

Chapter 6: Fine-Tuning Language Models for Token Classification



In this chapter, we will learn about fine-tuning language models for token classification. Tasks such as **Named Entity Recognition (NER)**, **Part-of-Speech (POS)** tagging, and **Question Answering (QA)** are explored in this chapter. We will learn how a specific language model can be fine-tuned on such tasks. We will focus on BERT more than other language models. You will learn how to apply POS, NER, and QA using BERT. You will get familiar with the theoretical details of these tasks such as their respective datasets and how to perform them. After finishing this chapter, you will be able to perform any token classification using Transformers.

In this chapter, we will fine-tune BERT for the following tasks: fine-tuning BERT for token classification problems such as NER and POS, fine-tuning a language model for an NER problem, and thinking of the QA problem as a start/stop token classification.

The following topics will be covered in this chapter:

- Introduction to token classification

- Fine-tuning language models for NER

- Question answering using token classification

Technical requirements

We will be using Jupyter Notebook to run our coding exercises and Python 3.6+ and the following packages need to be installed:

```
sklearn
```

```
transformers 4.0+
```

All notebooks with coding exercises will be available at the following GitHub link: <https://github.com/PacktPublishing/Mastering-Transformers/tree/main/CH06>.

Check out the following link to see the Code in Action video: <https://bit.ly/2UGMQP2>

Introduction to token classification

The task of classifying each token in a token sequence is called **token classification**. This task says that a specific model must be able to classify each token into a class. POS and NER are two of the most well-known tasks in this criterion. However, QA is also another major NLP task that fits in this category. We will discuss the basics of these three tasks in the following sections.

Understanding NER

One of the well-known tasks in the category of token classification is NER – the recognition of each token as an entity or not and identifying the type of each detected entity. For example, a text can contain multiple entities at the same time – person names, locations, organizations, and other types of entities. The following text is a clear example of NER:

George Washington is one the presidents of the United States of America.

George Washington is a person name while *the United States of America* is a location name. A sequence tagging model is expected to tag each word in the form of tags, each containing information about the tag. BIO's tags are the ones that are universally used for standard NER tasks.

The following table is a list of tags and their descriptions:

Tag	Description
O	Out of entity
B-PER	Beginning of Person entity
I-PER	Inside of Person entity
B-LOC	Beginning of Location entity
I-LOC	Inside of Location entity
B-ORG	Beginning of Organization entity
I-ORG	Inside of Organization entity
B-MISC	Beginning of Miscellaneous entity
I-MISC	Inside of Miscellaneous entity

Table 1 – Table of BIOS tags and their descriptions

From this table, **B** indicates the beginning of a tag, and **I** denotes the inside of a tag, while **O** is the outside of the entity. This is the reason that this type of annotation is called **BIO**. For example, the sentence shown earlier can be annotated using BIO:

Copy

Explain

[B-PER|George] [I-PER|Washington] [O|is] [O|one] [O|the] [O|presidents] [O|of] [B-LOC|United] [I-LOC|States] [I-LOC|of] [I-LOC|America] [O|.]

Accordingly, the sequence must be tagged in BIO format. A sample dataset can be in the format shown as follows:

```
Soccer NN B-NP 0
- : 0 0
Japan NNP B-NP B-LOC
Get VB B-VP 0
Lucky NNP B-NP 0
Win NNP I-NP 0
, , 0 0
China NNP B-NP B-PER
In IN B-PP 0
Surprise DT B-NP 0
Defeat NN I-NP 0
. . 0 0

Nadim NNP B-NP B-PER
Ladki NNP I-NP I-PER

AL-AIN NNP B-NP B-LOC
, , 0 0
United NNP B-NP B-LOC
Arab NNP I-NP I-LOC
Emirates NNPS I-NP I-LOC
1996-12-06 CD I-NP 0
```

Figure 6.1 – CONLL2003 dataset

In addition to the NER tags we have seen, there are POS tags available in this dataset

Understanding POS tagging

POS tagging, or grammar tagging, is annotating a word in a given text according to its respective part of speech. As a simple example, in a given text, identification of each word's role in the categories of noun, adjective, adverb, and verb is considered to be POS. However, from a linguistic perspective, there are many roles other than these four.

In the case of POS tags, there are variations, but the Penn Treebank POS tagset is one of the most well-known ones. The following screenshot shows a summary and respective description of these roles:

1. CC	Coordinating conjunction	25. TO	to
2. CD	Cardinal number	26. UH	Interjection
3. DT	Determiner	27. VB	Verb, base form
4. EX	Existential <i>there</i>	28. VBD	Verb, past tense
5. FW	Foreign word	29. VBG	Verb, gerund/present participle
6. IN	Preposition/subordinating conjunction	30. VBN	Verb, past participle
7. JJ	Adjective	31. VBP	Verb, non-3rd ps. sing. present
8. JJR	Adjective, comparative	32. VBZ	Verb, 3rd ps. sing. present
9. JJS	Adjective, superlative	33. WDT	<i>wh</i> -determiner
10. LS	List item marker	34. WP	<i>wh</i> -pronoun
11. MD	Modal	35. WP\$	Possessive <i>wh</i> -pronoun
12. NN	Noun, singular or mass	36. WRB	<i>wh</i> -adverb
13. NNS	Noun, plural	37. #	Pound sign
14. NNP	Proper noun, singular	38. \$	Dollar sign
15. NNPS	Proper noun, plural	39. .	Sentence-final punctuation
16. PDT	Predeterminer	40. ,	Comma
17. POS	Possessive ending	41. :	Colon, semi-colon
18. PRP	Personal pronoun	42. (Left bracket character
19. PP\$	Possessive pronoun	43.)	Right bracket character
20. RB	Adverb	44. "	Straight double quote
21. RBR	Adverb, comparative	45. '	Left open single quote
22. RBS	Adverb, superlative	46. "	Left open double quote
23. RP	Particle	47. '	Right close single quote
24. SYM	Symbol (mathematical or scientific)	48. "	Right close double quote

Figure 6.2 – Penn Treebank POS tags

Datasets for POS tasks are annotated like the example shown in *Figure 6.1*.

