval_accuracy': 0.58, 'eval_runtime': 609.7916, 'eval_s
```

```
{'eval_loss': 0.5366445183753967, 'eval_accuracy': 0.82, 'eval_runtime': 524.186, 'eval_sa
```

As you can see, between the two epochs the model gained an accuracy improvement of 41.38%, hitting a final accuracy of 82%. Considering the aforementioned elements, that's not bad!

3. Once the model is trained, we can save it locally, specifying the path as follows:

   ```
   trainer.save_model('models/sentiment-classifier')
   ```

4. To consume and test the model, you can load it with the following code:

   ```
   model = AutoModelForSequenceClassification.from_pretrained('models/sentiment-classifier
   ```

5. Finally, we need to test our model. To do so, let's pass a sentence to the model (to be first tokenized) on which it can perform sentiment classification:

   ```
   inputs = tokenizer("I cannot stand it anymore!", return_tensors="pt")
   outputs = model(**inputs)
   outputs
   ```

This yields the following output:

```
SequenceClassifierOutput(loss=None, logits=tensor([[ 0.6467, -0.0041]], grad_fn=<AddmmBack
```

Note that the model output is a `SequenceClassifierOutput` object, which is the base class for outputs of sentence classification models. Within this object, we are interested in the logit **tensor**, which is the vector of raw (non-normalized) predictions associated with labels that our classification model generated.

6. Since we are working with tensors, we will need to leverage the `tensorflow` library in Python. Plus, we will use the `softmax` function to obtain the probability vector associated with each label, so that we know that the final result corresponds to the label with the greatest probability:

   ```
   import tensorflow as tf
   predictions = tf.math.softmax(outputs.logits.detach(), axis=-1)
   print(predictions)
   ```

The following is the obtained output:

```
tf.Tensor([[0.6571879  0.34281212]], shape=(1, 2), dtype=float32)
```

Our model tells us that the sentiment of the sentence "I can't stand it anymore" is negative, with a probability of 65.71%.

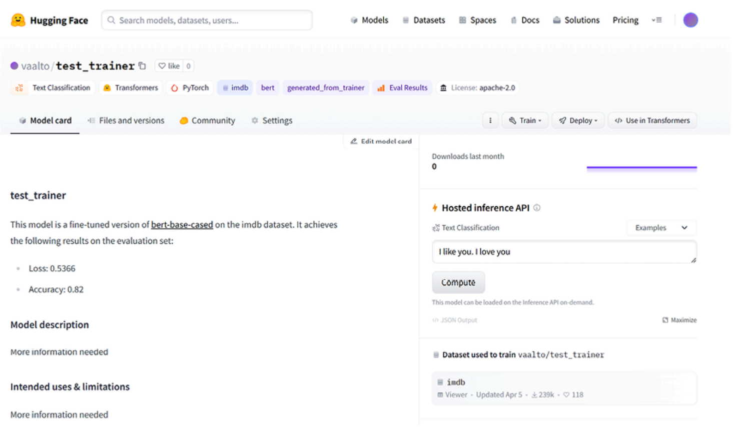
7. Note that you can also save the model in your Hugging Face account. To do so, you first need to allow the notebook to push the code to your account as follows:

```
from huggingface_hub import notebook_login
notebook_login()
```

8. You will be prompted to the Hugging Face login page, where you have to input your access token. Then, you can save the model, specifying your account name and model name:

```
trainer.push_to_hub('vaalto/sentiment-classifier')
```

By doing so, this model can be consumed via the Hugging Face Hub as easily as we saw in the previous chapter, as shown in the following screenshot:



test_trainer

This model is a fine-tuned version of bert-base-cased on the imdb dataset. It achieves the following results on the evaluation set:

- Loss: 0.5366
- Accuracy: 0.82

Model description

More information needed

Intended uses & limitations

More information needed

I like you. I love you

This model can be loaded on the Inference API on-demand.

Dataset used to train vaalto/test_trainer

imdb
Viewer · Updated Apr 5 · 239k · 118

Figure 11.4: Model card within the Hugging Face Hub space

Furthermore, you can also decide to make the model public, so that everyone within Hugging Face can test and consume your creation.

In this section, we fine-tuned a BERT model with just a few lines of code, thanks to Hugging Face libraries and accelerators. Again, if your goal is reducing the code amount, you can leverage the low-code AutoTrain platform hosted in Hugging Face to train and fine-tune models.

Hugging Face is definitely a solid platform for training your open-source LLM. In addition to that, there are further platforms you might want to leverage since proprietary models can also be fine-tuned. For example, OpenAI lets you fine-tune the GPT series with your own data, providing the computational power to train and host your customized models.

Overall, fine-tuning can be the icing on the cake that makes your LLM exceptional for your use case. Deciding a strategy to do so based on the framework we explored at the beginning is a pivotal step in building a successful application.

Summary

In this chapter, we covered the process of fine-tuning LLMs. We started with a definition of fine-tuning and general considerations to take into account if you have to decide to fine-tune your LLM.

We then went hands-on with practical sections on fine-tuning. We covered a scenario where, starting from a base BERT model, we wanted a powerful review sentiment analyzer. To do so, we fine-tuned the base model on the IMDB dataset using a full-code approach with Hugging Face Python libraries.

Fine-tuning is a powerful technique to further customize LLMs toward your goal. However, along with many other aspects of LLMs, it comes with some concerns and considerations in terms of ethics and security. In the next chapter, we are going to delve deeper into that, sharing how to establish guardrails with LLMs and, more generally, how governments and countries are approaching the problem from a regulatory perspective.

References

- Training dataset: **https://huggingface.co/datasets/imdb**
- HF AutoTrain: **https://huggingface.co/docs/autotrain/index**
- BERT paper: *Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova*, 2019, *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*: **https://arxiv.org/abs/1810.04805**