14 Question answering with a fine-tuned large language model

This chapter covers

- Building a question-answering application using an LLM
- Curating a question-answering dataset for training
- Fine-tuning a Transformer-based LLM
- Integrating a deep-learning-based NLP pipeline to extract and rank answers from search results

We covered the basics of semantic search using Transformers in chapter 13, so we're now ready to attempt one of the hardest problems in search: question answering.

Question answering is the process of returning an answer for a searcher's query, rather than just a list of search results. There are two types of question-answering approaches: extractive and abstractive. *Extractive question answering* is the process of finding exact answers to questions from your documents. It returns snippets of your documents containing the likely answer to the user's question so they don't need to sift through search results. In contrast, *abstractive question answering* is the process of generating responses to a user's question either as a summary of multiple documents or directly from an LLM with no source documents. In this chapter, we'll focus primarily on extractive question answering, saving abstractive question answering for chapter 15.

By solving the question-answering problem, you will accomplish three things:

- You'll better understand the Transformer tooling and ecosystem that you started learning about in chapter 13.
- You'll learn how to fine-tune a large language model to a specific task.
- You'll merge your search engine with advanced natural language techniques.

In this chapter, we'll show you how to provide direct answers to questions and produce a working question-answering application. The query types we'll address are single clause *who*, *what*, *when*, *where*, *why*, and *how* questions. We'll also continue using the Stack Exchange outdoors dataset from the last chapter. Our goal is to enable users to ask a *previously unseen question* and get a short answer in response, eliminating the need for users to read through multiple search results to find the answer themselves.

14.1 Question-answering overview

Traditional search returns lists of documents or pages in response to a query, but people may often be looking for a quick answer to their question. In this case, we want to save people from having to dig for an answer in blocks of text when a straightforward one exists in our content.

In this section, we'll introduce the question-answering task and then define the retriever-reader pattern for implementing question-answering.

14.1.1 How a question-answering model works

Let's look at how a question-answering model works in practice. Specifically, we're implementing *extractive question answering*, which finds the best answer to a question in a given text. For example, take the following question:

Q: `What are minimalist shoes?`

Extractive question answering works by looking through a large document that probably contains the answer, and it identifies the answer for you.

Let's look at a document that might contain the answer to our question. We provide the question `What are minimalist shoes?` and the following document text (the context) to a model:

```
There was actually a project done on the definition of what a minimalist shoe
is and the result was "Footwear providing minimal interference with the
natural movement of the foot due to its high flexibility, low heel to toe
drop, weight and stack height, and the absence of motion control and stability
devices". If you are looking for a simpler definition, this is what Wikipedia
says, Minimalist shoes are shoes intended to closely approximate barefoot
running conditions. 1 They have reduced cushioning, thin soles, and are of
lighter weight than other running shoes, allowing for more sensory contact
for the foot on the ground while simultaneously providing the feet with some protec
look like this.
```

The document can be broken into many small parts, known as *spans*, and the model extracts the best span as the answer. A working question-answering model will evaluate the question and context and may produce this span as the answer:

A: `shoes intended to closely approximate barefoot running conditions`

But how does the model know the probability of whether any given span is the answer? We could try to look at different spans and see if they all somehow represent the answer, but that would be very complicated. Instead, the problem can be simplified by first learning the probability of whether each token in the context is the start of the answer and also the probability of whether each token in the context is the end of the answer. Since we are only looking at the probability for one token to represent the start, and another the end, the problem is easier to understand and solve. Our tokens are treated as discrete values, and the extractive question-answering model is trained to learn a *probability mass function* (PMF), which is a function that gives the probability that a discrete random variable is exactly equal to some value. This is different than measuring values that are continuous and are used in probability distributions, as we discussed with the continuous beta distribution in chapter 11. The main difference between the two is that our tokens are discrete values.

Using this strategy, we can train one model that will learn two probability mass functions—one for the starting token of the answer span, and one for the ending token of the answer span. You may have caught before that we said the model "*may* produce" the previous answer. Since models trained with different data and hyperparameters will give different results, the specific answer provided for a question can vary based on the model's training parameters.

To illustrate how this works, we'll start with a model that someone has already trained for the extractive question-answering task. The model will output the likelihood of whether the token is the start or the end of an answer span. When we identify the most likely start and end of the

answer, that's our answer span. The pretrained model we'll use is `deepset/roberta-base-squad2`, available from the Hugging Face organization and trained by the Deepset team. We will pass the question and context through this model and pipeline in listings 14.1 through 14.3 to determine the answer span start and end probabilities, as well as the final answer. Figure 14.1 demonstrates how this process works by *tokenizing* the question and context input, *encoding* that input, and *predicting* the most appropriate answer span.

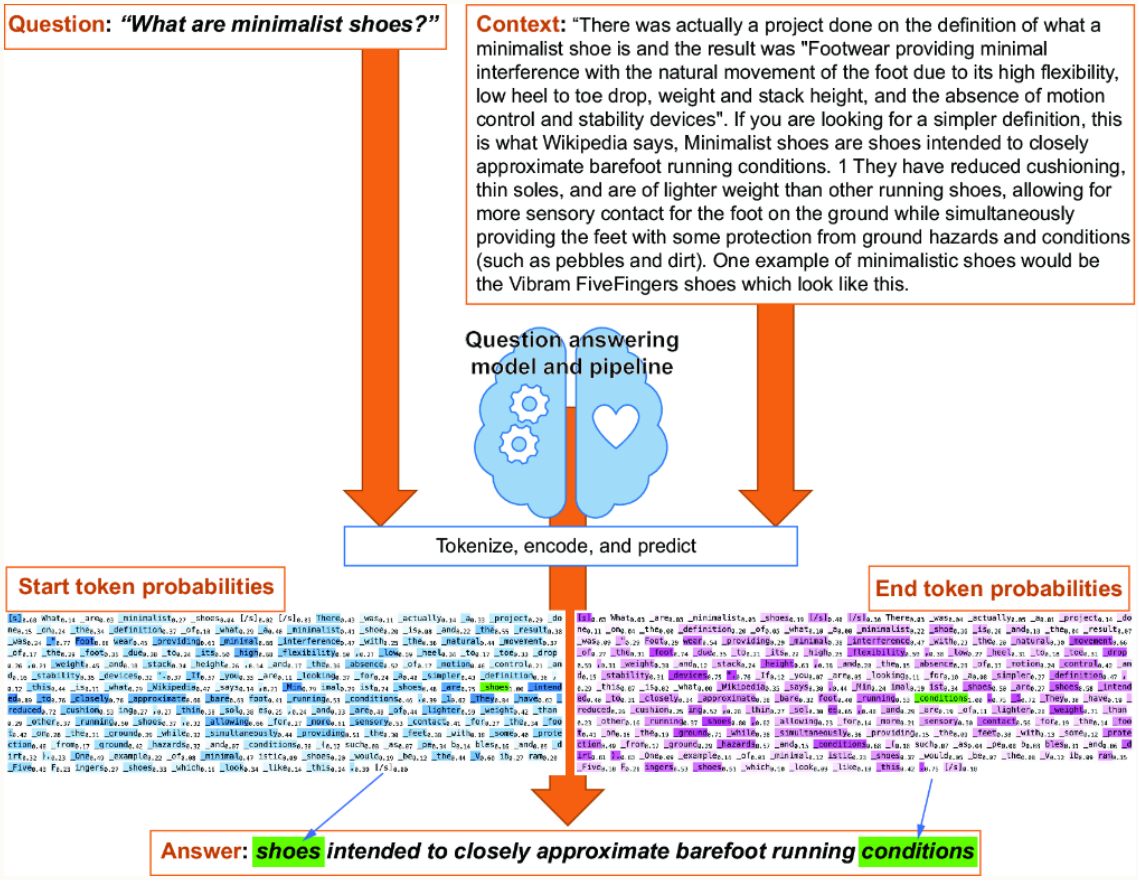

Figure 14.1 Extractive question-answering prediction process

In the figure, you can see that the question and context are first combined into a pair for tokenization. Tokenization is then performed on the pair, obtaining token inputs for the model. The model accepts those inputs and then outputs two sequences: the first sequence is the probabilities for whether each token in the context is the start of an answer, and the second sequence is the probabilities for whether each token in the context is the end of an answer. The start and end probability sequences are then combined to get the most likely answer span.