The annotation of these tags is very useful in specific NLP applications and is one of the building blocks of many other methods. Transformers and many advanced models can somehow understand the relation of words in their complex architecture.

Understanding QA

A QA or reading comprehension task comprises a set of reading comprehension texts with respective questions on them. An exemplary dataset from this scope is **SQUAD** or **Stanford Question Answering Dataset**. This dataset consists of Wikipedia texts and respective questions asked about them. The answers are in the form of segments of the original Wikipedia text.

The following screenshot shows an example of this dataset:

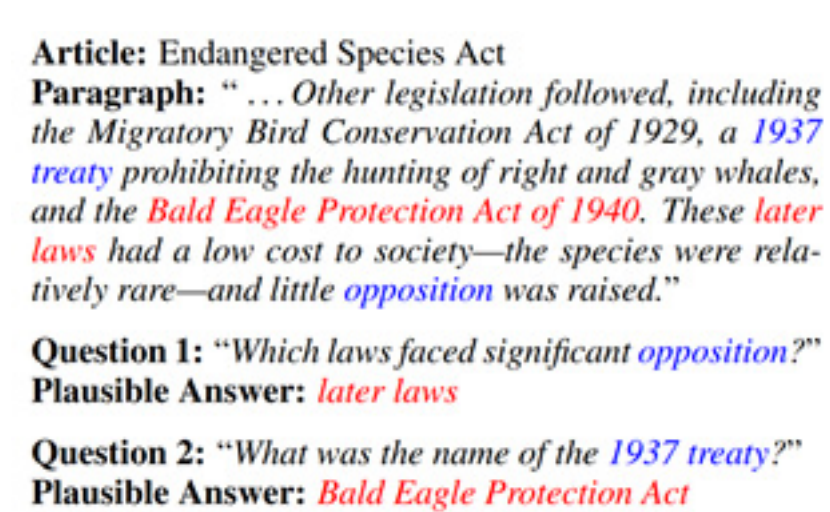


Figure 6.3 – SQUAD dataset example

The highlighted red segments are the answers and important parts of each question are highlighted in blue. It is required for a good NLP model to segment text according to the question, and this segmentation can be done in the form of sequence labeling.

The model labels the start and the end of the segment as answer start and end segments.

Up to this point, you have learned the basics of modern NLP sequence tagging tasks such as QA, NER, and POS. In the next section, you will learn how it is possible to fine-tune BERT for these specific tasks and use the related datasets from the `datasets` library.

Fine-tuning language models for NER

In this section, we will learn how to fine-tune BERT for an NER task. We first start with the `datasets` library and by loading the `conll2003` dataset.

The dataset card is accessible at <https://huggingface.co/datasets/conll2003>. The following screenshot shows this model card from the HuggingFace website:

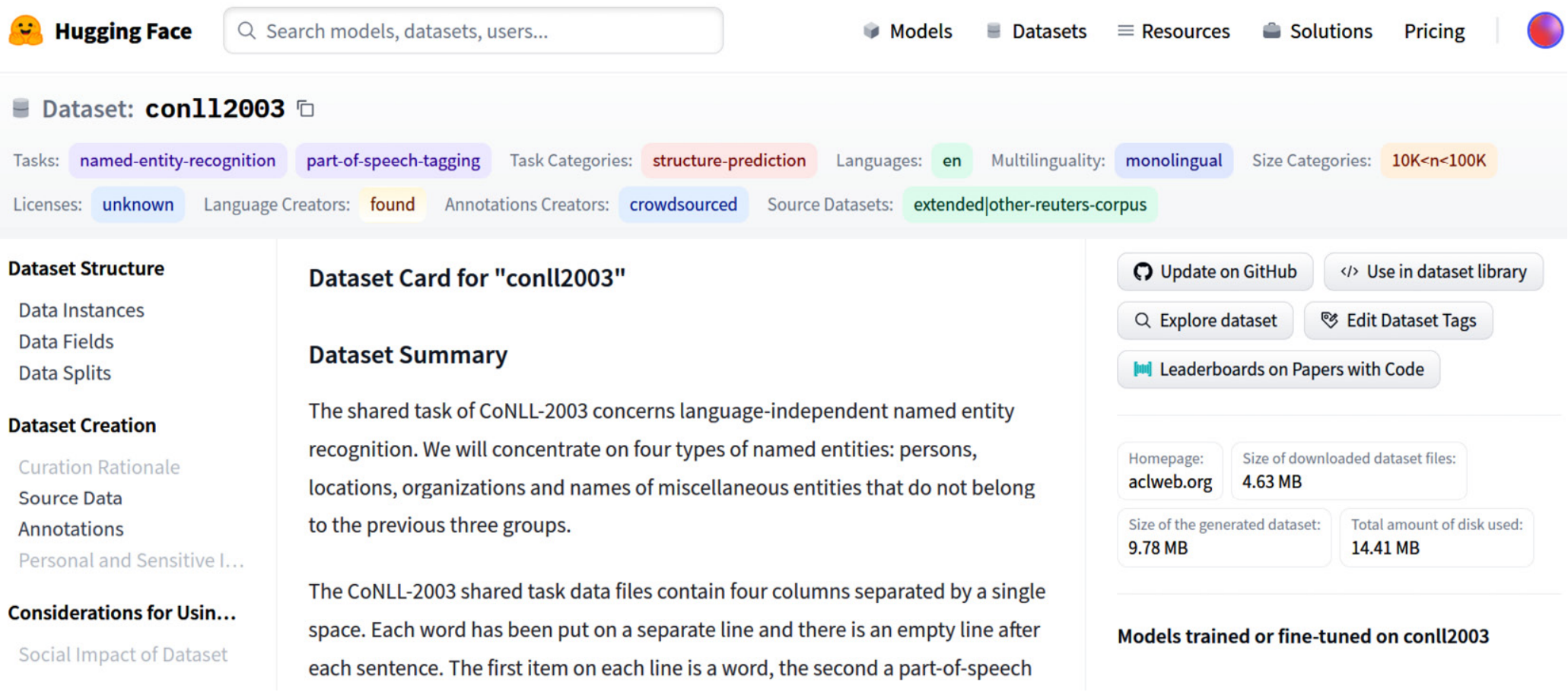


Figure 6.4 – CONLL2003 dataset card from HuggingFace

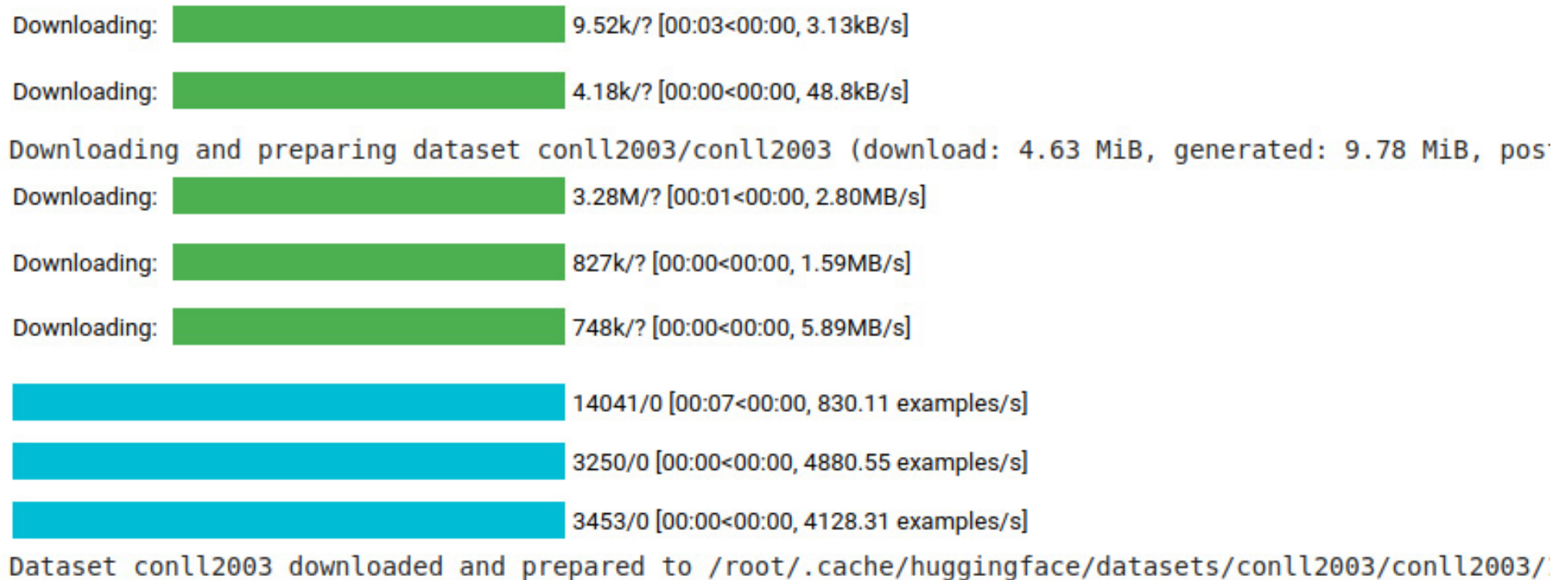
From this screenshot, it can be seen that the model is trained on this dataset and is currently available and listed in the right panel. However, there are also descriptions of the dataset such as its size and its characteristics:

1. To load the dataset, the following commands are used:

[Copy](#)[Explain](#)

```
import datasets
conll2003 = datasets.load_dataset("conll2003")
```

