Chapter 11. Fine-Tuning Representation Models for Classification

In Chapter 4, we used pretrained models to classify our text. We kept the pretrained models as they were without any modifications to them. This might make you wonder, what happens if we were to fine-tune them?

If we have sufficient data, fine-tuning tends to lead to some of the best-performing models possible. In this chapter, we will go through several methods and applications for fine-tuning BERT models. "Supervised Classification" demonstrates the general process of fine-tuning a classification model. Then, in "Few-Shot Classification", we look at SetFit, which is a method for efficiently fine-tuning a high-performing model using a small number of training examples. In "Continued Pretraining with Masked Language Modeling", we will explore how to continue training a pretrained model. Lastly, classification on a token level is explored in "Named-Entity Recognition".

We will focus on nongenerative tasks, as generative models will be covered in Chapter 12.

Supervised Classification

In Chapter 4, we explored supervised classification tasks by leveraging pretrained representation models that were either trained to predict sentiment (task-specific model) or to generate embeddings (embedding model), as shown in Figure 11-1.

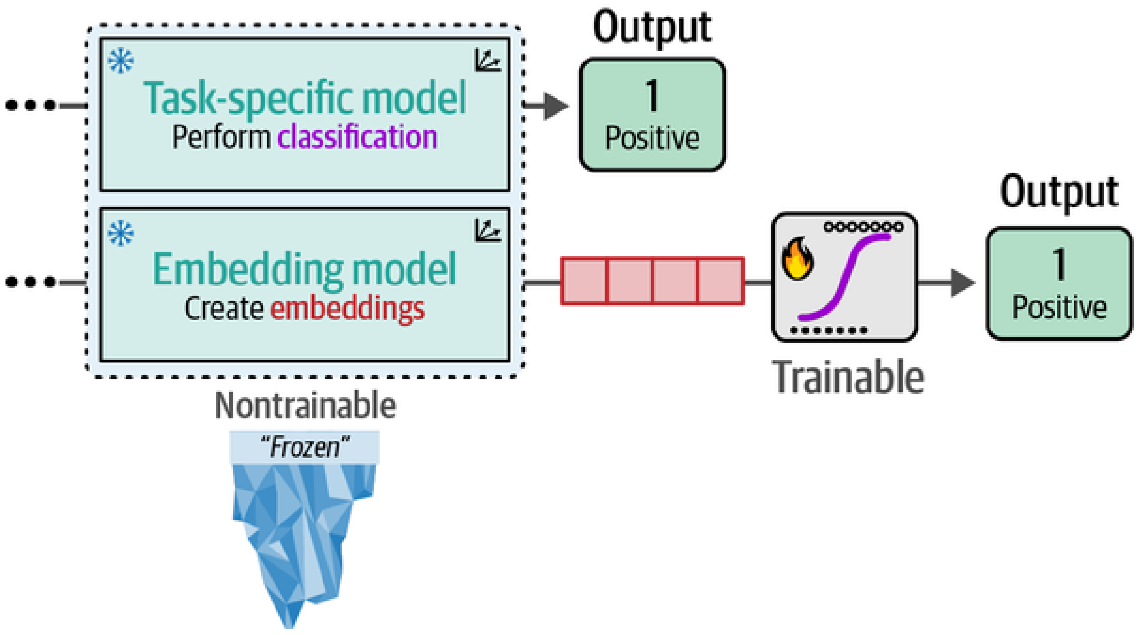


Figure 11-1. In Chapter 4, we used pretrained models to perform classification without updating their weight. These models were kept "frozen."

Both models were kept frozen (nontrainable) to showcase the potential of leveraging pretrained models for classification tasks. The embedding model uses a separate trainable classification head (classifier) to predict the sentiment of movie reviews.

In this section, we will take a similar approach but allow both the model and the classification head to be updated during training. As shown in Figure 11-2, instead of using an embedding model, we will fine-tune a pretrained BERT model to create a task-specific model similar to the one we used in Chapter 2. Compared to the embedding model approach, we will fine-tune both the representation model and the classification head as a single architecture.

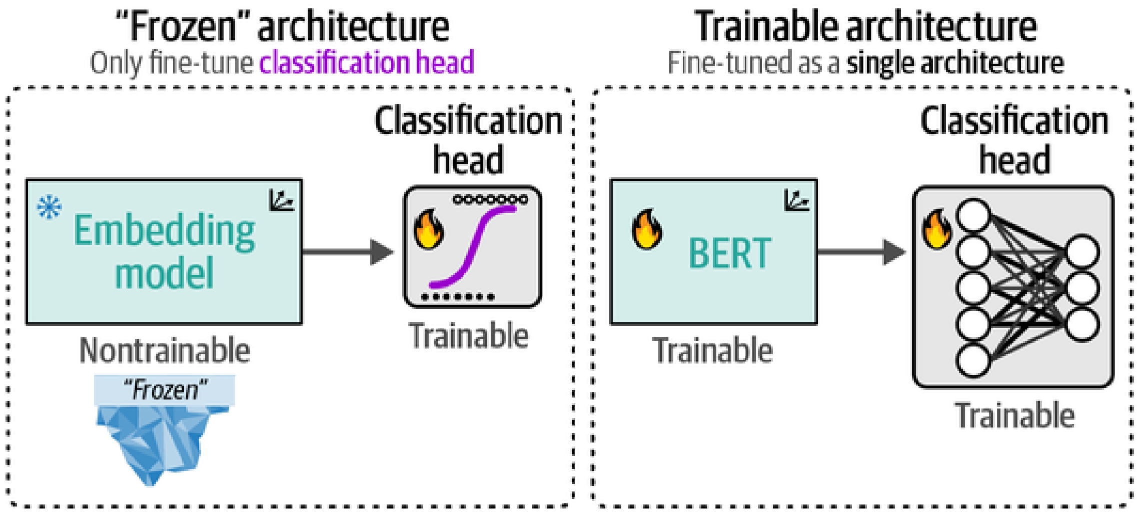


Figure 11-2. Compared to the "frozen" architecture, we instead train both the pretrained BERT model and the classification head. A backward pass will start at the classification head and go through BERT.

To do so, instead of freezing the model, we allow it to be trainable and update its parameters during training. As illustrated in Figure 11-3, we will use a pretrained BERT model and add a neural network as a classification head, both of which will be fine-tuned for classification.

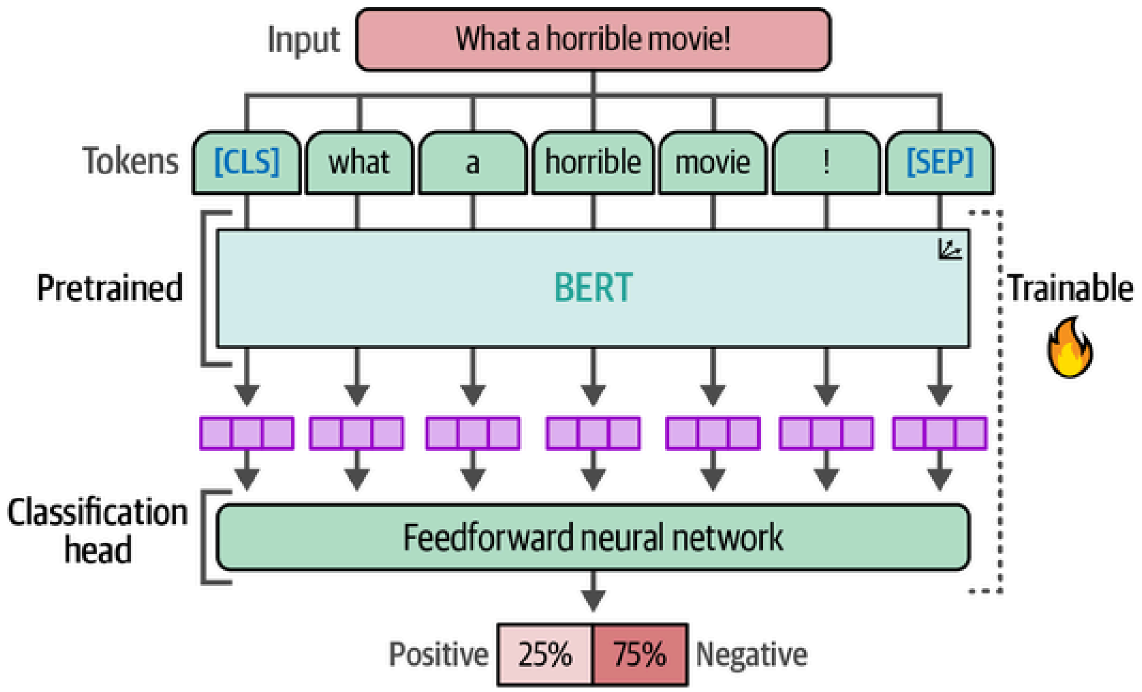


Figure 11-3. The architecture of a task-specific model. It contains a pretrained representation model (e.g., BERT) with an additional classification head for the specific task.

In practice, this means that the pretrained BERT model and the classification head are updated jointly. Instead of independent processes, they learn from one another and allow for more accurate representations.

Fine-Tuning a Pretrained BERT Model

We will be using the same dataset we used in Chapter 4 to fine-tune our model, namely the Rotten Tomatoes dataset, which contains 5,331 positive and 5,331 negative movie reviews from Rotten Tomatoes:

```
from datasets import load_dataset

# Prepare data and splits
tomatoes = load_dataset("rotten_tomatoes")
train_data, test_data = tomatoes["train"], tomatoes["test"]
```

The first step in our classification task is to select the underlying model we want to use. We use `"bert-base-cased"`, which was pretrained on the English Wikipedia as well as a large dataset consisting of unpublished books.[1]

We define the number of labels that we want to predict beforehand. This is necessary to create the feedforward neural network that is applied on top of our pretrained model:

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification

# Load model and tokenizer
model_id = "bert-base-cased"
model = AutoModelForSequenceClassification.from_pretrained(
    model_id, num_labels=2
)
tokenizer = AutoTokenizer.from_pretrained(model_id)
```

Next, we will tokenize our data:

```
from transformers import DataCollatorWithPadding

# Pad to the longest sequence in the batch
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)

def preprocess_function(examples):
    """Tokenize input data"""
    return tokenizer(examples["text"], truncation=True)

# Tokenize train/test data
tokenized_train = train_data.map(preprocess_function, batched=True)
tokenized_test = test_data.map(preprocess_function, batched=True)
```

Before creating the `Trainer`, we will want to prepare a special `DataCollator`. A `DataCollator` is a class that helps us build batches of data but also allows us to apply data augmentation.

During this process of tokenization, and as shown in Chapter 9, we will add padding to the input text to create equally sized representations. We use `DataCollatorWithPadding` for that.

Of course, an example would not be complete without defining some metrics:

```
import numpy as np
from datasets import load_metric

def compute_metrics(eval_pred):
    """Calculate F1 score"""
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)

    load_f1 = load_metric("f1")
    f1 = load_f1.compute(predictions=predictions, references=labels)["f1"]
    return {"f1": f1}
```

With `compute_metrics` we can define any number of metrics that we are interested in and that can be printed out or logged during training. This is especially helpful during training as it allows for detecting overfitting behavior.

Next, we instantiate our `Trainer`:

```
from transformers import TrainingArguments, Trainer

```

```
# Training arguments for parameter tuning
training_args = TrainingArguments(
   "model",
   learning_rate=2e-5,
   per_device_train_batch_size=16,
   per_device_eval_batch_size=16,
   num_train_epochs=1,
   weight_decay=0.01,
   save_strategy="epoch",
   report_to="none"
)

# Trainer which executes the training process
trainer = Trainer(
   model=model,
   args=training_args,
   train_dataset=tokenized_train,
   eval_dataset=tokenized_test,
   tokenizer=tokenizer,
   data_collator=data_collator,
   compute_metrics=compute_metrics,
)
```

The `TrainingArguments` class defines hyperparameters we want to tune, such as the learning rate and how many epochs (rounds) we want to train. The `Trainer` is used to execute the training process.

Finally, we can train our model and evaluate it:

```
trainer.evaluate()
```

```


{'eval_loss': 0.3663691282272339,
 'eval_f1': 0.8492366412213741,
 'eval_runtime': 4.5792,
 'eval_samples_per_second': 232.791,
 'eval_steps_per_second': 14.631,
 'epoch': 1.0}
```

We get an F1 score of 0.85, which is quite a bit higher than the task-specific model we used in Chapter 4, which resulted in an F1 score of 0.80. It shows that fine-tuning a model yourself can be more advantageous than using a pretrained model. It only costs us a couple of minutes to train.

Freezing Layers

To further showcase the importance of training the entire network, the next example will demonstrate how you can use Hugging Face Transformers to freeze certain layers of your network.

We will freeze the main BERT model and allow only updates to pass through the classification head. This will be a great comparison as we will keep everything the same, except for freezing specific layers.

To start, let's reinitialize our model so we can start from scratch:

```
# Load model and tokenizer
model = AutoModelForSequenceClassification.from_pretrained(
    model_id, num_labels=2
)
tokenizer = AutoTokenizer.from_pretrained(model_id)
```

Our pretrained BERT model contains a lot of layers that we can potentially freeze. Inspecting these layers gives insight into the structure of the network and what we might want to freeze:

```
# Print layer names
for name, param in model.named_parameters():
    print(name)
```

```
bert.embeddings.word_embeddings.weight
bert.embeddings.position_embeddings.weight
bert.embeddings.token_type_embeddings.weight
bert.embeddings.LayerNorm.weight
bert.embeddings.LayerNorm.bias
bert.encoder.layer.0.attention.self.query.weight
bert.encoder.layer.0.attention.self.query.bias
...
bert.encoder.layer.11.output.LayerNorm.weight
bert.encoder.layer.11.output.LayerNorm.bias
bert.pooler.dense.weight
bert.pooler.dense.bias
classifier.weight
classifier.bias
```

There are 12 (0–11) encoder blocks consisting of attention heads, dense networks, and layer normalization. We further illustrate this architecture in Figure 11-4 to demonstrate everything that could be potentially frozen. On top of that, we have our classification head.

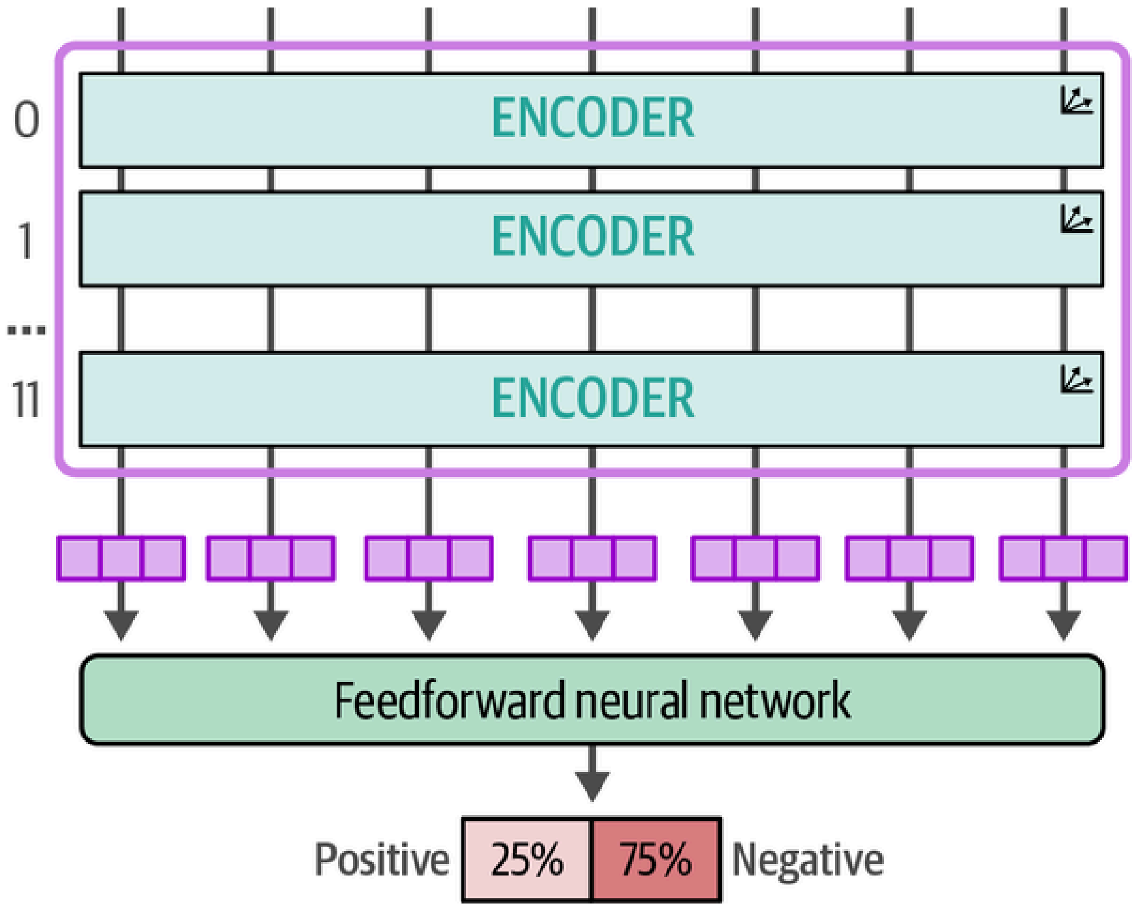


Figure 11-4. The basic architecture of BERT with the additional classification head.

We could choose to only freeze certain layers to speed up computing but still allow the main model to learn from the classification task. Generally, we want frozen layers to be followed by trainable layers.

We are going to freeze everything except for the classification head as we did in Chapter 2:

```
for name, param in model.named_parameters():

     # Trainable classification head
     if name.startswith("classifier"):
        param.requires_grad = True

      # Freeze everything else
     else:
        param.requires_grad = False
```

As shown in Figure 11-5, we have frozen everything except for the feed-forward neural network, which is our classification head.

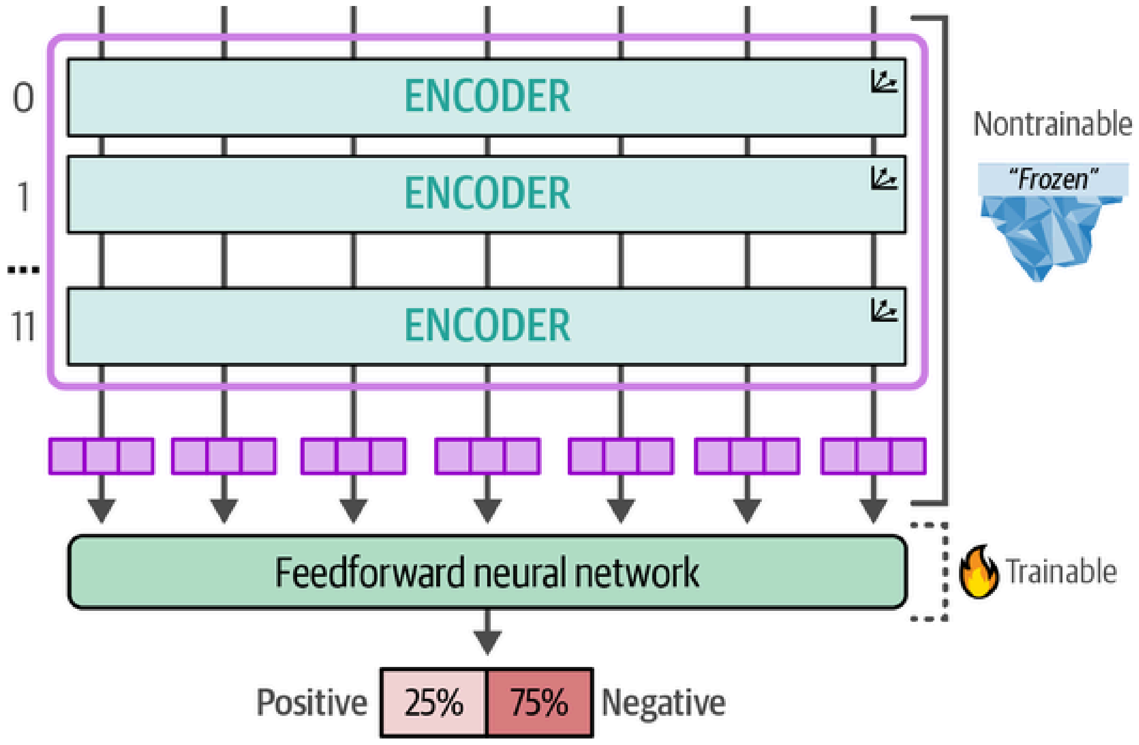


Figure 11-5. We fully freeze all encoder blocks and embedding layers such that the BERT model does not learn new representations during fine-tuning.

Now that we have successfully frozen everything but the classification head, we can move on to train our model:

```
from transformers import TrainingArguments, Trainer

# Trainer which executes the training process
trainer = Trainer(
   model=model,
   args=training_args,
   train_dataset=tokenized_train,
   eval_dataset=tokenized_test,
   tokenizer=tokenizer,
   data_collator=data_collator,
   compute_metrics=compute_metrics,
)
trainer.train()
```

You might notice that training has become much faster. That is because we are only training the classification head, which provides us with a significant speedup compared to fine-tuning the entire model:

```
trainer.evaluate()
```

```
{'eval_loss': 0.6821751594543457,
 'eval_f1': 0.6331058020477816,
 'eval_runtime': 4.0175,
 'eval_samples_per_second': 265.337,
 'eval_steps_per_second': 16.677,
 'epoch': 1.0}
```

When we evaluate the model, we only get an F1 score of 0.63, which is quite a bit lower compared to our original 0.85 score. Instead of freezing nearly all layers, let's freeze everything up until encoder block 10 as illustrated in Figure 11-6, and see how it affects performance. A major benefit is that this reduces computation but still allows updates to flow through part of the pretrained model:

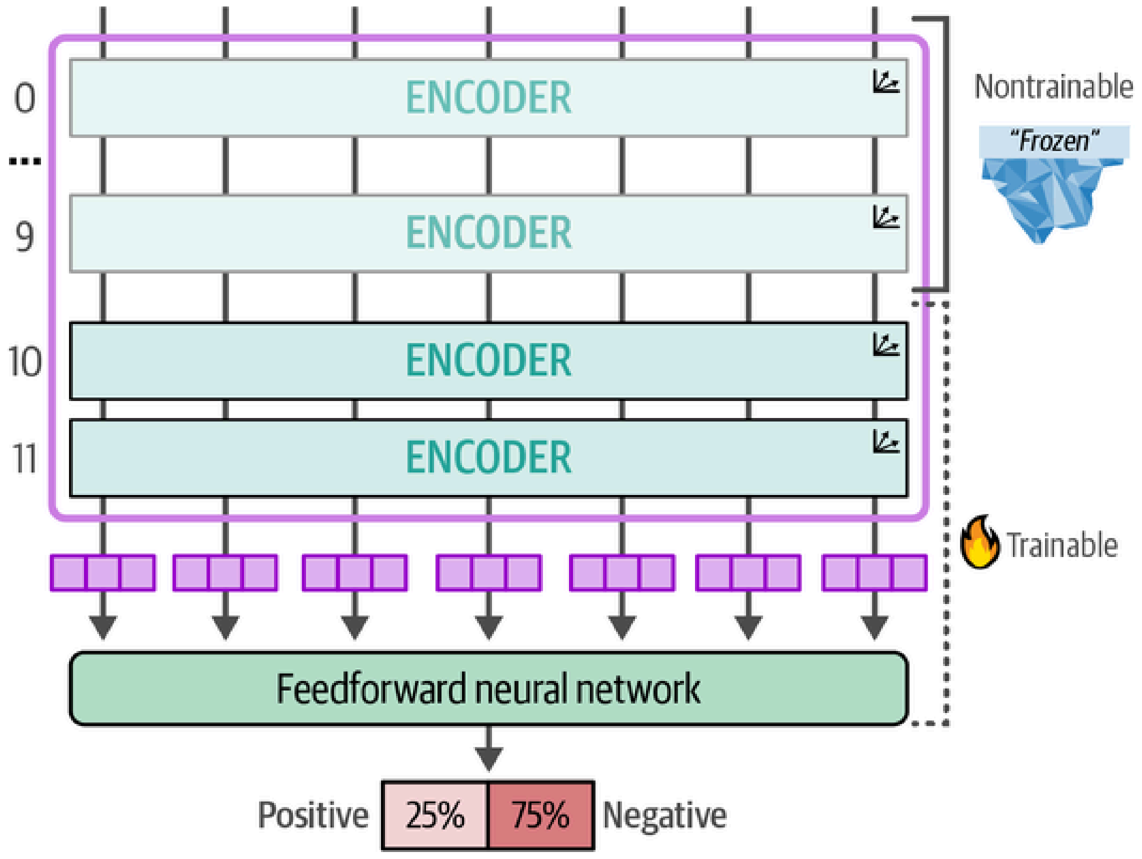


Figure 11-6. We freeze the first 10 encoder blocks of our BERT model. Everything else is trainable and will be fine-tuned.

```
# Load model
model_id = "bert-base-cased"
model = AutoModelForSequenceClassification.from_pretrained(
    model_id, num_labels=2
)
tokenizer = AutoTokenizer.from_pretrained(model_id)

# Encoder block 11 starts at index 165 and
# we freeze everything before that block
for index, (name, param) in enumerate(model.named_parameters()):
    if index < 165:
        param.requires_grad = False

# Trainer which executes the training process
trainer = Trainer(
   model=model,
   args=training_args,
   train_dataset=tokenized_train,
   eval_dataset=tokenized_test,
   tokenizer=tokenizer,
   data_collator=data_collator,
   compute_metrics=compute_metrics,
)
trainer.train()
```

After training, we evaluate the results:

```
trainer.evaluate()
{'eval_loss': 0.40812647342681885,
 'eval_f1': 0.8,
 'eval_runtime': 3.7125,
 'eval_samples_per_second': 287.137,
 'eval_steps_per_second': 18.047,
 'epoch': 1.0}
```

We got an F1 score of 0.8, which is much higher than our previous score of 0.63 when freezing all layers. It demonstrates that although we generally want to train as many layers as possible, you can get away with training less if you do not have the necessary computing power.

To further illustrate this effect, we tested the effect of iteratively freezing encoder blocks and fine-tuning them as we did thus far. As shown in Figure 11-7, training only the first five encoder blocks (red vertical line) is enough to almost reach the performance of training all encoder blocks.

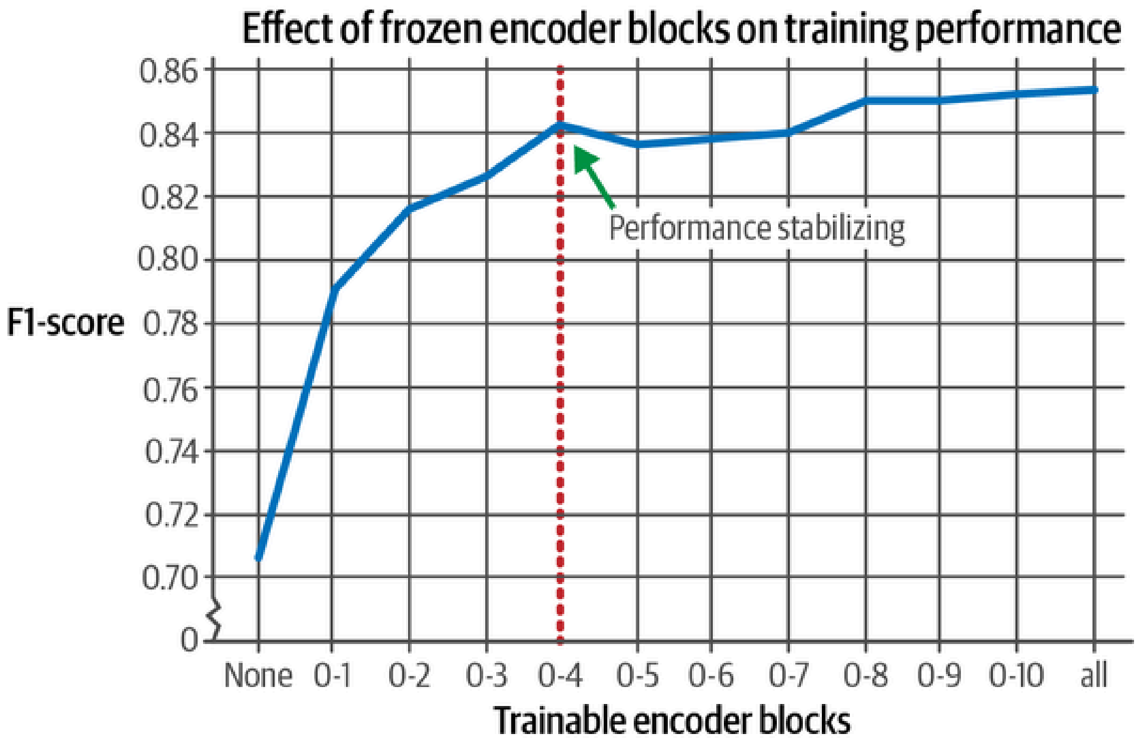


Figure 11-7. The effect of freezing certain encoder blocks on the performance of the model. Training more blocks leads to improved performance but stabilizes early on.

NOTE

When you are training for multiple epochs, the difference (in training time and resources) between freezing and not freezing often becomes larger. It is therefore advised to play around with a balance that works for you.

Few-Shot Classification

Few-shot classification is a technique within supervised classification where you have a classifier learn target labels based on only a few labeled examples. This technique is great when you have a classification task but do not have many labeled data points readily available. In other words, this method allows you to label a few high-quality data points per class on which to train the model. This idea of using a few labeled data points for training your model is shown in Figure 11-8.

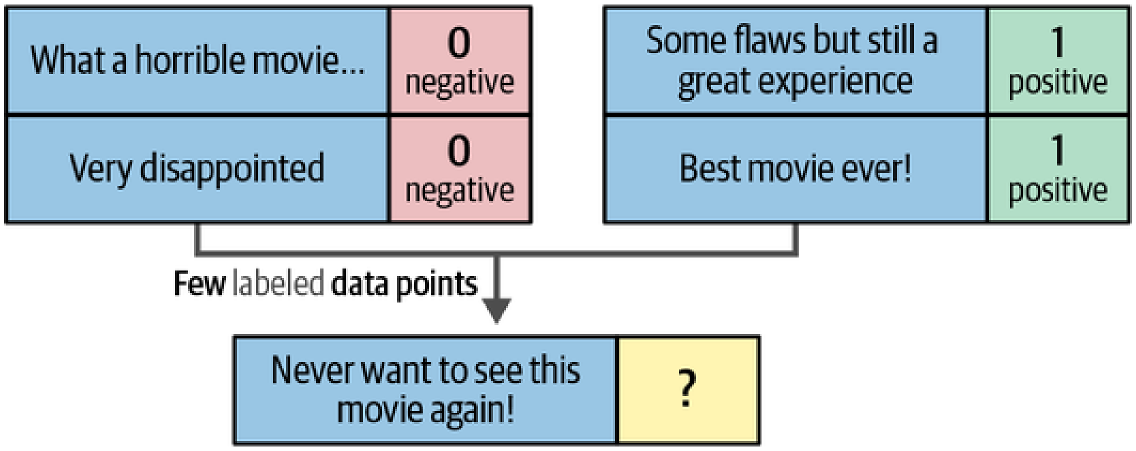


Figure 11-8. In few-shot classification, we only use a few labeled data points to learn from.

SetFit: Efficient Fine-Tuning with Few Training Examples

To perform few-shot text classification, we use an efficient framework called SetFit.[2] It is built on top of the architecture of `sentence-transformers` to generate high-quality textual representations that are updated during training. Only a few labeled examples are needed for this framework to be competitive with fine-tuning a BERT-like model on a large, labeled dataset as we explored in the previous example.

The underlying algorithm of SetFit consists of three steps:

1. *Sampling training data*

   Based on in-class and out-class selection of labeled data it generates positive (similar) and negative (dissimilar) pairs of sentences

2. *Fine-tuning embeddings*

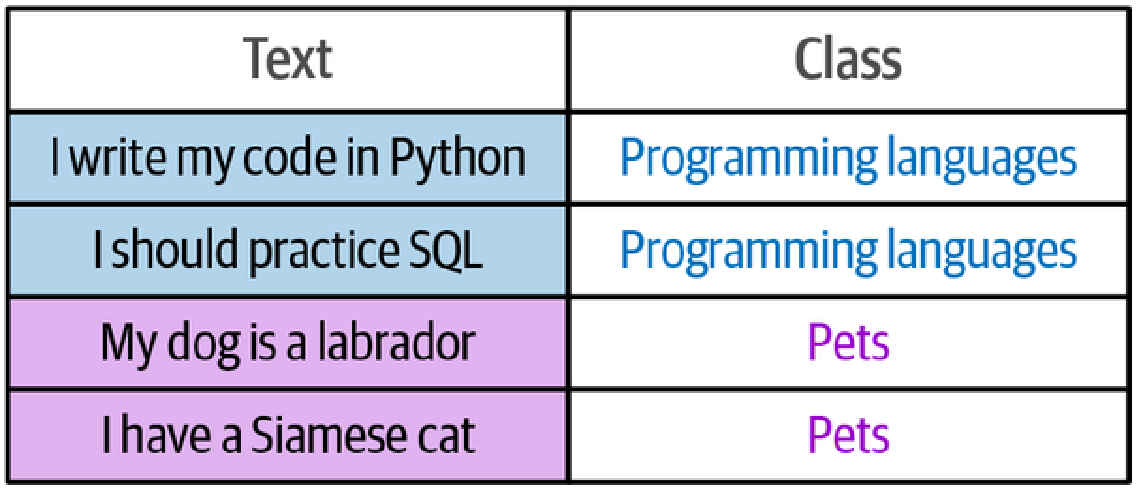
   Fine-tuning a pretrained embedding model based on the previously generated training data

3. *Training a classifier*

   Create a classification head on top of the embedding model and train it using the previously generated training data

Before fine-tuning an embedding model, we need to generate training data. The model assumes the training data to be samples of positive (similar) and negative (dissimilar) pairs of sentences. However, when we are dealing with a classification task, our input data is generally not labeled as such.

Say, for example, we have the training dataset in Figure 11-9 that classifies text into two categories: text about programming languages, and text about pets.

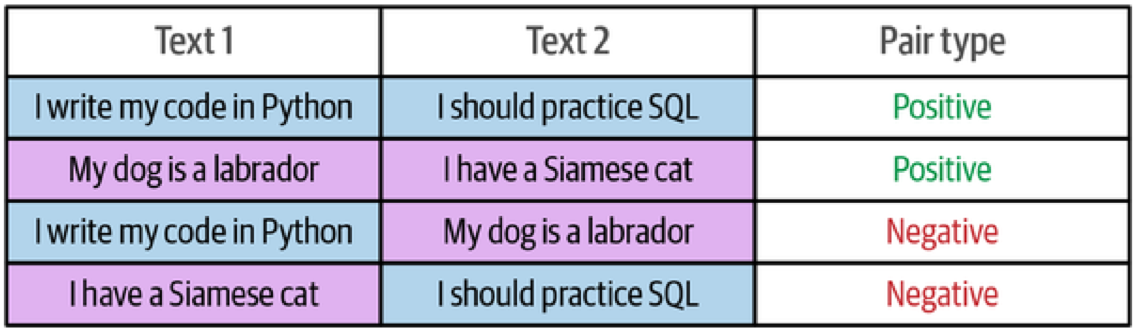


Text	Class
I write my code in Python	Programming languages
I should practice SQL	Programming languages
My dog is a labrador	Pets
I have a Siamese cat	Pets

Figure 11-9. Data in two classes: text about programming languages and text about pets.

In step 1, SetFit handles this problem by generating the necessary data based on in-class and out-class selection as we illustrate in Figure 11-10.

For example, when we have 16 sentences about sports, we can create 16 * (16 – 1) / 2 = *120* pairs that we label as *positive* pairs. We can use this process to generate *negative* pairs by collecting pairs from different classes.



Text 1	Text 2	Pair type
I write my code in Python	I should practice SQL	Positive
My dog is a labrador	I have a Siamese cat	Positive
I write my code in Python	My dog is a labrador	Negative
I have a Siamese cat	I should practice SQL	Negative

Figure 11-10. Step 1: sampling training data. We assume sentences within a class are similar and create positive pairs while sentences in different classes become negative pairs.

In step 2, we can use the generated sentence pairs to fine-tune the embedding model. This leverages a method called contrastive learning to fine-tune a pretrained BERT model. As we reviewed in Chapter 10, contrastive learning allows accurate sentence embeddings to be learned from pairs of similar (positive) and dissimilar (negative) sentences.

Since we generated these pairs in the previous step, we can use them to fine-tune a `SentenceTransformers` model. Although we have discussed contrastive learning before, we again illustrate the method in Figure 11-11 as a refresher.

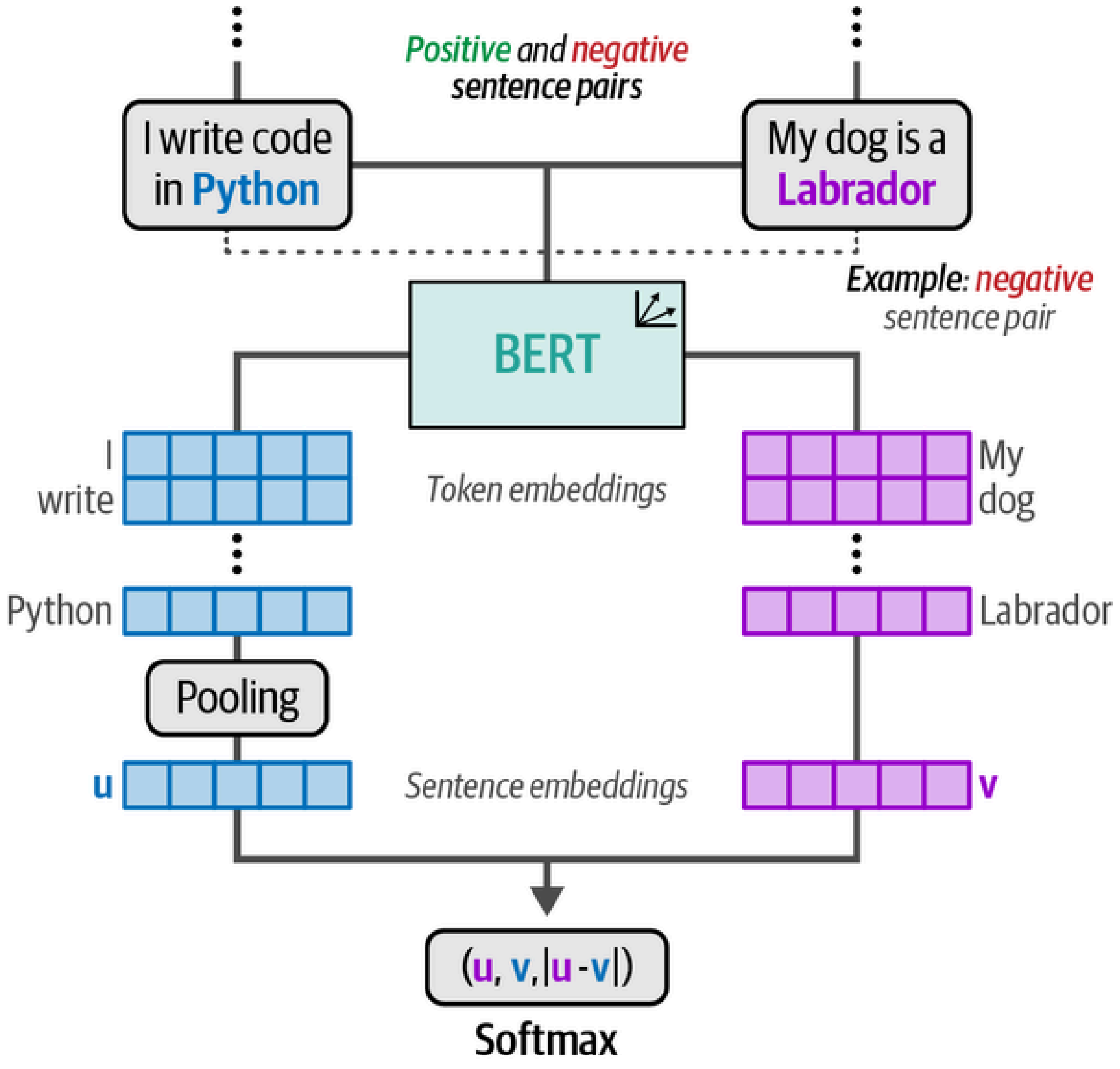
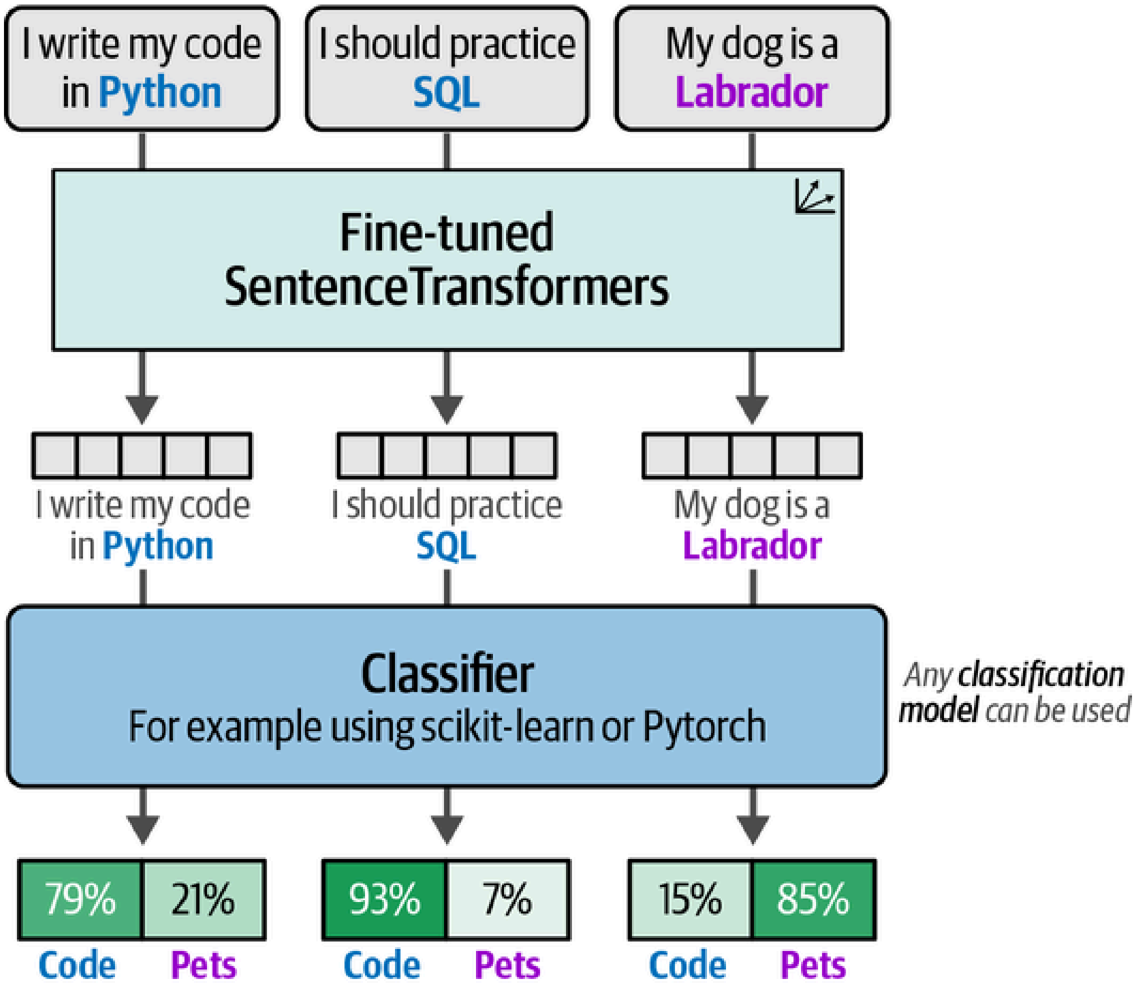


Figure 11-11. Step 2: Fine-tuning a `SentenceTransformers` model. Using contrastive learning, embeddings are learned from positive and negative sentence pairs.

The goal of fine-tuning this embedding model is that it can create embeddings that are tuned to the classification task. The relevance of the classes, and their relative meaning, are distilled into the embeddings through fine-tuning the embedding model.

In step 3, we generate embeddings for all sentences and use those as the input of a classifier. We can use the fine-tuned `SentenceTransformers` model to convert our sentences into embeddings that we can use as features. The classifier learns from our fine-tuned embeddings to accurately predict unseen sentences. This last step is illustrated in Figure 11-12.



I write my code in Python
I should practice SQL
My dog is a Labrador
Fine-tuned SentenceTransformers
I write my code in Python
I should practice SQL
My dog is a Labrador
Classifier
For example using scikit-learn or Pytorch
Any classification model can be used
79%
21%
93%
7%
15%
85%
Code
Pets
Code
Pets
Code
Pets

Figure 11-12. Step 3: Training a classifier. The classifier can be any scikit-learn model or a classification head.

When we put all the steps together, we get an efficient and elegant pipeline for performing classification when you only have a few labels per class. It cleverly makes use of the idea that we have labeled data, although not in the way that we would like it. The three steps together are illustrated in Figure 11-13 to give a single overview of the entire procedure.

First, sentence pairs are generated based on in-class and out-class selection. Second, the sentence pairs are used to fine-tune a pretrained `SentenceTransformer` model. Third, the sentences are embedded with the fine-tuned model on which a classifier is trained to predict the classes.

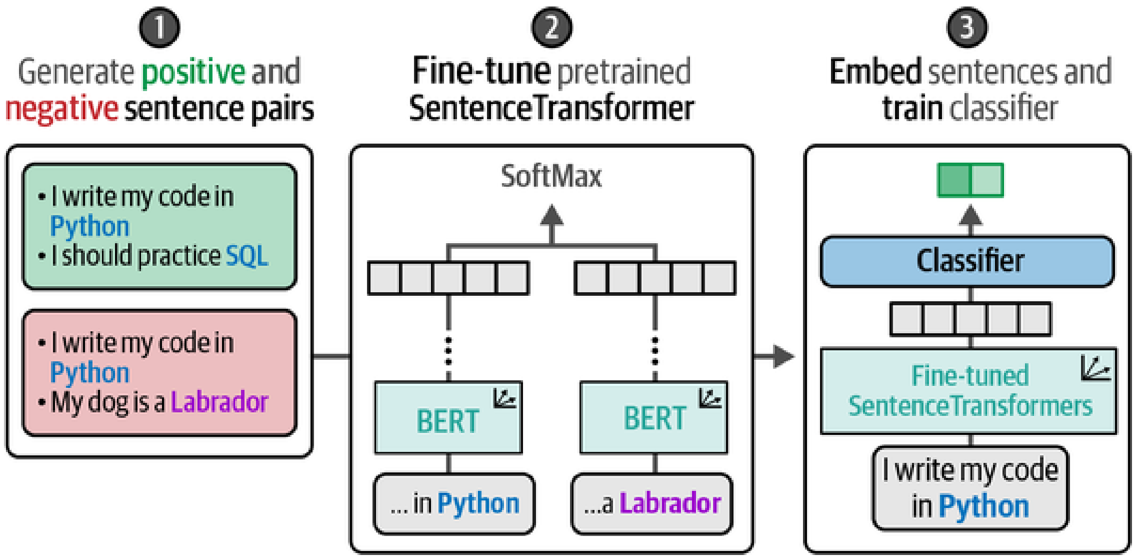

1
Generate positive and negative sentence pairs
I write my code in Python
I should practice SQL
I write my code in Python
My dog is a Labrador
2
Fine-tune pretrained SentenceTransformer
SoftMax
BERT
BERT
... in Python
...a Labrador
3
Embed sentences and train classifier
Classifier
Fine-tuned SentenceTransformers
I write my code in Python

Figure 11-13. The three main steps of SetFit.

Fine-Tuning for Few-Shot Classification

We previously trained on a dataset containing roughly 8,500 movie reviews. However, since this is a few-shot setting, we will only sample 16 examples per class. With two classes, we will only have 32 documents to train on compared to the 8,500 movie reviews we used before!