The following listing walks through the first step of this process: tokenization.

Listing 14.1 Loading the tokenizer and model

```
from transformers import AutoTokenizer, AutoModelForQuestionAnswering
model_name = "deepset/roberta-base-squad2"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForQuestionAnswering.from_pretrained(model_name)
```

The model name in listing 14.1 is a public model specifically pretrained for extractive question answering. With the model and tokenizer ready, we can now pass in a question and answer pair, shown in the following listing. The response will show that the number of tokens is equal to the number of start and end probabilities.

Listing 14.2 Tokenizing a question and context

```
question = "What are minimalist shoes"
context = """There was actually a project done on the definition of what a
minimalist shoe is and the result was "Footwear providing minimal
interference with the natural movement of the foot due to its high
flexibility, low heel to toe drop, weight and stack height, and the absence
of motion control and stability devices". If you are looking for a simpler
definition, this is what Wikipedia says, Minimalist shoes are shoes intended
to closely approximate barefoot running conditions. 1 They have reduced
cushioning, thin soles, and are of lighter weight than other running shoes,
allowing for more sensory contact for the foot on the ground while
simultaneously providing the feet with some protection from ground
hazards and conditions (such as pebbles and dirt).
One example of minimalistic shoes would be the Vibram FiveFingers
shoes which look like this."""

inputs = tokenizer(question, context, add_special_tokens=True,
                   return_tensors="pt")
input_ids = inputs["input_ids"].tolist()[0]

outputs = model(**inputs)
start_logits_norm = normalize(outputs[0].detach().numpy())
end_logits_norm = normalize(outputs[1].detach().numpy())

print(f"Total number of tokens: {len(input_ids)}")
print(f"Total number of start probabilities: {start_logits_norm.shape[1]}")
print(f"Total number of end probabilities: {end_logits_norm.shape[1]}")
```

Response:

```
Total number of tokens: 172
Total number of start probabilities: 172
Total number of end probabilities: 172
```

The inputs are obtained by tokenizing the question and context together. The outputs are obtained by making a forward pass with the inputs through the model. The `outputs` variable is a two-item list. The first item contains the start probabilities, and the second item contains the end probabilities.

Figure 14.2 visually demonstrates the probabilities for whether each token in the context is likely the start of an answer span, and figure 14.3 likewise demonstrates whether each token is likely the end of an answer span (darker highlighting indicates a higher probability).

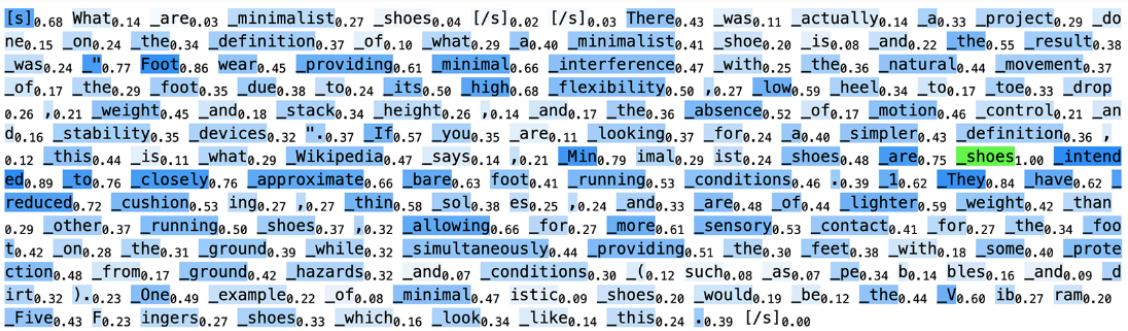

Figure 14.2 Probabilities for whether the token is the start of an answer span

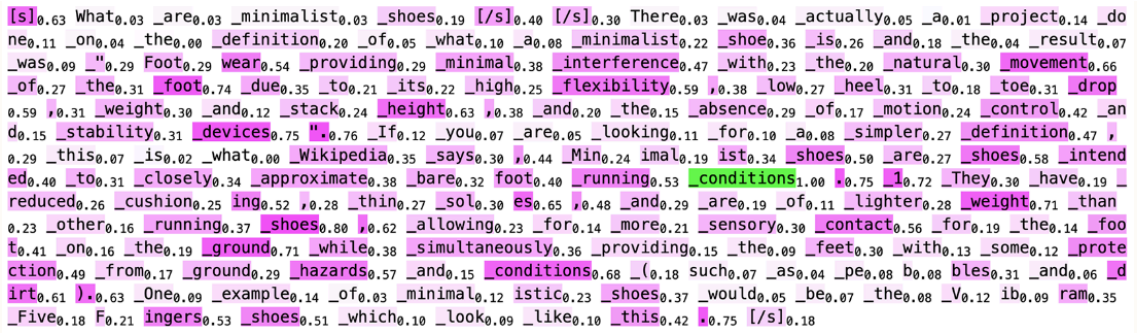

Figure 14.3 Probabilities for whether the token is the end of an answer span

Note that each token has a start probability (in figure 14.2) and an end probability (in figure 14.3) at its respective index. We also normalized the start and end probabilities at each index to be between `0.0` and `1.0`, which makes them easier to think about and calculate. We call these start and end probability lists *logits*, since they are lists of statistical probabilities.

For example, for the 17th token (`_definition`), the probability of this token being the start of the answer is ~`0.37`, and the probability of it being the end of the answer is ~`0.20`. Since we've normalized both lists, the start of our answer span is the token where `start_logits_norm == 1.0`, and the end of the answer span is where `end_logits_norm == 1.0`.

The following listing demonstrates both how to generate the token lists in figures 14.2 and 14.3, as well as how to extract the final answer span.

Listing 14.3 Identifying an answer span from a tokenized context

```
start_tokens = []
end_tokens = []
terms = tokenizer.convert_ids_to_tokens(input_ids)
start_token_id = 0
end_token_id = len(terms)
for i, term in enumerate(terms):
  start_tokens.append(stylize(term, [0, 127, 255], start_logits_norm[0][i]))
  end_tokens.append(stylize(term, [255, 0, 255], end_logits_norm[0][i]))
  if start_logits_norm[0][i] == 1.0:
    start_token_id = i
  if end_logits_norm[0][i] == 1.0:
    end_token_id = i + 1

answer = terms[start_token_id:end_token_id]
display(HTML(f'<h3>{clean_token(" ".join(answer))}</h3>'))  #1
display(HTML(f'<pre>{" ".join(start_tokens)}</pre>'))  #2
display(HTML(f'<pre>{" ".join(end_tokens)}</pre>'))  #3
```

#1 Extracting the answer span, shown in the following output
#2 The start probabilities, shown in figure 14.2
#3 The end probabilities, shown in figure 14.3

Output:

```
_shoes _intended _to _closely _approximate _bare foot _running _conditions
```

By fitting the start and end probability mass functions during training to a dataset of question, context, and answer triples, we create a model that can provide probabilities for the most likely answers to new questions and contexts. We then use this model in listing 14.3 to perform a probability search to identify the most likely span in the text that answers the question.