A download progress bar will appear and after finishing the downloading and caching, the dataset will be ready to use. The following screenshot shows the progress bars:



```
Downloading: ██████████ 9.52k/? [00:03<00:00, 3.13kB/s]
Downloading: ██████████ 4.18k/? [00:00<00:00, 48.8kB/s]
Downloading and preparing dataset conll2003/conll2003 (download: 4.63 MiB, generated: 9.78 MiB, pos
Downloading: ██████████ 3.28M/? [00:01<00:00, 2.80MB/s]
Downloading: ██████████ 827k/? [00:00<00:00, 1.59MB/s]
Downloading: ██████████ 748k/? [00:00<00:00, 5.89MB/s]

██████████ 14041/0 [00:07<00:00, 830.11 examples/s]
██████████ 3250/0 [00:00<00:00, 4880.55 examples/s]
██████████ 3453/0 [00:00<00:00, 4128.31 examples/s]
Dataset conll2003 downloaded and prepared to /root/.cache/huggingface/datasets/conll2003/conll2003/
```

Figure 6.5 – Downloading and preparing the dataset

2. You can easily double-check the dataset by accessing the train samples using the following command:

[Copy](#)[Explain](#)

```
>>> conll2003["train"][0]
```

The following screenshot shows the result:

```
{'chunk_tags': [11, 21, 11, 12, 21, 22, 11, 12, 0],
 'id': '0',
 'ner_tags': [3, 0, 7, 0, 0, 0, 7, 0, 0],
 'pos_tags': [22, 42, 16, 21, 35, 37, 16, 21, 7],
 'tokens': ['EU',
 'rejects',
 'German',
 'call',
 'to',
 'boycott',
 'British',
 'lamb',
 '.']}]
```

Figure 6.6 – CONLL2003 train samples from the datasets library

3. The respective tags for POS and NER are shown in the preceding screenshot. We will use only NER tags for this part. You can use the following command to get the NER tags available in this dataset:

[Copy](#)[Explain](#)

```
>>> conll2003["train"].features["ner_tags"]
```

4. The result is also shown in *Figure 6.7*. All the BIO tags are shown and there are nine tags in total:

[Copy](#)[Explain](#)

```
>>> Sequence(feature=ClassLabel(num_classes=9, names=['O', 'B-PER', 'I-PER', 'B-ORG', 'I-ORG', 'B-LOC', 'I-LOC', 'B-MISC', 'I-MISC'], names_file=None, id=None), length=-1, id=None)
```

5. The next step is to load the BERT tokenizer:

[Copy](#)[Explain](#)

```
from transformers import BertTokenizerFast
tokenizer = BertTokenizerFast.from_pretrained("bert-base-uncased")
```

6. The `tokenizer` class can work with white-space tokenized sentences also. We need to enable our tokenizer for working with white-space tokenized sentences, because the NER task has a token-based label for each token. Tokens in this task are usually the white-space tokenized words rather than BPE or any other tokenizer tokens. According to what is said, let's see how `tokenizer` can be used with a white-space tokenized sentence:

[Copy](#)[Explain](#)

```
>>> tokenizer(["Oh","this","sentence","is","tokenized","and",
"splitted","by","spaces"], is_split_into_words=True)
```

As you can see, by just setting `is_split_into_words` to `True`, the problem is solved.

7. It is required to preprocess the data before using it for training. To do so, we must use the following function and map into the entire dataset:

Explain

```
def tokenize_and_align_labels(examples):
    tokenized_inputs = tokenizer(examples["tokens"],
                                  truncation=True, is_split_into_words=True)
    labels = []
    for i, label in enumerate(examples["ner_tags"]):
        word_ids = \
            tokenized_inputs.word_ids(batch_index=i)
        previous_word_idx = None
        label_ids = []
        for word_idx in word_ids:
            if word_idx is None:
                label_ids.append(-100)
            elif word_idx != previous_word_idx:
                label_ids.append(label[word_idx])
            else:
                label_ids.append(label[word_idx] if label_all_tokens else -100)
            previous_word_idx = word_idx
        labels.append(label_ids)
    tokenized_inputs["labels"] = labels
    return tokenized_inputs
```

8. This function will make sure that our tokens and labels are aligned properly. This alignment is required because the tokens are tokenized in pieces, but the words must be of one piece. To test and see how this function works, you can run it by giving a single sample to it:

Explain

```
q = tokenize_and_align_labels(conll2003['train'][4:5])
print(q)
```

And the result is shown as follows:

Explain

```
>>> {'input_ids': [[101, 2762, 1005, 1055, 4387, 2000, 1996, 2647, 2586, 1005,
1055, 15651, 2837, 14121, 1062, 9328, 5804, 2056, 2006, 9317, 10390, 2323, 4965,
8351, 4168, 4017, 2013, 3032, 2060, 2084, 3725, 2127, 1996, 4045, 6040, 2001,
24509, 1012, 102]], 'token_type_ids': [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 'labels': [[-100, 5,
0, -100, 0, 0, 0, 3, 4, 0, -100, 0, 0, 1, 2, -100, -100, 0, 0, 0, 0, 0, 0, 0,
-100, -100, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, -100]]}
```

9. But this result is not readable, so you can run the following code to have a readable version:

[Copy](#)[Explain](#)

```
for token, label in zip(tokenizer.convert_ids_to_tokens(q["input_ids"]
[0]),q["labels"][0]):
    print(f"{token:_<40} {label}")
```

The result is shown as follows:

```
[CLS] _____ -100
germany _____ 5
' _____ 0
s _____ -100
representative _____ 0
to _____ 0
the _____ 0
european _____ 3
union _____ 4
' _____ 0
s _____ -100
veterinary _____ 0
committee _____ 0
werner _____ 1
z _____ 2
##wing _____ -100
##mann _____ -100
said _____ 0
on _____ 0
wednesday _____ 0
consumers _____ 0
should _____ 0
buy _____ 0
sheep _____ 0
##me _____ -100
##at _____ -100
from _____ 0
countries _____ 0
other _____ 0
than _____ 0
britain _____ 5
until _____ 0
the _____ 0
scientific _____ 0
advice _____ 0
was _____ 0
clearer _____ 0
' _____ 0
[SEP] _____ -100
```