```
from setfit import sample_dataset

# We simulate a few-shot setting by sampling 16 examples per class
sampled_train_data = sample_dataset(tomatoes["train"], num_samples=16)
```

After sampling the data, we choose a pretrained `SentenceTransformer` model to fine-tune. The official documentation contains an overview of pretrained `SentenceTransformer` models from which we are going to be using `"sentence-transformers/all-mpnet-base-v2"`. It is one of the best-performing models on the MTEB leaderboard, which shows the performance of embedding models across a variety of tasks:

```
from setfit import SetFitModel

# Load a pretrained SentenceTransformer model
model = SetFitModel.from_pretrained("sentence-transformers/all-mpnet-base-v2")
```

After loading in the pretrained `SentenceTransformer` model, we can start defining our `SetFitTrainer`. By default, a logistic regression model is chosen as the classifier to train.

Similar to what we did with Hugging Face Transformers, we can use the trainer to define and play around with relevant parameters. For example, we set the `num_epochs` to 3 so that contrastive learning will be performed for three epochs:

```
from setfit import TrainingArguments as SetFitTrainingArguments
from setfit import Trainer as SetFitTrainer

# Define training arguments
args = SetFitTrainingArguments(
    num_epochs=3, # The number of epochs to use for contrastive learning
    num_iterations=20  # The number of text pairs to generate
)
args.eval_strategy = args.evaluation_strategy

# Create trainer
trainer = SetFitTrainer(
    model=model,
    args=args,
    train_dataset=sampled_train_data,
    eval_dataset=test_data,
    metric="f1"
)
```

We only need to call `train` to start the training loop. When we do, we should get the following output:

```
# Training loop
trainer.train()
```

```
***** Running training *****
  Num unique pairs = 1280
  Batch size = 16
```

```
    Num epochs = 3
    Total optimization steps = 240
```

Notice that the output mentions that 1,280 sentence pairs were generated for fine-tuning the `SentenceTransformer` model. As a default, 20 sentence pair combinations are generated for each sample in our data, which would be 20 * 32 = 680 samples. We will have to multiply this value by 2 for each positive and negative pair generated, 680 * 2 = 1,280 sentence pairs. Generating 1,280 sentence pairs is quite impressive considering we only had 32 labeled sentences to start with!

TIP

When we do not specifically define a classification head, by default a logistic regression is used. If we would like to specify a classification head ourselves, we can do so by specifying the following model in `SetFitTrainer`:

```python
# Load a SetFit model from Hub
model = SetFitModel.from_pretrained(
    "sentence-transformers/all-mpnet-base-v2",
    use_differentiable_head=True,
    head_params={"out_features": num_classes},
)

# Create trainer
trainer = SetFitTrainer(
    model=model,
    ...
)
```

Here, `num_classes` refers to the number of classes that we want to predict.

Next, let's evaluate the model to get a feeling of its performance:

```
# Evaluate the model on our test data
trainer.evaluate()
```

```
{'f1': 0.8363988383349468}
```

With only 32 labeled documents, we get an F1 score of 0.85. Considering that the model was trained on a tiny subset of the original data, this is very impressive! Moreover, in Chapter 2, we got the same performance but instead trained a logistic regression model on the embeddings of the full data. Thus, this pipeline demonstrates the potential of taking the time to label just a few instances.

TIP

Not only can SetFit perform few-shot classification tasks, but it also has support for when you have no labels at all, also called zero-shot classification. SetFit generates synthetic examples from the label names to resemble the classification task and then trains a SetFit model on them. For example, if the target labels are "happy" and "sad," then synthetic data could be "The example is happy" and "This example is sad."

Continued Pretraining with Masked Language Modeling

In the examples thus far, we leveraged a pretrained model and fine-tuned it to perform classification. This process describes a two-step process: first pretraining a model (which was already done for us) and then fine-tuning it for a particular task. We illustrate this process in Figure 11-14.

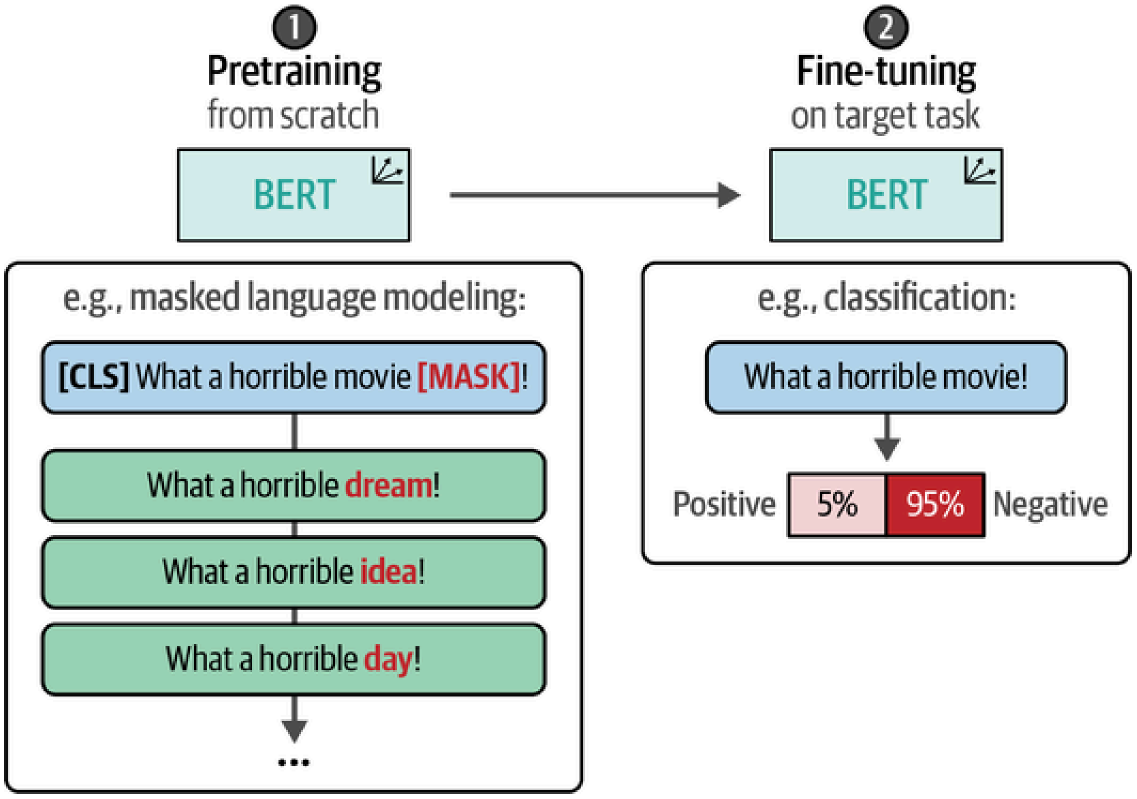


Figure 11-14. To fine-tune the model on a target task—for example, classification—we either start with pretraining a BERT model or use a pretrained one.

This two-step approach is typically used throughout many applications. It has its limitations when faced with domain-specific data. The pretrained model is often trained on very general data, like Wikipedia pages, and might not be tuned to your domain-specific words.

Instead of adopting this two-step approach, we can squeeze another step between them, namely continue pretraining an already pretrained BERT model. In other words, we can simply continue training the BERT model using masked language modeling (MLM) but instead use data from our domain. It is like going from a general BERT model to a BioBERT model specialized for the medical domain, to a fine-tuned BioBERT model to classify medication.

This will update the subword representations to be more tuned toward words it would not have seen before. This process is illustrated in Figure 11-15 and demonstrates how this additional step updates a masked language modeling task. Continuing pretraining on a pretrained BERT model has been shown to improve the performance of models in classification tasks and is a worthwhile addition to the fine-tuning pipeline.[3]

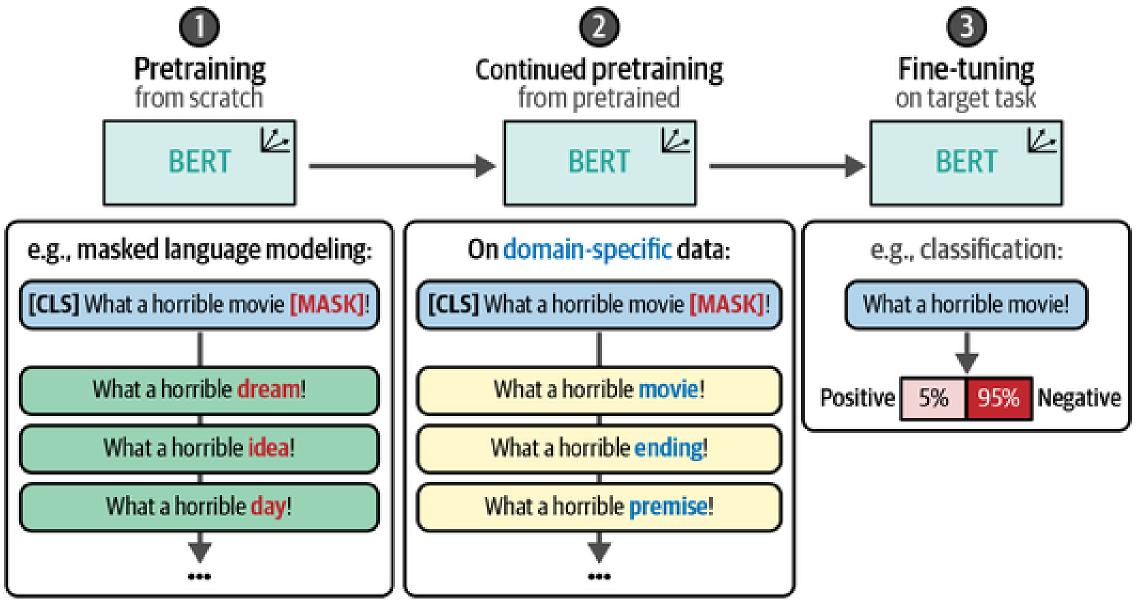


Figure 11-15. Instead of a two-step approach, we can add another step that continues to pretrain the pretrained model before fine-tuning it on the target task. Notice how the masks were filled with abstract concepts in 1 while they were filled with movie-specific concepts in 2.

Instead of having to pretrain an entire model from scratch, we can simply continue pretraining before fine-tuning it for classification. This also helps the model to adapt to a certain domain or even the lingo of a specific organization. The genealogy of models a company might want to adopt is further illustrated in Figure 11-16.

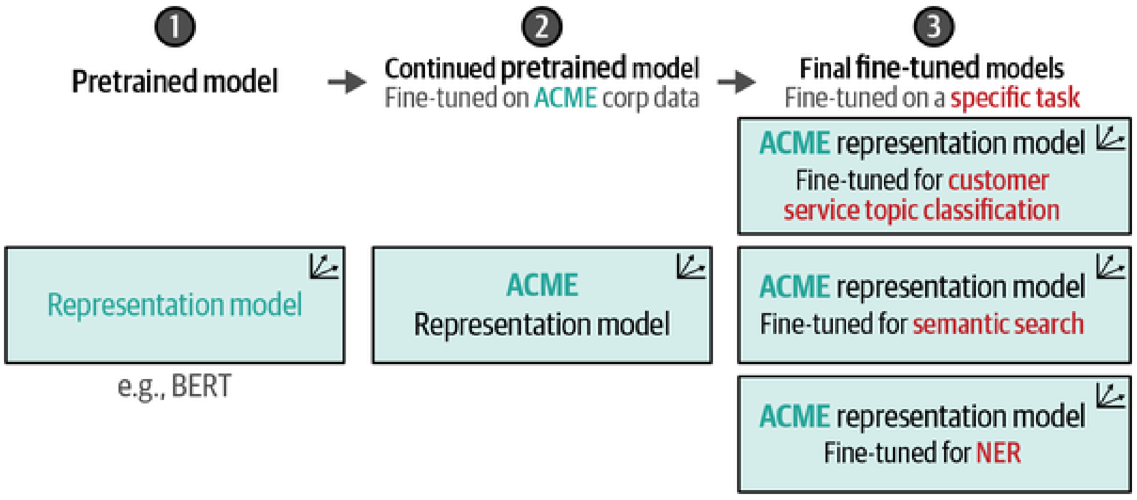

1 Pretrained model
2 Continued pretrained model
Fine-tuned on ACME corp data
3 Final fine-tuned models
Fine-tuned on a specific task
ACME representation model
Fine-tuned for customer service topic classification
Representation model
e.g., BERT
ACME Representation model
ACME representation model
Fine-tuned for semantic search
ACME representation model
Fine-tuned for NER

Figure 11-16. The three-step approach illustrated for specific use cases.

In this example, we will demonstrate how to apply step 2 and continue pretraining an already pretrained BERT model. We use the same data that we started with, namely the Rotten Tomatoes reviews.

We start by loading the `"bert-base-cased"` model we have used thus far and prepare it for MLM:

```
from transformers import AutoTokenizer, AutoModelForMaskedLM

# Load model for masked language modeling (MLM)
model = AutoModelForMaskedLM.from_pretrained("bert-base-cased")
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
```

We need to tokenize the raw sentences. We will also remove the labels since this is not a supervised task:

```
def preprocess_function(examples):
   return tokenizer(examples["text"], truncation=True)

# Tokenize data
tokenized_train = train_data.map(preprocess_function, batched=True)
tokenized_train = tokenized_train.remove_columns("label")
tokenized_test = test_data.map(preprocess_function, batched=True)
tokenized_test = tokenized_test.remove_columns("label")
```

Previously, we used `DataCollatorWithPadding`, which dynamically pads the input it receives.

Instead, we will have a `DataCollator` that will perform the masking of tokens for us. There are two methods that are generally used for this: token and whole-word masking. With token masking, we randomly mask 15% of the tokens in a sentence. It might happen that part of a word will be masked. To enable masking of the entire word, we could apply whole-word masking, as illustrated in Figure 11-17.

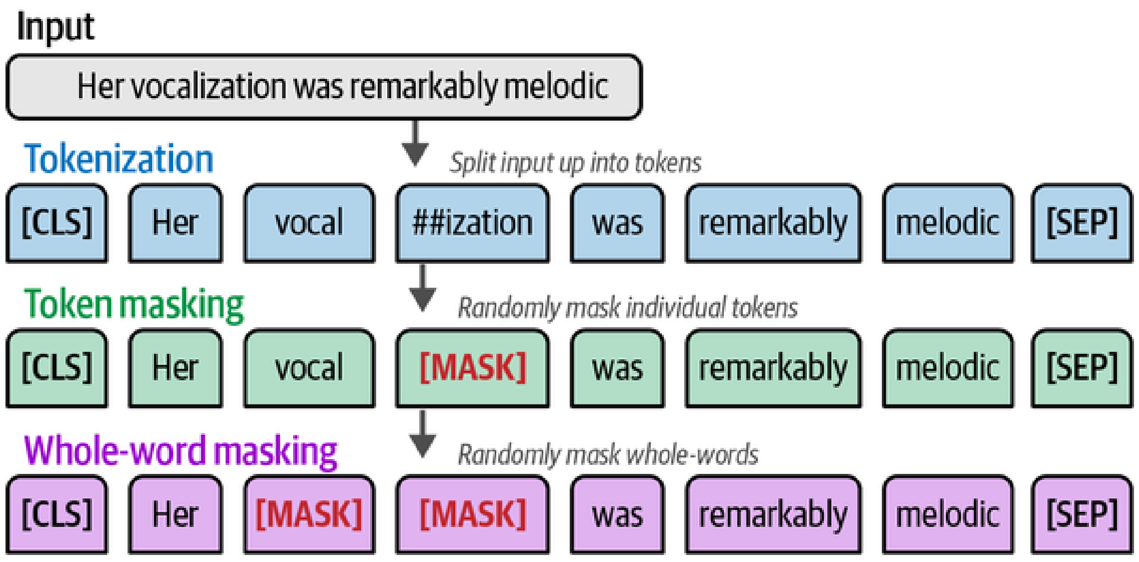


Figure 11-17. Different methods for randomly masking tokens.

Generally, predicting whole words tends to be more complicated than tokens, which makes the model perform better as it needs to learn more accurate and precise representations during training. However, it tends to take a bit more time to converge. We will be going with token masking in this example using `DataCollatorForLanguageModeling` for faster convergence. However, we can use whole-word masking by replacing `DataCollatorForLanguageModeling` with `DataCollatorForWholeWordMask`. Lastly, we set the probability that a token is masked in a given sentence to 15% (`mlm_probability`):

```
from transformers import DataCollatorForLanguageModeling

# Masking Tokens
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer,
    mlm=True,
    mlm_probability=0.15
)
```

Next, we will create the `Trainer` for running the MLM task and specify certain parameters:

```
# Training arguments for parameter tuning
training_args = TrainingArguments(
   "model",
   learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=10,
    weight_decay=0.01,
    save_strategy="epoch",
    report_to="none"
)

# Initialize Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_train,
    eval_dataset=tokenized_test,
    tokenizer=tokenizer,
    data_collator=data_collator
)
```

Several parameters are worth noting. We train for 20 epochs and keep the task short. You can experiment with the learning rate and weight decay to ascertain whether they assist in fine-tuning the model.

Before we start our training loop we will first save our pretrained tokenizer. The tokenizer is not updated during training so there is no need to save it after training. We will, however, save our model after we continue pretraining:

```
# Save pre-trained tokenizer
tokenizer.save_pretrained("mlm")

# Train model
trainer.train()

# Save updated model
model.save_pretrained("mlm")
```

This gives us an updated model in the *mlm* folder. To evaluate its performance we would normally fine-tune the model on a variety of tasks. For our purposes, however, we can run some masking tasks to see if it has learned from its continued training.

We will do so by loading in our pretrained model before we continue pre-training. Using the sentence `"What a horrible [MASK]!"` the model will predict which word would be in place of `"[MASK]"`:

```
from transformers import pipeline

# Load and create predictions
mask_filler = pipeline("fill-mask", model="bert-base-cased")
preds = mask_filler("What a horrible [MASK]!")