In practice, it works as follows:

1. We pick a minimum and maximum span size—where a span is a set of continuous words. For example, the answer might be one word long or, like the previous answer, it could be eight words long. We need to set those span sizes up front.
2. For each span, we check the probability of whether or not the span is the correct answer. The answer is the span with the highest probability.
3. When we're done checking all the spans, we present the correct answer.

Building the model requires lots of question/context/answer triples and a way to provide these triples to the model so that calculations can be performed. Enter the Transformer encoder, which you should already be familiar with from chapter 13. We first encode lots of training data using an LLM that produces dense vectors. Then we train a neural network to learn the probability mass function of whether a given span's encoding answers the question using positive and negative training examples.

We'll see how to fine-tune a question-answering model later in the chapter, but first we need to address a very important detail: When someone asks a question, where do we get the context?

14.1.2 The retriever-reader pattern

In reading about how extractive question answering works, you may have thought "so, for every question query, I need to check the probabilities of every span in the entire corpus?" No! That would be extremely slow and unnecessary, since we already have a really fast and accurate way of getting relevant documents that probably contain the answer: search.

What we're really going to make is a very powerful text highlighter. Think of the entire question-answering system as an automatic reference librarian of sorts. It knows what document contains your answer, and then it reads that document's text so it can point the exact answer out to you.

This is known as the retriever-reader pattern. This pattern uses one component to retrieve and rank the candidate documents (running a query against the search engine) and another component to read the spans of the most relevant documents and extract the appropriate answer. This is very similar to how highlighting works in Lucene-based search engines like Solr, OpenSearch, or Elasticsearch: the unified highlighter will find the best passage(s) containing the analyzed query terms and use them as the context. It then identifies the exact location of the queried keywords in that context to show the end user the surrounding context.

We're going to build something similar to a highlighter, but instead of showing the context containing the user's queried keywords, our question-answering highlighter will answer questions like this:

```
Q: What are minimalist shoes?
A: shoes intended to closely approximate barefoot running conditions
```

Let's look at the full context. When we ask `What are minimalist shoes?` we first use the *retriever* to get the document that is the most likely to contain the answer. In this case, this document (abridged here, but shown in full in section 14.1.1) was returned:

```
There was actually a project done on the definition... this is what Wikipedia
says, Minimalist shoes are shoes intended to closely approximate barefoot
running conditions. 1 They have reduced cushioning, thin soles, ...
```

With the document in hand, the *reader* then scans it and finds the text that is most likely to answer the question.

Aside from using fancy Transformers to find the correct answer, we're taking a step beyond basic search in that we'll actually use question/answer reader confidence as a reranker. So if we're not sure which document is the most likely to contain the answer during the retriever step, we'll have the reader look through a bunch of documents and see which is the best answer from all of them. The "bunch of documents" will be our reranking window, which we can set to any size.

Keep in mind, though, that analyzing documents in real time is not efficient. We shouldn't ask the reader to look at 100 documents—that's going to take too long. We'll limit it to a much smaller number, like 3 or 5. Limiting the reader window forces us to make sure our search engine is very accurate. The results must be relevant, as a retriever that doesn't get relevant candidates in the top 5 window size won't give the reader anything useful to work with.

The retriever-reader has two separated concerns, so we can even replace our retriever with something else. We've shown how you can use a lexical search engine with out-of-the-box ranking (BM25, as covered in chapter 3), but you can also try it with a dense vector index of embeddings, as covered in chapter 13.

Before we can take live user questions, we'll also need to train a question-answering model to predict the best answer from a context. The full set of steps we'll walk through for building our question-answering application is as follows:

1. *Set up a retriever using our search engine*—We'll use a simple high-recall query for candidate answers for our example.
2. *Wrangle and label the data appropriately*—This includes getting the data into the correct format, and getting a first pass at our answers from the base pretrained model. Then we'll take that first pass and manually fix it and mark what examples to use for training and testing.
3. *Understand the nuances of the data structures*—We'll use an existing data structure that will represent our training and test data in the correct format for a fine-tuning task.
4. *Fine-tune the model*—Using the corrections we made in the previous step, we'll train a fine-tuned question-answering model for better accuracy than the baseline.
5. *Use the model as the reader at query time*—We'll put everything together that lets us request a query, get candidates from the search engine, read/rerank the answers from the model, and show them as a response.

Figure 14.4 shows the flow of the entire architecture for our retriever, reader, and reranker:

1. A question is asked by a user, and documents are queried in the retriever (search engine) using the question.
2. The search engine matches and ranks to get the top-*k* most relevant documents for the reader.
3. The original question is paired with each top-*k* retrieved context and sent into the QA pipeline.
4. The question/context pairs are tokenized and encoded into spans by the reader, which then predicts the top-*n* most likely answer spans, with their probabilities for likeliness as the score.

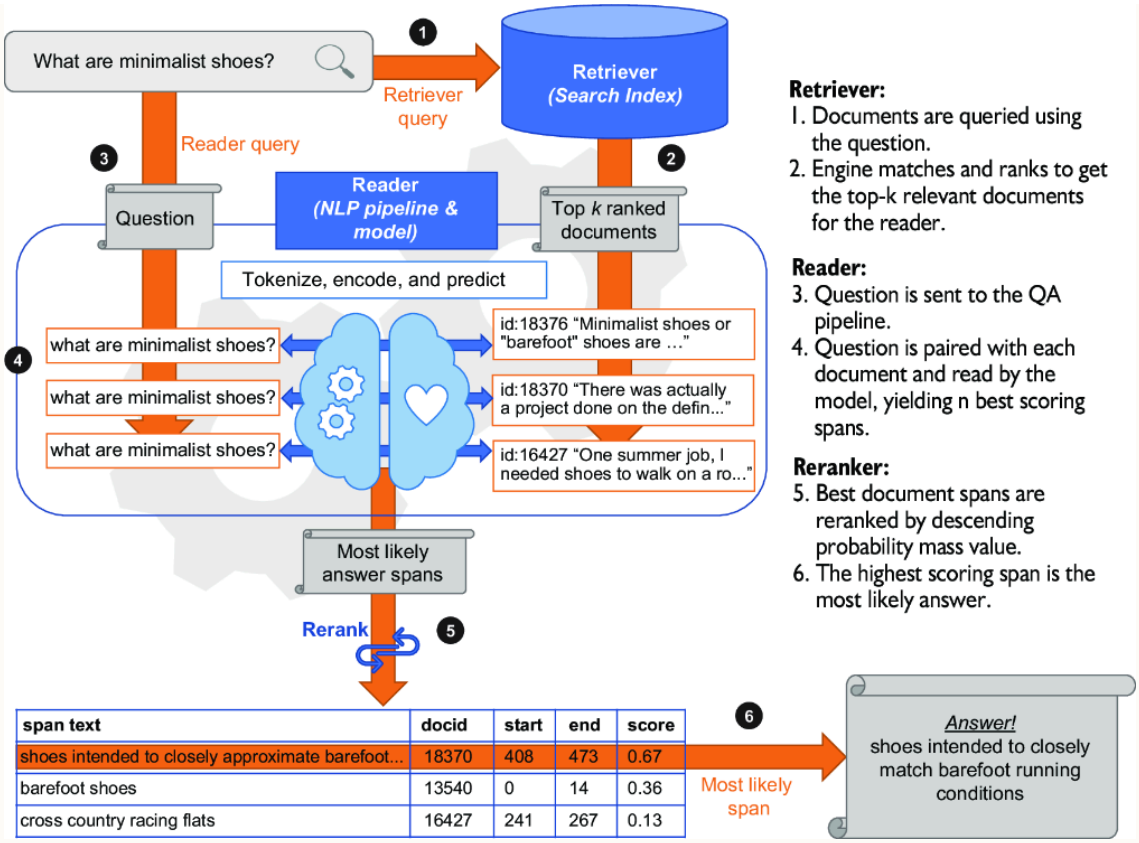

Figure 14.4 Retriever-reader pattern for extractive question answering

51. The reranker sorts the score for each top-*n* answer span in descending order.
62. The top ranked answer from the reranker is the accepted answer and is shown to the user.