Figure 6.7 – Result of the tokenize and align functions

10. The mapping of this function to the dataset can be done by using the **map** function of the **datasets** library:

[Copy](#)[Explain](#)

```
>>> tokenized_datasets = \ conll2003.map(tokenize_and_align_labels, batched=True)
```

11. In the next step, it is required to load the BERT model with the respective number of labels:

[Copy](#)[Explain](#)

```
from transformers import\ AutoModelForTokenClassification
model = AutoModelForTokenClassification.from_pretrained("bert-base-uncased",
num_labels=9)
```

12. The model will be loaded and ready to be trained. In the next step, we must prepare the trainer and training parameters:

[Copy](#)[Explain](#)

```
from transformers import TrainingArguments, Trainer
args = TrainingArguments(
    "test-ner",
    evaluation_strategy = "epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
)
```

13. It is required to prepare the data collator. It will apply batch operations on the training dataset to use less memory and perform faster. You can do so as follows:

[Copy](#)[Explain](#)

```
from transformers import \ DataCollatorForTokenClassification
data_collator = \ DataCollatorForTokenClassification(tokenizer)
```

14. To be able to evaluate model performance, there are many metrics available for many tasks in HuggingFace's **datasets** library. We will be using the sequence evaluation metric for NER. **seqeval** is a good Python framework to evaluate sequence tagging algorithms and models. It is necessary to install the **seqeval** library:

[Copy](#)[Explain](#)

```
pip install seqeval
```

15. Afterward, you can load the metric:

[Copy](#)[Explain](#)

```
>>> metric = datasets.load_metric("seqeval")
```

16. It is easily possible to see how the metric works by using the following code:

[Copy](#)[Explain](#)

```
example = conll2003['train'][0]
label_list = \ conll2003["train"].features["ner_tags"].feature.names
labels = [label_list[i] for i in example["ner_tags"]]
metric.compute(predictions=[labels], references=[labels])
```

The result is as follows:

```
{'MISC': {'f1': 1.0, 'number': 2, 'precision': 1.0, 'recall': 1.0},
'ORG': {'f1': 1.0, 'number': 1, 'precision': 1.0, 'recall': 1.0},
'overall_accuracy': 1.0,
'overall_f1': 1.0,
'overall_precision': 1.0,
'overall_recall': 1.0}
```

Figure 6.8 – Output of the seqeval metric

Various metrics such as accuracy, F1-score, precision, and recall are computed for the sample input.

17. The following function is used to compute the metrics:

```
import numpy as np def compute_metrics(p):
    predictions, labels = p
    predictions = np.argmax(predictions, axis=2)
    true_predictions = [
        [label_list[p] for (p, l) in zip(prediction, label) if l != -100]
        for prediction, label in zip(predictions, labels)
    ]
    true_labels = [
        [label_list[l] for (p, l) in zip(prediction, label) if l != -100]
        for prediction, label in zip(predictions, labels)
    ]
    results = \
        metric.compute(predictions=true_predictions,
            references=true_labels)
    return {
        "precision": results["overall_precision"],
        "recall": results["overall_recall"],
        "f1": results["overall_f1"],
        "accuracy": results["overall_accuracy"],
    }
```

Copy

Explain

18. The last steps are to make a trainer and train it accordingly:

```
trainer = Trainer(
    model,
    args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["validation"],
    data_collator=data_collator,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics
)
trainer.train()
```

Copy

Explain

19. After running the `train` function of `trainer`, the result will be as follows:

Epoch	Training Loss	Validation Loss	Precision	Recall	F1	Accuracy	Runtime	Samples Per Second
1	0.035800	0.043440	0.937072	0.944800	0.940920	0.988454	17.061700	190.486000
2	0.019100	0.043591	0.939359	0.951531	0.945406	0.989311	16.797100	193.486000
3	0.014500	0.043591	0.939359	0.951531	0.945406	0.989311	16.790100	193.567000

Figure 6.9 – Trainer results after running train

20. It is necessary to save the model and tokenizer after training:

```
model.save_pretrained("ner_model")
tokenizer.save_pretrained("tokenizer")
```

[Copy](#)
[Explain](#)

21. If you wish to use the model with the pipeline, you must read the config file and assign `label2id` and `id2label` correctly according to the labels you have used in the `label_list` object:

```
id2label = {
    str(i): label for i, label in enumerate(label_list)
}
label2id = {
    label: str(i) for i, label in enumerate(label_list)
}
import json
config = json.load(open("ner_model/config.json"))
config["id2label"] = id2label
config["label2id"] = label2id
json.dump(config, open("ner_model/config.json", "w"))
```

[Copy](#)
[Explain](#)

22. Afterward, it is easy to use the model as in the following example:

```
from transformers import pipeline
model = \ AutoModelForTokenClassification.from_pretrained("ner_model")
nlp = \
    pipeline("ner", model=model, tokenizer=tokenizer)
example = "I live in Istanbul"
ner_results = nlp(example)
print(ner_results)
```

[Copy](#)
[Explain](#)

And the result will appear as seen here:

[Copy](#)[Explain](#)

```
[{'entity': 'B-L0C', 'score': 0.9983942, 'index': 4, 'word': 'istanbul', 'start': 10, 'end': 18}]
```

Up to this point, you have learned how to apply POS using BERT. You learned how to train your own POS tagging model using Transformers and you also tested the model. In the next section, we will focus on QA.