# Print results
for pred in preds:
    print(f">>> {pred["sequence"]}")
```

```
>>> What a horrible idea!
>>> What a horrible dream!
>>> What a horrible thing!
>>> What a horrible day!
>>> What a horrible thought!
```

The output demonstrates concepts like "idea," "dream," and "day," which definitely make sense. Next, let's see what our updated model predicts:

```
# Load and create predictions
mask_filler = pipeline("fill-mask", model="mlm")
preds = mask_filler("What a horrible [MASK]!")

# Print results
for pred in preds:
    print(f">>> {pred["sequence"]}")
```

```
>>> What a horrible movie!
>>> What a horrible film!
>>> What a horrible mess!
>>> What a horrible comedy!
>>> What a horrible story!
```

A horrible movie, film, mess, etc. clearly shows us that the model is more biased toward the data that we fed it compared to the pretrained model.

The next step would be to fine-tune this model on the classification task that we did at the beginning of this chapter. Simply load the model as follows and you are good to go:

```
from transformers import AutoModelForSequenceClassification

# Fine-tune for classification
model = AutoModelForSequenceClassification.from_pretrained("mlm", num_labels=2)
tokenizer = AutoTokenizer.from_pretrained("mlm")
```

Named-Entity Recognition

In this section, we will delve into the process of fine-tuning a pretrained BERT model specifically for NER (named-entity recognition). Instead of classifying entire documents, this procedure allows for the classification of individual tokens and/or words, including people and locations. This is especially helpful for de-identification and anonymization tasks when there is sensitive data.

NER shares similarities with the classification example we explored at the beginning of this chapter. Nevertheless, a key distinction lies in the preprocessing and classification of data. Given that we are focusing on classifying individual words instead of entire documents, we must preprocess the data to consider this granular structure. Figure 11-18 provides a visual representation of this word-level approach.

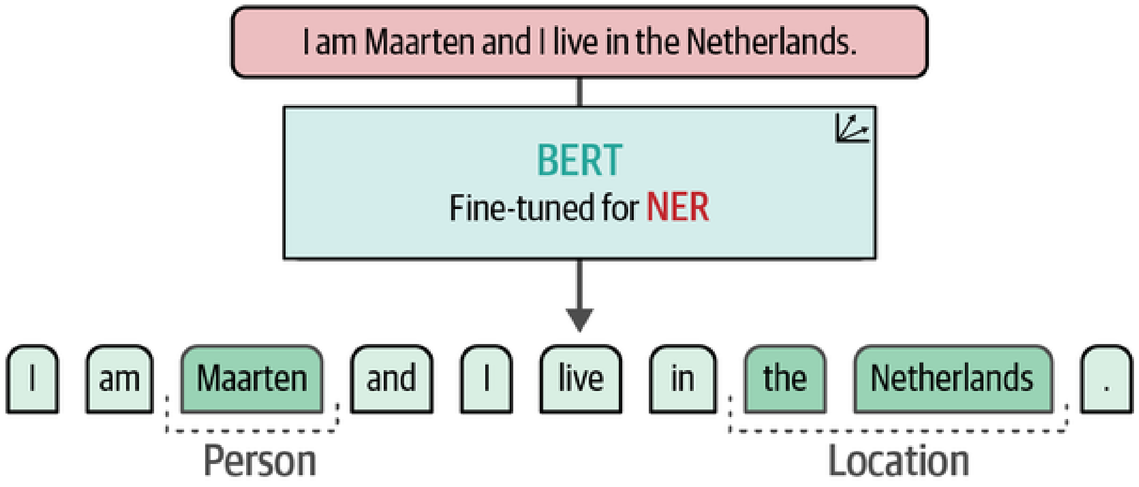


Figure 11-18. Fine-tuning a BERT model for NER allows for the detection of named entities, such as people or locations.

Fine-tuning the pretrained BERT model follows a similar architecture akin to what we observed with document classification. However, there is a fundamental shift in the classification approach. Rather than relying on the aggregation or pooling of token embeddings, the model now makes predictions for individual tokens in a sequence. It is crucial to emphasize that our word-level classification task does not entail classifying entire words, but rather the tokens that collectively constitute those words. Figure 11-19 provides a visual representation of this token-level classification.

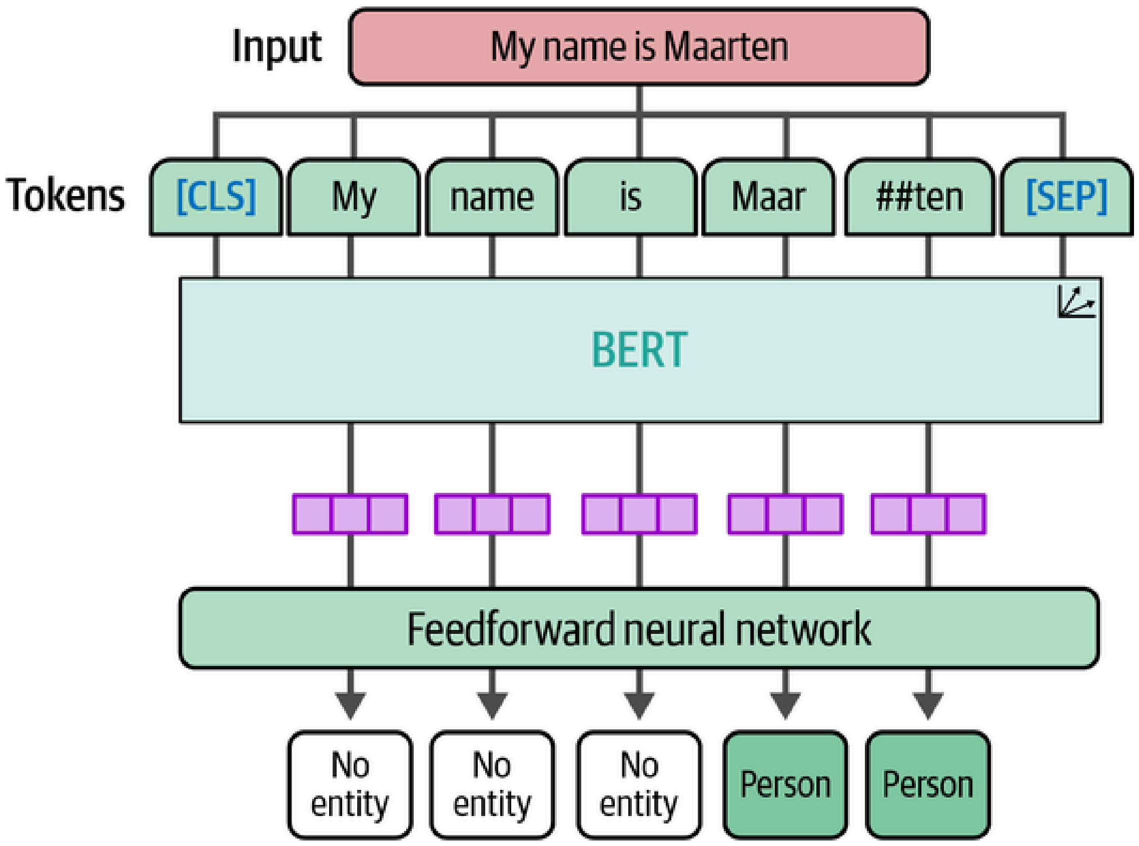


Figure 11-19. During the fine-tuning process of a BERT model, individual tokens are classified instead of words or entire documents.

Preparing Data for Named-Entity Recognition

In this example, we will use the English version of the `CoNLL-2003` dataset, which contains several different types of named entities (person, organization, location, miscellaneous, and no entity) and has roughly 14,000 training samples.[4]

```
# The CoNLL-2003 dataset for NER
dataset = load_dataset("conll2003", trust_remote_code=True)
```

TIP

While researching datasets to use for this example, there were a few more that we wanted to share. `wnut_17` is a task that focuses on emerging and rare entities, those that are more difficult to spot. Furthermore, the `tner/mit_movie_trivia` and `tner/mit_restaurant` datasets are quite fun to use. `tner/mit_movie_trivia` is for detecting entities like actor, plot, and soundtrack whereas `tner/mit_restaurant` aims to detect entities such as amenity, dish, and cuisine.[5]

Let’s inspect the structure of the data with an example:

```
example = dataset["train"][848]
example
```

```
{'id': '848',
 'tokens': ['Dean',
  'Palmer',
  'hit',
  'his',
  '30th',
  'homer',
  'for',
  'the',
  'Rangers',
  '.'],
 'pos_tags': [22, 22, 38, 29, 16, 21, 15, 12, 23, 7],
 'chunk_tags': [11, 12, 21, 11, 12, 12, 13, 11, 12, 0],
 'ner_tags': [1, 2, 0, 0, 0, 0, 0, 0, 3, 0]}
```

This dataset provides us with labels for each word given in a sentence. These labels can be found in the `ner_tags` key, which refers to the following possible entities:

```
label2id = {
    "O": 0, "B-PER": 1, "I-PER": 2, "B-ORG": 3, "I-ORG": 4,
    "B-LOC": 5, "I-LOC": 6, "B-MISC": 7, "I-MISC": 8
}
id2label = {index: label for label, index in label2id.items()}
label2id
```

```
{'O': 0,
 'B-PER': 1,
 'I-PER': 2,
 'B-ORG': 3,
 'I-ORG': 4,
 'B-LOC': 5,
 'I-LOC': 6,
 'B-MISC': 7,
 'I-MISC': 8}
```

These entities correspond to specific categories: a person (PER), organization (ORG), location (LOC), miscellaneous entities (MISC), and no entity (O). Note that these entities are prefixed with either a B (beginning) or an I (inside). If two tokens that follow each other are part of the same phrase, then the start of that phrase is indicated with B, which is followed by an I to show that they belong to each other and are not independent entities.

This process is further illustrated in Figure 11-20. In the figure, since "Dean" is the start of the phrase and "Palmer" is the end, we know that "Dean Palmer" is a person and that "Dean" and "Palmer" are not individual people.

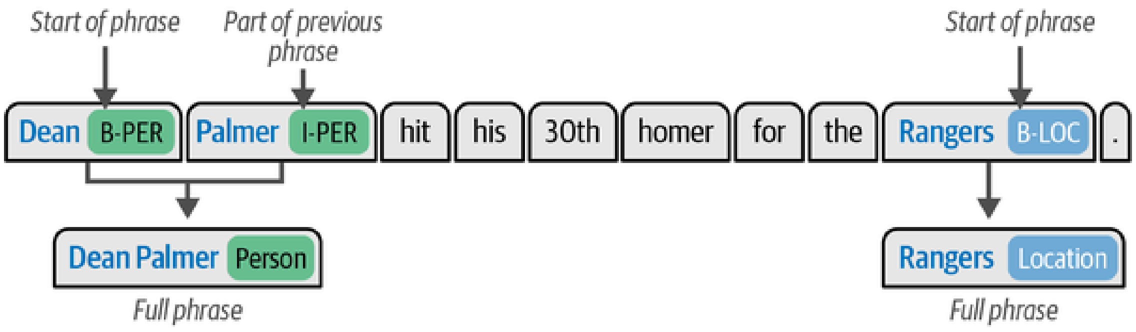


Figure 11-20. By indicating the start and end of the phrase with the same entity, we can recognize entities of entire phrases.

Our data is preprocessed and split up into words but not yet tokens. To do so, we will tokenize it further with the tokenizer of the pretrained model we used throughout this chapter, namely `bert-base-cased`:

```
from transformers import AutoModelForTokenClassification

# Load tokenizer
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")

# Load model
model = AutoModelForTokenClassification.from_pretrained(
    "bert-base-cased",
    num_labels=len(id2label),
    id2label=id2label,
    label2id=label2id
)
```

Let’s explore how the tokenizer would process our example:

```
# Split individual tokens into sub-tokens
token_ids = tokenizer(example["tokens"], is_split_into_words=True)["input_ids"]
sub_tokens = tokenizer.convert_ids_to_tokens(token_ids)
sub_tokens
```

```
['[CLS]',
 'Dean',
 'Palmer',
 'hit',
 'his',
 '30th',
 'home',
 '##r',
 'for',
 'the',
 'Rangers',
 '.',
 '[SEP]']
```

The tokenizer added the `[CLS]` and `[SEP]` tokens as we learned in Chapters 2 and 3. Note that the word `'homer'` was further split up into the tokens `'home'` and `'##r'`. This creates a bit of a problem for us since we have labeled data at the word level but not at the token level. This can be resolved by aligning the labels with their subtoken counterparts during tokenization.

Let’s consider the word `'Maarten'`, which has the label B-PER to signal that this is a person. If we pass that word through the tokenizer, it splits the word up into the tokens `'Ma'`, `'##arte'`, and `'##n'`. We cannot use the B-PER entity for all tokens as that would signal that the three tokens are all independent people. Whenever an entity is split into tokens, the first token should have B (for beginning) and the following should be I (for inner).

Therefore, `'Ma'` will get the B-PER to signal the start of a phrase, and `'##arte'`, and `'##n'` will get the I-PER to signal they belong to a phrase. This alignment process is illustrated in Figure 11-21.

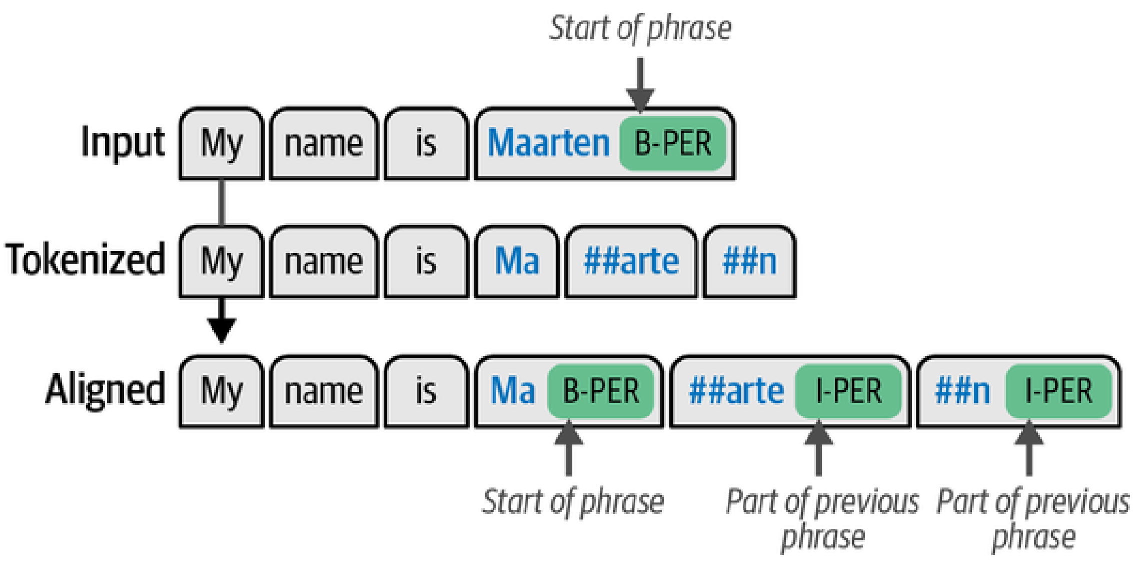


Figure 11-21. The alignment process of labeling tokenized input.

We create a function, `align_labels`, that will tokenize the input and align these tokens with their updated labels during tokenization:

```
def align_labels(examples):
    token_ids = tokenizer(
        examples["tokens"],
        truncation=True,
        is_split_into_words=True
    )
    labels = examples["ner_tags"]

    updated_labels = []
    for index, label in enumerate(labels):

        # Map tokens to their respective word
        word_ids = token_ids.word_ids(batch_index=index)
        previous_word_idx = None
        label_ids = []
        for word_idx in word_ids:
            # The start of a new word
            if word_idx != previous_word_idx:

                previous_word_idx = word_idx
                updated_label = -100 if word_idx is None else label[word_idx]
                label_ids.append(updated_label)

            # Special token is -100
            elif word_idx is None:
                label_ids.append(-100)

            # If the label is B-XXX we change it to I-XXX
            else:
                updated_label = label[word_idx]
                if updated_label % 2 == 1:
                    updated_label += 1
                label_ids.append(updated_label)

        updated_labels.append(label_ids)

    token_ids["labels"] = updated_labels
    return token_ids

tokenized = dataset.map(align_labels, batched=True)
```

Looking at our example, note that additional labels (`-100`) were added for the `[CLS]` and `[SEP]` tokens:

```
# Difference between original and updated labels
print(f"Original: {example["ner_tags"]}")
print(f"Updated: {tokenized["train"][848]["labels"]}")
```

```
Original: [1, 2, 0, 0, 0, 0, 0, 0, 3, 0]
Updated: [-100, 1, 2, 0, 0, 0, 0, 0, 0, 0, 3, 0, -100]
```

Now that we have tokenized and aligned the labels, we can start thinking about defining our evaluation metrics. This is also different from what we have seen before. Instead of a single prediction per document, we now have multiple predictions per document, namely per token.

We will make use of the `evaluate` package by Hugging Face to create a `compute_metrics` function that allows us to evaluate performance on a token level:

```python
import evaluate

# Load sequential evaluation
seqeval = evaluate.load("seqeval")

def compute_metrics(eval_pred):
    # Create predictions
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=2)

    true_predictions = []
    true_labels = []

    # Document-level iteration
    for prediction, label in zip(predictions, labels):

      # Token-level iteration
      for token_prediction, token_label in zip(prediction, label):

        # We ignore special tokens
        if token_label != -100:
          true_predictions.append([id2label[token_prediction]])
          true_labels.append([id2label[token_label]])

    results = seqeval.compute(
    predictions=true_predictions, references=true_labels
)
    return {"f1": results["overall_f1"]}
```

Fine-Tuning for Named-Entity Recognition

We are nearly there. Instead of `DataCollatorWithPadding`, we need a collator that works with classification on a token level, namely `DataCollatorForTokenClassification`:

```
from transformers import DataCollatorForTokenClassification

# Token-classification DataCollator
data_collator = DataCollatorForTokenClassification(tokenizer=tokenizer)
```

Now that we have loaded our model, the rest of the steps are similar to previous training procedures in this chapter. We define a trainer with specific arguments that we can tune and create a `Trainer`:

```
# Training arguments for parameter tuning
training_args = TrainingArguments(
   "model",
   learning_rate=2e-5,
   per_device_train_batch_size=16,
   per_device_eval_batch_size=16,
   num_train_epochs=1,
   weight_decay=0.01,
   save_strategy="epoch",
   report_to="none"
)

# Initialize Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized["train"],
    eval_dataset=tokenized["test"],
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)
trainer.train()
```

We then evaluate the model that we created:

```
# Evaluate the model on our test data
trainer.evaluate()
```

Lastly, let's save the model and use it in a pipeline for inference. This allows us to check certain data so we can manually inspect what happens during inference and if we are satisfied with the output:

```
from transformers import pipeline

# Save our fine-tuned model
trainer.save_model("ner_model")

# Run inference on the fine-tuned model
token_classifier = pipeline(
    "token-classification",
    model="ner_model",
)
token_classifier("My name is Maarten.")
```

```
[{'entity': 'B-PER',
  'score': 0.99534035,
  'index': 4,
  'word': 'Ma',
  'start': 11,
  'end': 13},
 {'entity': 'I-PER',
  'score': 0.9928328,
  'index': 5,
  'word': '##arte',
  'start': 13,
  'end': 17},
 {'entity': 'I-PER',
  'score': 0.9954301,
  'index': 6,
  'word': '##n',
  'start': 17,
  'end': 18}]
```

In the sentence `"My name is Maarten"`, the word `"Maarten"` and its subtokens were correctly identified as a person!

Summary

In this chapter, we explored several tasks for fine-tuning pretrained representation models on specific classification tasks. We started by demonstrating how to fine-tune a pretrained BERT model and extended the examples by freezing certain layers of its architectures.

We experimented with a few-shot classification technique called SetFit, which involves fine-tuning a pretrained embedding model together with a classification head using limited labeled data. Using only a few labeled data points, this model generated similar performance to the models we explored in earlier chapters.

Next, we delved into the concept of continued pretraining, where we used a pretrained BERT model as a starting point and continued training it using different data. The underlying process, masked language modeling, is not only used for creating a representation model but can also be used to continue pretraining models.

Finally, we looked at named-entity recognition, a task that involves identifying specific entities such as people and places in unstructured text. Compared to previous examples, this classification was done on a word level rather than on a document level.

In the next chapter, we continue with the field of fine-tuning language models but focus on generative models instead. Using a two-step process, we will explore how to fine-tune a generative model to properly follow instructions and then fine-tune it for human preference.

[1] Jacob Devlin et al. "BERT: Pre-training of deep bidirectional transformers for language understanding." *arXiv preprint arXiv:1810.04805* (2018).

[2] Lewis Tunstall et al. "Efficient few-shot learning without prompts." *arXiv preprint arXiv:2209.11055* (2022).

3 Chi Sun et al. “How to fine-tune GERT for text classification?” *Chinese Computational Linguistics: 18th China National Conference*, CCL 2019, Kunming, China, October 18–20, 2019, proceedings 18. Springer International Publishing, 2019.

4 Erik F. Sang and Fien De Meulder. “Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition.” *arXiv preprint cs/0306050* (2003).

5 Jingjing Liu et al. “Asgard: A portable architecture for multilingual dialogue systems.” *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013.