To accomplish all of this, we need to tune the retriever, wrangle data to train the reader model, fine-tune the reader model using that data, and build a reranker. Strategies for tuning the retriever (the search engine) have already been covered thoroughly in this book, so for our next step, we'll wrangle the data.

14.2 Constructing a question-answering training dataset

In this section, we'll create a dataset we can use to train our question-answering model. This involves several steps:

- Gathering and cleaning a dataset to fit the question-answering problem space to our content
- Automatically creating a silver set (a semi-refined dataset that needs further labeling) from an existing model and corpus
- Manually correcting the silver set to produce the golden set (a trustworthy dataset we can use for training)
- Splitting the dataset for training, testing, and validation of a fine-tuned model

We'll start with our Stack Exchange outdoors dataset because its data is already well-suited to a question-and-answer application. We need question/answer pairs to use when fine-tuning a base model.

The outdoors dataset is already well-formatted and in bite-size question-and-answer chunks. With the power of Transformers, we can take off-the-shelf tools and models and construct the solution relatively quickly. This is far easier than trying to construct a question/answer dataset from something else, like the book *Great Expectations*. If you're working with long-form text, such as a book or lengthy documents, you'd need to first split the text into paragraphs and manually devise questions for the paragraphs.

GOLDEN SETS AND SILVER SETS

In machine learning, a *golden set* is an accurately labeled dataset that is used to train, test, and validate models. We treat golden sets as highly valuable assets, since gathering them often requires significant manual effort. The accuracy and usability of a trained model are limited by the accuracy and breadth of the golden set. Thus, the longer you spend growing and verifying your golden set, the better the model.

To reduce some of the effort required in labeling data, we can save time by letting a machine try to generate a labeled dataset for us. This automatically generated set of labeled data is called a *silver set*, and it prevents us from having to start from scratch.

A silver set is not as trustworthy as a golden set. Since we automatically obtain a silver set through machine-automated processes that aren't as accurate as humans, there will be mistakes. Thus, a silver set should ideally be improved with a manual audit and corrections to increase its accuracy. Using silver sets to bootstrap your training dataset can save a lot of time and mental effort in the long term, and it can help you scale your training data curation.

14.2.1 Gathering and cleaning a question-answering dataset

On to our first step: let's construct a dataset we can label and use to train a model. For this dataset, we need questions with associated contexts that contain their answers. Listing 14.4 shows how to get the questions and the contexts that contain answers in rows of a pandas dataframe. We need to construct two queries: one to get the community questions, and one to get the accepted community answers to those questions. We will only be using question/answer pairs that have an accepted answer. We will execute the two queries separately and join the two together.

The model that we are using refers to the content from which we pluck our answer as a context. Remember, we're not generating answers, we're just finding the most appropriate answer inside a body of text.

Listing 14.4 Pulling training questions from Solr

```
def get_questions():
  question_types = ["who", "what", "when",  #1
                    "where", "why", "how"]  #1
  questions = []
  for type in question_types:
    request = {"query": type,
               "query_fields": ["title"],
               "return_fields": ["id", "url", "owner_user_id",
                                 "title", "accepted_answer_id"],
               "filters": [("accepted_answer_id", "*")],  #2
               "limit": 10000}
    docs = outdoors_collection.search(**request)["docs"]
    questions += [document for document in docs  #3
                  if document["title"].lower().startswith(type)]  #3
  return questions
```

#1 Narrows the scope of question types that we retrieve
#2 Only retrieves questions that have accepted answers
#3 Only uses titles starting with a question type

With the list of questions from listing 14.4, we next need to get contexts associated with each question. Listing 14.5 returns a dataframe with the following columns: `id`, `url`, `question`,

and `context`. We'll use the `question` and `context` to generate the training and evaluation data for our question-answering model in the coming sections.

Listing 14.5 Searching for accepted answer contexts

```
def get_answers_from_questions(questions, batch_size=500):
  answer_ids = list(set([str(q["accepted_answer_id"])  #1
                         for q in questions]))  #1
  batches = math.ceil(len(answer_ids) / batch_size)  #2
  answers = {}

  for n in range(0, batches):  #3
    ids = answer_ids[n * batch_size:(n + 1) * batch_size]
    request = {"query": "(" + " ".join(ids) + ")",
               "query_fields": "id",
               "limit": batch_size,
               "filters": [("post_type", "answer")],
               "order_by": [("score", "desc")]}
    docs = outdoors_collection.search(**request)["docs"]
    answers |= {int(d["id"]): d["body"] for d in docs}
  return answers

def get_context_dataframe(questions):
  answers = get_answers_from_questions(questions) #4
  contexts = {"id": [], "question": [], "context": [], "url": []}
  for question in questions:
    contexts["id"].append(question["id"])
    contexts["url"].append(question["url"])
    contexts["question"].append(question["title"]),
    if question["accepted_answer_id"] in answers:
      context = answers[question["accepted_answer_id"]]
    else:
      context = "Not found"
    contexts["context"].append(context)
  return pandas.DataFrame(contexts)

questions = get_questions()  #5
contexts = get_context_dataframe(questions)  #6
display(contexts[0:5])
```

#1 Gets a list of all distinct answer ids
#2 Calculates the number of search requests that need to be made
#3 Aggregates all answers
#4 Retrieves all answer data for questions
#5 Load the questions from listing 14.4.
#6 Load the contexts for each question.

Output:

```
id    question                                context
4410  Who places the anchors that rock c...  There are two distinct styl...
5347  Who places the bolts on rock climb...  What you're talking about i...
20662 Who gets the bill if you activate ...  Almost always the victim ge...
11587 What sort of crane, and what sort ...  To answer the snake part of...
7623  What knot is this one? What are it...  Slip knot It's undoubtably ...
```

We encourage you to examine the full output for the question and context pairs to appreciate the variety of input and language used. We also included the original URL if you'd like to visit

the Stack Exchange outdoors website and explore the source data yourself in the Jupyter notebook.

14.2.2 Creating the silver set: Automatically labeling data from a pretrained model

Now that we have our dataset, we have to label it. For the training to work, we need to tell the model what the correct answer is inside of the context (document), given the question. An LLM exists that already does a decent job at selecting answers: `deepset/roberta-base-squad2`. This model was pretrained by the Deepset company using the SQuAD2 dataset and is freely available on their Hugging Face page (**https://huggingface.co/deepset**). SQuAD is the *Stanford Question Answering Dataset*, which is a large public dataset made up of thousands of question and answer pairs. The Deepset team started with the RoBERTa architecture (covered in chapter 13) and fine-tuned a model based on this dataset for the task of question answering.

NOTE It's a good idea to familiarize yourself with the Hugging Face website (**https://huggingface.co**). The Hugging Face community is very active and has provided thousands of free pretrained models, available for use by anyone.

Our strategy is to use the best pretrained model available to attempt to answer all the questions first. We'll call these answers "guesses" and the entire automatically labeled dataset the "silver set". Then we will go through the silver set guesses and correct them ourselves, to get a "golden set".

Listing 14.6 shows our question-answering function, which uses a Transformers pipeline type of `question-answering` and the `deepset/roberta-base-squad2` model. We use these to construct a pipeline with an appropriate tokenizer and target device (either CPU or GPU). This gives us everything we need to pass in raw data and obtain the silver set, as illustrated in figure 14.5.