Question answering using token classification

A **QA** problem is generally defined as an NLP problem with a given text and a question for AI, and getting an answer back. Usually, this answer can be found in the original text but there are different approaches to this problem. In the case of **Visual Question Answering (VQA)**, the question is about a visual entity or visual concept rather than text but the question itself is in the form of text.

Some examples of VQA are as follows:

Who is wearing glasses?

man



woman

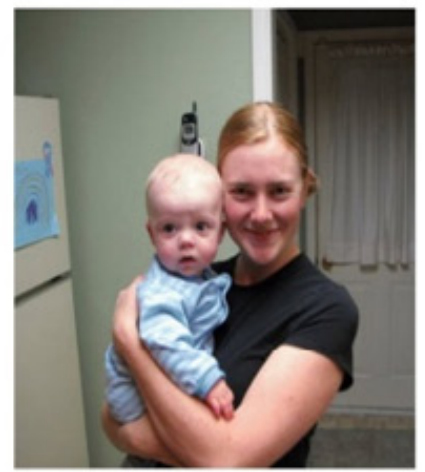


Where is the child sitting?

fridge



arms



Is the umbrella upside down?

yes



no



How many children are in the bed?

2



1

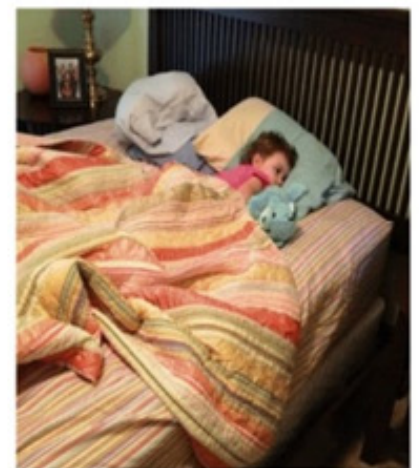


Figure 6.10 – VQA examples

Most of the models that are intended to be used in VQA are multimodal models that can understand the visual context along with the question and generate the answer properly. However, unimodal fully textual QA or just QA is based on textual context and textual questions with respective textual answers:

1. SQUAD is one of the most well-known datasets in the field of QA. To see examples of SQUAD and examine them, you can use the following code:

```
from pprint import pprint
from datasets import load_dataset
squad = load_dataset("squad")
for item in squad["train"][1].items():
    print(item[0])
    pprint(item[1])
    print("="*20)
```

Copy

Explain

The following is the result:

[Copy](#)[Explain](#)

```
answers
{'answer_start': [188], 'text': ['a copper statue of Christ']}
=====
Context
('Architecturally, the school has a Catholic character. Atop the Main '
'Building's gold dome is a golden statue of the Virgin Mary. Immediately in '
'front of the Main Building and facing it, is a copper statue of Christ with '
'arms upraised with the legend "Venite Ad Me Omnes". Next to the Main ' 'Building
is the Basilica of the Sacred Heart. Immediately behind the ' 'basilica is the
Grotto, a Marian place of prayer and reflection. It is a ' 'replica of the grotto
at Lourdes, France where the Virgin Mary reputedly ' 'appeared to Saint
Bernadette Soubirous in 1858. At the end of the main drive ' '(and in a direct
line that connects through 3 statues and the Gold Dome), is ' 'a simple, modern
stone statue of Mary.')
=====
Id
'5733be284776f4190066117f'
=====
Question
'What is in front of the Notre Dame Main Building?'
=====
Title
'University_of_Notre_Dame'
=====
```

However, there is version 2 of the SQUAD dataset, which has more training samples, and it is highly recommended to use it. To have an overall understanding of how it is possible to train a model for a QA problem, we will focus on the current part of this problem.

2. To start, load SQUAD version 2 using the following code:

[Copy](#)[Explain](#)

```
from datasets import load_dataset
squad = load_dataset("squad_v2")
```

3. After loading the SQUAD dataset, you can see the details of this dataset by using the following code:

[Copy](#)[Explain](#)

```
>>> squad
```

The result is as follows:


```
DatasetDict({
  train: Dataset({
    features: ['id', 'title', 'context', 'question', 'answers'],
    num_rows: 130319
  })
  validation: Dataset({
    features: ['id', 'title', 'context', 'question', 'answers'],
    num_rows: 11873
  })
})
```

Figure 6.11 – SQUAD dataset (version 2) details

The details of the SQUAD dataset will be shown as seen in *Figure 6.11*. As you can see, there are more than 130,000 training samples with more than 11,000 validation samples.

4. As we did for NER, we must preprocess the data to have the right form to be used by the model. To do so, you must first load your tokenizer, which is a pretrained tokenizer as long as you are using a pretrained model and want to fine-tune it for a QA problem:

```
from transformers import AutoTokenizer
model = "distilbert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model)
```

Copy

Explain

As you have seen, we are going to use the **distilBERT** model.

According to our SQUAD example, we need to give more than one text to the model, one for the question and one for the context. Accordingly, we need our tokenizer to put these two side by side and separate them with the special **[SEP]** token because **distilBERT** is a BERT-based model.