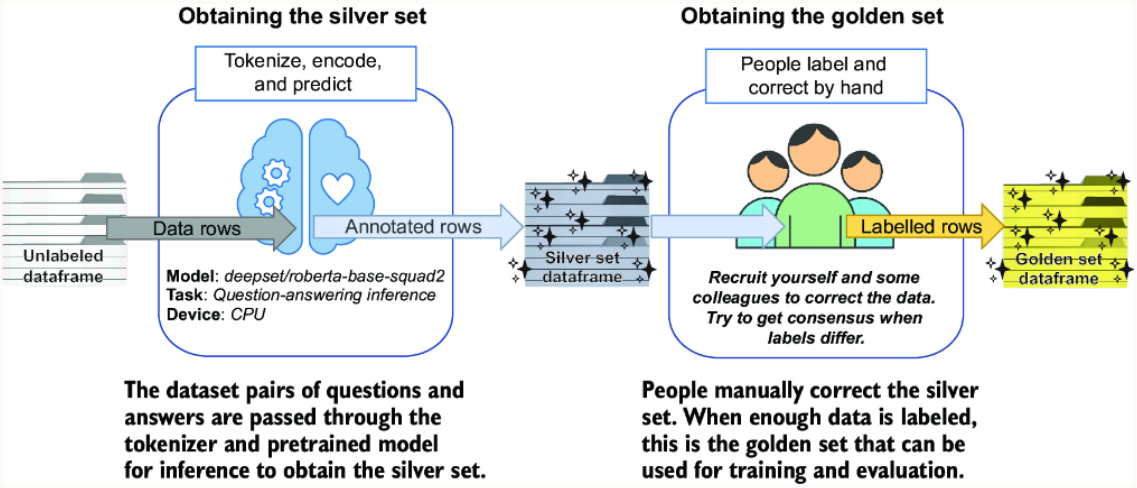

Figure 14.5 Obtaining the silver set and golden set from a prepared dataframe

In Python, we create a function called `answer_questions` that accepts the list of contexts that we extracted from our retriever. This function runs each question and context through the pipeline to generate the answer and appends it to the list. We won't presume that they're actually answers at this point, because many of them will be incorrect (as you'll see when you open the file). We will only count something as an *answer* when it's been vetted by a person. This is the nature of upgrading a silver set to a golden set.

The `device` (CPU or GPU) will be chosen automatically based on whether you have a GPU available to your Docker environment or not. Now is a good time to mention that if you're running or training these models on a CPU-only home computer or Docker configuration, you may be waiting a while for the inference of all the data to be complete. If you're not using a GPU,

feel free to skip running listings 14.6–14.7, as we have already provided the output needed to run later listings in this notebook's dataset.

Listing 14.6 generates the silver set to extract out the most likely answers for our pairs of question and accepted answer contexts loaded previously in listing 14.5.

Listing 14.6 Generating answers given question/context pairs

```
from transformers import pipeline  #1
import torch
import tqdm  #2

def get_processor_device():  #3
  return 0 if torch.cuda.is_available() else -1  #3

def answer_questions(contexts, k=10):
  nlp = pipeline("question-answering", model=model_name,  #4
                 tokenizer=model_name, device=device)  #4
  guesses = []
  for _, row in tqdm.tqdm(contexts[0:k].iterrows(), total=k):  #5
    result = nlp({"question": row["question"],  #6
                  "context": row["context"]})  #6
    guesses.append(result)
  return guesses

model_name = "deepset/roberta-base-squad2"
device = get_processor_device()  #7

guesses = answer_questions(contexts, k=len(contexts))
display_guesses(guesses)
```

#1 This is the pipeline that we illustrated in figure 14.1.
#2 tqdm prints the progress of the operation as a progress bar.
#3 Process with GPU (CUDA) if available; otherwise use the CPU.
#4 This is the pipeline that we illustrated in figure 14.1.
#5 tqdm prints the progress of the operation as a progress bar.
#6 Gets the answer (and confidence score) for every question/context pair
#7 Process with GPU (CUDA) if available; otherwise use the CPU.

Output:

```
score     start  end   answer
0.278927  474    516   a local enthusiast or group of enthusiasts
0.200848  81     117   the person who is creating the climb
0.018632  14     24    the victim
...
0.247008  227    265   the traditional longbow made from wood
0.480407  408    473   shoes intended to closely approximate barefoot run...
0.563754  192    232   a tube of lightweight, stretchy material
```

Congratulations, we have now obtained the silver set! In the next section, we'll improve it.

GPU RECOMMENDED

Feel free to run these listings on your personal computer, but be warned—some of them take a while on a CPU. The total execution time for listing 14.6, for example, was reduced by about 20 times when running on a GPU versus a mid-market CPU during our tests.

Note that the fine-tuning examples later in the chapter will benefit significantly from having a GPU available. If you cannot access a GPU for those listings, that's okay—we've trained the model and already included it for you as part of the outdoors dataset. You can follow along in the listings to see how the model is trained, and if you don't have a GPU available, you can just skip running them. You can also use free services such as Google Colab or rent a server with a GPU from a cloud provider, which is typically a few US dollars per hour.

If you're interested in learning more about GPUs and why they are better suited to tasks such as training models, we recommend checking out *Parallel and High Performance Computing* by Robert Robey and Yuliana Zamora (Manning, 2021).

14.2.3 Human-in-the-loop training: Manually correcting the silver set to produce a golden set

The silver set CSV file (question-answering-squad2-guesses.csv) is used as a first pass at attempting to answer the questions. We'll use this with human-in-the-loop manual correction and labeling of the data to refine the silver set into the golden set.

NOTE No Python code can generate a golden set for you. The data *must* be labeled by an intelligent person (or perhaps one day an AI model highly optimized for making relevance judgments) with an understanding of the domain. All further listings will use this golden set. We are giving you a break, though, since we already labeled the data for you. For reference, it took 4 to 6 hours to label about 200 guesses produced by the `deepset/roberta-base -squad2` model.

Labeling data yourself will give you a deeper appreciation for the difficulty of this NLP task. We *highly* encourage you to label even more documents and rerun the fine-tuning tasks coming up. Having an appreciation for the effort needed to obtain quality data, and the effect it has on the model accuracy, is a lesson you can only learn through experience.

Before diving in and just labeling data, however, we need to have a plan for how and what to label. For each row, we need to classify it and, if necessary, write the correct answer ourselves into another column.

Here is the key, as shown in figure 14.6, which we used for all the labeled rows in the `class` field:

- `-2` = This is a negative example (an example where we know the guess is wrong!).
- `-1` = Ignore this question, as it is too vague or we are missing some information. For example, `What is this bird?` We can't answer without a picture of the bird, so we don't even try.
- `0` = This is an example that has been corrected by a person to highlight a better answer span in the same context. The guess given by `deepset/roberta-base -squad2` was incorrect or incomplete, so we changed it.
- `1` = This is an example that was given a correct answer by `deepset/roberta -base-squad2`, so we did not change the answer.
- (blank) = We didn't check this row, so we'll ignore it.

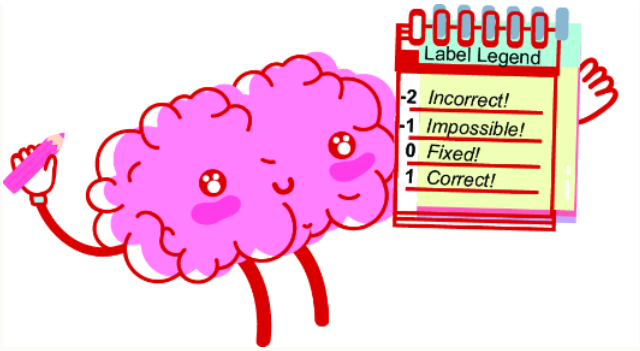

Figure 14.6 Legend of label classes

You should open the outdoors_golden_answers.csv file and look through the rows yourself. Understand the ratio of questions that we labeled as `0` and `1`. You can even try opening the file in pandas and doing a little bit of analysis to familiarize yourself with the golden set.

14.2.4 Formatting the golden set for training, testing, and validation

Now that we have labeled data, we're almost ready to train our model, but first we need to get the data into the right format for the training and evaluation pipeline. Once our data is in the right format, we'll also need to split it into training, testing, and validation sets for use when training the model to ensure it does not overfit our data.

CONVERTING THE LABELED DATA INTO A STANDARDIZED DATA FORMAT

Hugging Face provides a library called `datasets`, which we'll use to prepare our data. The `datasets` library can accept the names of many publicly available datasets and provide a standard interface for working with them. The SQuAD2 dataset is one of the available datasets, but since our golden set is in a custom format, we first need to convert it to the standardized `datasets` configuration format shown in the following listing.

Listing 14.7 Transforming a golden dataset into SQuAD formatting

```
from datasets import Dataset, DatasetDict

def get_training_data(filename):
  golden_answers = pandas.read_csv(filename)
  golden_answers = golden_answers[golden_answers["class"] != None]
  qa_data = []
  for _, row in golden_answers.iterrows():
    answers = row["gold"].split("|")
    starts = [row["context"].find(a) for a in answers]
    missing = -1 in starts
    if not missing:
      row["title"] = row["question"]
      row["answers"] = {"text": answers, "answer_start": starts}
      qa_data.append(row)
  columns = ["id", "url", "title", "question", "context", "answers"]
  df = pandas.DataFrame(qa_data, columns=columns) \  #1
            .sample(frac=1, random_state=0)  #1
  train_split = int(len(df) * 0.75)  #2
  eval_split = (int((len(df) - train_split) / 1.25) +   #3
                train_split - 1)  #3
  train_dataset = Dataset.from_pandas(df[:train_split])
  test_dataset = Dataset.from_pandas(df[train_split:eval_split])
  validation_dataset = Dataset.from_pandas(df[eval_split:])  #4
  return DatasetDict({"train": train_dataset,  #5
                      "test": test_dataset,  #5
                      "validation": validation_dataset})  #5

datadict = get_training_data("data/outdoors/outdoors_golden_answers.csv")
model_path = "data/question-answering/question-answering-training-set"
datadict.save_to_disk(model_path)
```

#1 Randomly sorts all the examples
#2 75% of the examples will be used for training. This will give us 125 training samples.
#3 20% of the examples will be used for testing. We subtract 1 from train_split to allow for 125/32/10 records on the three splits.
#4 The remaining 5% of the examples will be used for validation holdout. This will be 10 samples.
#5 SQuAD requires three groups of data: train, test, and validation

The first part of the function in listing 14.7 loads the CSV into a pandas dataframe and does some preprocessing and formatting. Once formatted, the data is split up into three parts and converted.

The object returned and saved from listing 14.7 is a dataset dictionary (a `datadict`) that contains our three slices for training, testing, and validation. For our data table, with the split defined in `get_training_data`, we have 125 training examples, 32 testing examples, and 10 validation examples.

AVOIDING OVERFITTING WITH A TEST SET AND HOLDOUT VALIDATION SET

Overfitting a model means you trained it to only memorize the training examples provided. This means it won't generalize well enough to handle previously unseen data.

To prevent overfitting, we needed to split our dataset into separate training, testing, and validation slices, as we did in listing 14.7. The testing and the holdout validation sets are used to measure the success of the model after it is trained. After you've gone through the process from end to end, consider labeling some more data and doing different splits across the training, testing, and validation slices to see how the model performs.

We use a training/test split to give some data to the model training and some to test the outcome. We iteratively tune hyperparameters for the model training to achieve higher accuracy (measured using a loss function) when the model is applied to the test set.

The holdout validation set is the real-world proxy for unseen data, and it's not checked until the end. After training and testing are completed, you then validate the final model version by applying it against the holdout examples. If this score is much lower than the final test accuracy, you've overfit your model.

NOTE The number of examples we're using is quite small (125 training examples, 32 testing examples, and 10 holdout validation examples) compared to what you'd use for fine-tuning data for a customer-facing system. As a general rule of thumb, aim for about 500 to 2,000 labeled examples. Sometimes you can get away with less, but typically the more you have the better. This will take considerable time investment, but it's well worth the effort.

14.3 Fine-tuning the question-answering model

We'll now walk through obtaining a better model by fine-tuning the existing `deepset/roberta-base-squad2` model with our golden set.

Unfortunately, this next notebook can be quite slow to run on a CPU. If you are going through the listings on a machine that is CUDA-capable and can configure your Docker environment to use the GPUs, then you should be all set! Otherwise, we recommend you use a service like Google Colab, which offers easy running of Jupyter notebooks on GPUs at no cost, or another cloud computing or hosting provider that has a CUDA device ready to go. You can load the notebook directly from Google Colab and run it without any other dependencies aside from our dataset. A link is provided above listing 14.8 in the associated notebook.

TIP As we noted previously, if you don't want to go through the hassle of setting up a GPU-compatible environment, you can also follow along with listings 14.8–14.13 without running them, since we have already trained the model and included it for you to use. However, if you can, we do encourage you to go through the effort of getting GPU access and training the model yourself to see how the process works and to enable you to tinker with the hyperparameters. Figure 4.7 shows the kind of speedup that GPUs can provide for massively parallel computations like language model training.

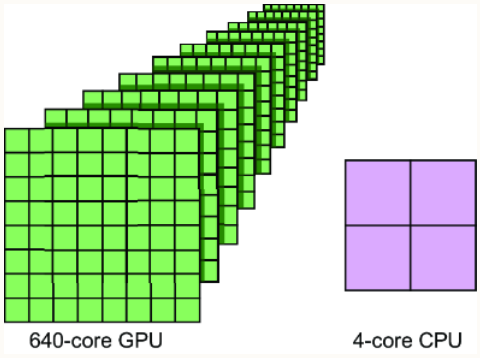

Figure 14.7 A V100 GPU (commonly available with cloud providers) has 640 tensor compute cores, compared to a 4-core x86-64 CPU. A Tesla T4 has 2,560 tensor compute cores. Individually, CPU cores are more powerful, but most GPUs have two to three orders of magnitude more cores than a CPU. This is important when doing massively parallel computations for millions of model parameters.

The first thing we need to do is require access to the GPU device. The code in the following listing will initialize and return the device ID of the available processor. If a GPU is configured and available, we should see that device's ID. If you are using Colab and having any problems with listing 14.8, you may need to change the runtime type to `GPU` in the settings.

Listing 14.8 Detecting and initializing a GPU device

```
def get_processor_type():
  gpu_device = torch.device("cuda:0")
  cpu_device = torch.device("cpu")
  return gpu_device or cpu_device

def get_processor_device():
  return 0 if torch.cuda.is_available() else -1

print("Processor: " + str(get_processor_type()))
print("Device id: " + str(get_processor_device()))
```

Output:

```
Processor: device(type='cuda', index=0)
Device id: 0
```

We have a GPU (in this listing output, at least). In the response, `device(type='cuda', index=0)` is what we were looking for. If a GPU isn't available when you run the listing, `device(type='cpu')` will be returned instead, indicating that the CPU will be used for processing. If you have more than one capable device available to the notebook, it will list each of them with an incrementing numerical id. You can access the device later in training by specifying an `id` (in our case, `0`).

With our device ready to go, we will next load and tokenize our previously labeled dataset from listing 14.7.

14.3.1 Tokenizing and shaping our labeled data

The model trainer doesn't recognize words; it recognizes tokens that exist in the RoBERTa vocabulary. We covered tokenization in chapter 13, where we used it as an initial step when encoding documents and queries into dense vectors for semantic search. Similarly, we need to tokenize our question-answering dataset before we can use it to train the model. The model accepts token values as the input parameters, just like any other Transformer model.