There is another problem in the scope of QA, and it is the size of the context. The context size can be longer than the model input size, but we cannot reduce it to the size the model accepts. With some problems, we can do so but in QA, it is possible that the answer could be in the truncated part. We will show you an example where we tackle this problem using document stride.

5. The following is an example to show how it works using **tokenizer**:

[Copy](#)[Explain](#)

```
max_length = 384
doc_stride = 128
example = squad["train"][173]
tokenized_example = tokenizer(
    example["question"],
    example["context"],
    max_length=max_length,
    truncation="only_second",
    return_overflowing_tokens=True,
    stride=doc_stride
)
```

6. The stride is the document stride used to return the stride for the second part, like a window, while the `return_overflowing_tokens` flag gives the model information on whether it should return the extra tokens. The result of `tokenized_example` is more than a single tokenized output, instead having two input IDs. In the following, you can see the result:

[Copy](#)[Explain](#)

```
>>> len(tokenized_example['input_ids'])
>>> 2
```

7. Accordingly, you can see the full result by running the following `for` loop:

[Copy](#)[Explain](#)

```
for input_ids in tokenized_example["input_ids"][:2]:
    print(tokenizer.decode(input_ids))
    print("-"*50)
```

The result is as follows:

[CLS] beyonce got married in 2008 to whom? [SEP] on april 4, 2008, beyonce married jay z. she publicly revealed their marriage in a video montage at the listening party for her third studio album, i am... sasha fierce, in manhattan's sony club on october 22, 2008. i am... sasha fierce was released on november 18, 2008 in the united states. the album formally introduces beyonce's alter ego sasha fierce, conceived during the making of her 2003 single " crazy in love ", selling 482, 000 copies in its first week, debuting atop the billboard 200, and giving beyonce her third consecutive number – one album in the us. the album featured the number – one song " single ladies (put a ring on it) " and the top – five songs " if i were a boy " and " halo ". achieving the accomplishment of becoming her longest – running hot 100 single in her career, " halo "'s success in the us helped beyonce attain more top – ten singles on the list than any other woman during the 2000s. it also included the successful " sweet dreams ", and singles " diva ", " ego ", " broken – hearted girl " and " video phone ". the music video for " single ladies " has been parodied and imitated around the world, spawning the " first major dance craze " of the internet age according to the toronto star. the video has won several awards, including best video at the 2009 mtv europe music awards, the 2009 scottish mobo awards, and the 2009 bet awards. at the 2009 mtv video music awards, the video was nominated for nine awards, ultimately winning three including video of the year. its failure to win the best female video category, which went to american country pop singer taylor swift's " you belong with me ", led to kanye west interrupting the ceremony and beyonce [SEP]

[CLS] beyonce got married in 2008 to whom? [SEP] single ladies " has been parodied and imitated around the world, spawning the " first major dance craze " of the internet age according to the toronto star. the video has won several awards, including best video at the 2009 mtv europe music awards, the 2009 scottish mobo awards, and the 2009 bet awards. at the 2009 mtv video music awards, the video was nominated for nine awards, ultimately winning three including video of the year. its failure to win the best female video category, which went to american country pop singer taylor swift's " you belong with me ", led to kanye west interrupting the ceremony and beyonce improvising a re – presentation of swift's award during her own acceptance speech. in march 2009, beyonce embarked on the i am... world tour, her second headlining worldwide concert tour, consisting of 108 shows, grossing \$ 119. 5 million. [SEP]

As you can see from the preceding output, with a window of 128 tokens, the rest of the context is replicated again in the second output of input IDs.

Another problem is the end span, which is not available in the dataset, but instead, the start span or the start character for the answer is given. It is easy to find the length of the answer and add it to the start span, which would automatically yield the end span.

8. Now that we know all the details of this dataset and how to deal with them, we can easily put them together to make a preprocessing function (link:

https://github.com/huggingface/transformers/blob/master/examples/pytorch/question-answering/run_qa.py):

`run_qa.py`


```

def prepare_train_features(examples):
    # tokenize examples
    tokenized_examples = tokenizer(
        examples["question" if pad_on_right else "context"],
        examples["context" if pad_on_right else "question"],
        truncation="only_second" if pad_on_right else "only_first",
        max_length=max_length,
        stride=doc_stride,
        return_overflowing_tokens=True,
        return_offsets_mapping=True,
        padding="max_length",
    )
    # map from a feature to its example
    sample_mapping = \ tokenized_examples.pop("overflow_to_sample_mapping")
    offset_mapping = \ tokenized_examples.pop("offset_mapping")
    tokenized_examples["start_positions"] = []
    tokenized_examples["end_positions"] = []
    # label impossible answers with CLS
    # start and end token are the answers for each one
    for i, offsets in enumerate(offset_mapping):
        input_ids = tokenized_examples["input_ids"][i]
        cls_index = \ input_ids.index(tokenizer.cls_token_id)
        sequence_ids = \ tokenized_examples.sequence_ids(i)
        sample_index = sample_mapping[i]
        answers = examples["answers"][sample_index]
        if len(answers["answer_start"]) == 0:
            tokenized_examples["start_positions"].\ append(cls_index)
            tokenized_examples["end_positions"].\ append(cls_index)
        else:
            start_char = answers["answer_start"][0]
            end_char = \
                start_char + len(answers["text"][0])
            token_start_index = 0
            while sequence_ids[token_start_index] != (1 if pad_on_right else 0):
                token_start_index += 1
            token_end_index = len(input_ids) - 1
            while sequence_ids[token_end_index] != (1 if pad_on_right else 0):
                token_end_index -= 1
            if not (offsets[token_start_index][0] <= start_char and
offsets[token_end_index][1] >= end_char):
                tokenized_examples["start_positions"].append(cls_index)
                tokenized_examples["end_positions"].append(cls_index)
            else:
                while token_start_index < len(offsets) and
offsets[token_start_index][0] <= start_char:
                    token_start_index += 1
                tokenized_examples["start_positions"].append(token_start_index -
1)

                while offsets[token_end_index][1] >= end_char:
                    token_end_index -= 1
                tokenized_examples["end_positions"].append(token_end_index + 1)

    return tokenized_examples