The following listing shows how we'll tokenize the data prior to model training.

Listing 14.9 Tokenizing our training set

```
# This function adapted from:
# https://github.com/huggingface/notebooks/blob/master/examples/
#question_answering.ipynb
# Copyright 2001, Hugging Face. Apache 2.0 Licensed.
from datasets import load_from_disk
from transformers import RobertaTokenizerFast

file = "data/question-answering/question-answering-training-set"
datadict = datasets.load_from_disk(file) #1
tokenizer = from_pretrained("roberta-base") #2
...



def tokenize_dataset(examples):
  maximum_tokens = 384  #3
  document_overlap = 128  #4
  pad_on_right = tokenizer.padding_side == "right" #5
  tokenized_examples = tokenizer(  #6
    examples["question" if pad_on_right  #6
                         else "context"],  #6
    examples["context" if pad_on_right  #6
                       else "question"],  #6
    truncation="only_second" if pad_on_right  #6
                             else "only_first",  #6
    max_length=maximum_tokens,  #6
    stride=document_overlap,  #6
    return_overflowing_tokens=True,  #6
    return_offsets_mapping=True,  #6
    padding="max_length"  #6
  )
  ...  #7
  return tokenized_examples

tokenized_datasets = datadict.map(  #8
  tokenize_dataset,  #8
  batched=True,  #8
  remove_columns=datadict["train"].column_names)  #8
```

#1 Loads the datadict we created in listing 14.7 from our golden set
#2 Loads a pretrained tokenizer (roberta-base)
#3 This will be the number of tokens in both the question and context.
#4 Sometimes we need to split the context into smaller chunks, so these chunks will overlap by this many tokens.
#5 Add padding tokens to the end for question/context pairs that are shorter than the model input size.
#6 Performs the tokenization for each of the examples
#7 Additional processing to identify start and end positions for questions and contexts. See the notebook for the full algorithm.
#8 Invokes the tokenizer on each example in our golden dataset

We load a tokenizer (trained on the `roberta-base` model), load our `question -answer-ing-training-set` golden set from disk (data/question-answering/question -answering-training-set/), and then run the examples from the golden set through the tokenizer to generate a `tokenized_datasets` object with the training and test datasets we'll soon pass to the model trainer.

For each context, we generate a list of tensors with a specific number of embeddings per tensor and a specific number of floats per embedding. The shape of the tensors containing the tokens must be the same for all the examples we provide to the trainer and evaluator. We accomplish this with a window-sliding technique.

Window sliding is a technique that involves splitting a long list of tokens into many sublists of tokens, but where each sublist after the first shares an overlapping number of tokens with the previous sublist. In our case, `maximum_tokens` defines the size of each sublist, and `document_overlap` defines the overlap. This window-sliding process is demonstrated in figure 14.8.

Figure 14.8 demonstrates very small `maximum_tokens` (`24`) and `document_overlap` (`8`) numbers for illustration purposes, but the real tokenization process splits the contexts into tensors of `384` tokens with an overlap of `128`.

The window-sliding technique also makes use of *padding* to ensure that each tensor is the same length. If the number of tokens in the last tensor of the context is less than the maximum (`384`), then the rest of the positions in the tensor are filled with an empty marker token so that the final tensor size is also `384`.

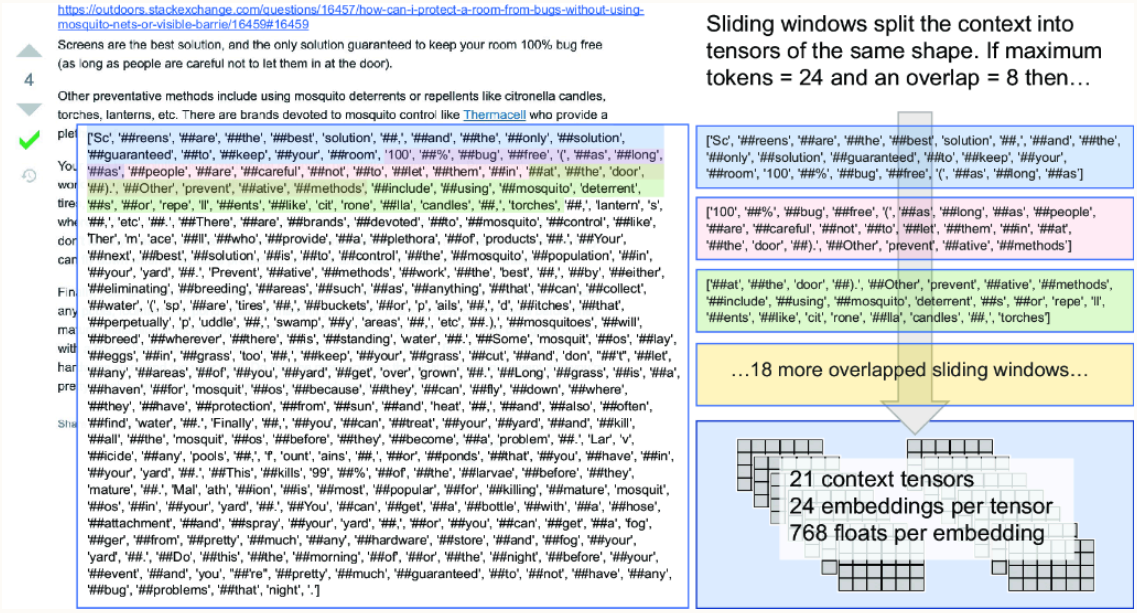

Figure 14.8 Visualizing the sliding window technique that splits one context into tensors of the same shape

Knowing how the contexts are processed is important, as it can affect both accuracy and processing time. If we're trying to identify answers in lengthy documents, the window-sliding process may reduce accuracy, particularly if the `maximum_tokens` and `document_overlap` are small and thus fragment the context too much. Long documents will also get sliced up into multiple tensors that collectively take longer to process. Most of the contexts in the outdoors dataset fit into the maximums we specified, but these trade-offs are important to consider in other datasets when choosing your `maximum_tokens` and `document_overlap` parameters.

14.3.2 Configuring the model trainer

We have one last step before we train our model: we need to specify *how* training and evaluation will happen.

When training our model, we need to specify the base model and training arguments (hyperparameters), as well as our training and testing datasets. You'll want to understand the following key concepts when configuring the hyperparameters for the model trainer:

- *Epochs*—The number of times the trainer will iterate over the dataset. More epochs help reinforce the context and reduce loss over time. Having too many epochs will likely overfit your model, though, and 3 epochs is a common choice when fine-tuning Transformers.
- *Batch sizes*—The number of examples that will be trained/evaluated at once. A higher batch size might produce a better model. This setting is constrained by GPU core count and avail-

able memory, but common practice is to fit as much in a batch as possible to make the most of the available resources.

- *Warmups*—When training a model, it can be helpful to slowly tune the model initially, so that the early examples don't have an undue influence on the model's learned parameters. Warmup steps allow gradual improvements to the model (the *learning rate*), which helps prevent the trainer from overfitting on early examples.
- *Decay*—Weight decay is used to reduce overfitting by multiplying each weight by this constant value at each step. It is common to use 0.01 as the weight decay, but this can be changed to a higher value if the model is quickly overfitting or to a lower value if you don't see improvement fast enough.

Listing 4.10 demonstrates configuring the model trainer. The hyperparameters (`training_args`) we've specified in the listing are those used by SQuAD2 by default, but feel free to adjust any of them to see how it improves the quality of the question-answering model for your own questions.

When trying to choose the best settings, a common technique is to perform a grid search over these hyperparameters. A *grid search* is a process that automatically iterates over parameter values and tests how adjusting each of them in different combinations improves the quality of the trained models. We include a grid search example in the accompanying notebooks, should you wish to dive deeper into parameter tuning, but for now we'll proceed with the hyperparameters specified in listing 14.10.