```

9. Mapping this function to the dataset would apply all the required changes:

[Copy](#)[Explain](#)

```
>>> tokenized_datasets = squad.map(prepare_train_features, batched=True,
remove_columns=squad["train"].column_names)
```

10. Just like other examples, you can now load a pretrained model to be fine-tuned

[Copy](#)[Explain](#)

```
from transformers import AutoModelForQuestionAnswering, TrainingArguments, Trainer
model = AutoModelForQuestionAnswering.from_pretrained(model)
```

11. The next step is to create training arguments:

[Copy](#)[Explain](#)

```
args = TrainingArguments(
    "test-squad",
    evaluation_strategy = "epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
)
```

12. If we are not going to use a data collator, we will give a default data collator to the model trainer:

[Copy](#)[Explain](#)

```
from transformers import default_data_collator
data_collator = default_data_collator
```

13. Now, everything is ready to make the trainer:

[Copy](#)[Explain](#)

```
trainer = Trainer(
    model,
    args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["validation"],
    data_collator=data_collator,
    tokenizer=tokenizer,
)
```

14. And the trainer can be used with the `train` function:

Copy Explain

```
trainer.train()
```

The result will be something like the following:

[16599/16599 58:06, Epoch 3/3]

Epoch	Training Loss	Validation Loss	Runtime	Samples Per Second
1	1.220600	1.160322	39.574900	272.496000
2	0.945200	1.121690	39.706000	271.596000
3	0.773000	1.157358	39.734000	271.405000

Figure 6.12 – Training results

As you can see, the model is trained with three epochs and the outputs for loss in validation and training are reported.

15. Like any other model, you can easily save this model by using the following function:

Copy Explain

```
>>> trainer.save_model("distillBERT_SQUAD")
```

If you want to use your saved model or any other model that is trained on QA, the `transformers` library provides a pipeline that's easy to use and implement with no extra effort.

16. By using this pipeline functionality, you can use any model. The following is an example given for using a model with the QA pipeline:

Copy Explain

```
from transformers import pipeline
qa_model = pipeline('question-answering', model='distilbert-base-cased-distilled-squad', tokenizer='distilbert-base-cased')
```

The pipeline just requires two inputs to make the model ready for usage, the model and the tokenizer. Although, you are also required to give it a pipeline type, which is QA in the given example.

17. The next step is to give it the inputs it requires, `context` and `question`:

[Copy](#)[Explain](#)

```
>>> question = squad["validation"][0]["question"]
>>> context = squad["validation"][0]["context"]
The question and the context can be seen by using following code:
>>> print("Question:")
>>> print(question)
>>> print("Context:")
>>> print(context)
Question:
In what country is Normandy located?
Context:
('The Normans (Norman: Nourmands; French: Normands; Latin: Normanni) were the '
'people who in the 10th and 11th centuries gave their name to Normandy, a '
'region in France. They were descended from Norse ("Norman" comes from '
'"Norseman") raiders and pirates from Denmark, Iceland and Norway who, under '
'their leader Rollo, agreed to swear fealty to King Charles III of West '
'Francia. Through generations of assimilation and mixing with the native '
'Frankish and Roman-Gaulish populations, their descendants would gradually '
'merge with the Carolingian-based cultures of West Francia. The distinct '
'cultural and ethnic identity of the Normans emerged initially in the first '
'half of the 10th century, and it continued to evolve over the succeeding '
'centuries.')
```

18. The model can be used by the following example:

[Copy](#)[Explain](#)

```
>>> qa_model(question=question, context=context)
```

And the result can be seen as follows:

[Copy](#)[Explain](#)

```
{'answer': 'France', 'score': 0.9889379143714905, 'start': 159, 'end': 165,}
```

Up to this point, you have learned how you can train on the dataset you want. You have also learned how you can use the trained model using pipelines.

Summary

In this chapter, we discussed how to fine-tune a pretrained model to any token classification task. Fine-tuning models on NER and QA problems were explored. Using the pretrained and fine-tuned models on specific tasks with pipelines was detailed

with examples. We also learned about various preprocessing steps for these two tasks. Saving pretrained models that are fine-tuned on specific tasks was another major learning point of this chapter. We also saw how it is possible to train models with a limited input size on tasks such as QA that have longer sequence sizes than the model input. Using tokenizers more efficiently to have document splitting with document stride was another important item in this chapter too.

In the next chapter, we will discuss text representation methods using Transformers. By studying the chapter, you will learn how to perform zero-/few-shot learning and semantic text clustering.

[Previous Chapter](#)[Next Chapter](#)