Listing 14.10 Initializing the trainer and its hyperparameters

```
from transformers import RobertaForQuestionAnswering, TrainingArguments,
                         Trainer, default_data_collator

model = RobertaForQuestionAnswering.from_pretrained(
  "deepset/roberta-base-squad2")

training_args = TrainingArguments(
  evaluation_strategy="epoch",  #1
  num_train_epochs=3,  #2
  per_device_train_batch_size=16,  #3
  per_device_eval_batch_size=64,  #4
  warmup_steps=500,  #5
  weight_decay=0.01, #6
  logging_dir="data/question-answering/logs",
  output_dir="data/question-answering/results")

trainer = Trainer(
  model=model,  #7
  args=training_args,  #8
  data_collator=default_data_collator,
  tokenizer=tokenizer,
  train_dataset=tokenized_datasets["train"], #9
  eval_dataset=tokenized_datasets["test"])  #10
```

#1 Evaluates loss per epoch
#2 Total number of training epochs
#3 Batch size per device during training
#4 Batch size for evaluation
#5 Number of warmup steps for learning rate scheduler
#6 Strength of weight decay
#7 The instantiated Hugging Face Transformers model to be trained
#8 Training arguments
#9 Specifies the training dataset
#10 Specifies the evaluation dataset

14.3.3 Performing training and evaluating loss

With our hyperparameters all set, it's now time to train the model. The following listing runs the previously configured trainer, returns the training output showing the model's performance, and saves the model.

Listing 14.11 Training and saving the model

```
trainer.train()
model_name = "data/question-answering/roberta-base-squad2-fine-tuned"
trainer.save_model(model_name)
```

Output:

```
[30/30 00:35, Epoch 3/3]
Epoch   Training Loss   Validation Loss     Runtime     Samples Per Second
1       No log          2.177553            1.008200    43.642000
2       No log          2.011696            1.027800    42.811000
3       No log          1.938573            1.047700    41.996000

TrainOutput(global_step=30, training_loss=2.531823984781901,
            metrics={'train_runtime': 37.1978,
                     'train_samples_per_second': 0.806,
                     'total_flos': 133766734473216, 'epoch': 3.0})
```

A *loss function* is a decision function that uses the error to give a quantified estimate of how bad a model is. Lower loss means a higher-quality model. What we're looking for is a gradual reduction in loss with each epoch, which indicates that the model is continuing to get better with more training. We went from a validation loss of `2.178` to `2.012` to `1.939` on our testing set. The numbers are all getting smaller at a steady rate (no huge jumps), and that's a good sign.

The overall training loss for this freshly fine-tuned model is `2.532`, and the validation loss on our testing set is `1.939`. Given the constraints of our small fine-tuning dataset and hyperparameter configuration, a validation loss as small as `1.939` is quite good.

14.3.4 Holdout validation and confirmation

How do we know if our trained model can be used successfully for real-world question answering? Well, we need to test the model against our holdout validation dataset. Recall that the holdout validation set is the third dataset (with only 10 examples) in our `datadict` from listing 14.9.

Figure 14.9 underscores the purpose of the holdout validation set. We want the loss from the evaluation of our holdout set to be as good as our validation loss of `1.939` from listing 14.11. If our holdout loss turns out higher, that would be a red flag that we may have overfit! Let's see how our model performs in the following listing.

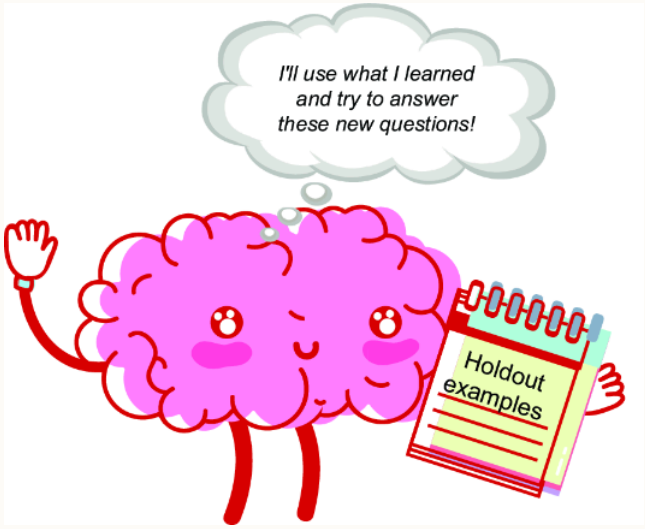

Figure 14.9 Holdout set: answering previously unseen questions with our trained mode

Listing 14.12 Evaluating the trained model on the holdout examples

```
evaluation = trainer.evaluate(eval_dataset=tokenized_datasets["validation"])
display(evaluation)
```

Output:

```
{"eval_loss": 1.7851890325546265,
 "eval_runtime": 2.9417,
 "eval_samples_per_second": 5.099,
 "eval_steps_per_second": 0.34,
 "epoch": 3.0}
```

The `eval_loss` of `1.785` from testing our holdout validation set looks great. It's even better than the training and testing loss. This means that our model is working well and is likely not overfitting the training or testing data.

Feel free to continue training and improving the model, but we'll continue with this as the fully trained model that we'll integrate into the reader for our question-answering system.

14.4 Building the reader with the new fine-tuned model

Now that our reader's model training is completed, we'll integrate it into a question-answering pipeline to produce our finalized reader that can extract answers from questions and contexts. The following listing demonstrates how we can load our model into a `question-answering` pipeline provided by the `transformers` library.

Listing 14.13 Loading the fine-tuned outdoors question-answering model

```
device = get_processor_device()
model_name = "data/question-answering/roberta-base-squad2-fine-tuned"
nlp2 = pipeline("question-answering", model=model_name,
                tokenizer=model_name, device=device)
```

With the question-answering pipeline loaded, we'll extract some answers from some question/context pairs. Let's use our 10-document holdout validation set used earlier in section 14.3.4. The holdout examples were not used to train or to test the model, so they should be a good litmus test for how well our model works in practice.

In the following listing, we test the accuracy of our question-answering model on the holdout validation set examples.

Listing 14.14 Evaluating the fine-tuned question-answering model

```
def answer_questions(examples):
  answers = []
  success = 0
  for example in examples:
    question = {"question": example["question"][0],
                "context": example["context"][0]}
    answer = nlp2(question)
    label = example["answers"][0]["text"][0]
    result = answer["answer"]
    print(question["question"])
    print("Label:", label)
    print("Result:", result)
    print("----------")
    success += 1 if label == result else 0
    answers.append(answer)
  print(f"{success}/{len(examples)} correct")

datadict["validation"].set_format(type="pandas", output_all_columns=True)
validation_examples = [example for example in datadict["validation"]]
answer_questions(validation_examples)
```

Output:

```
How to get pine sap off my teeth
Label: Take a small amount of margarine and rub on the sap
Result: Take a small amount of margarine and rub on the sap

Why are backpack waist straps so long?
Label: The most backpacks have only one size for everyone
Result: The most backpacks have only one size for everyone

...

How efficient is the Altai skis "the Hok"?
Label: you can easily glide in one direction (forward) and if you try to
       glide backwards, the fur will "bristle up"
Result: you can easily go uphill, without (much) affecting forward gliding performa

7/10 Correct
```

Successfully extracting 7 out of 10 correct answers is an impressive result. Congratulations, you've now fine-tuned an LLM for a real-world use case! This completes the `reader` component of our architecture, but we still need to combine it with a `retriever` that finds the initial candidate contexts to pass to the `reader`. In the next section, we'll incorporate the retriever (our search engine) to finalize the end-to-end question-answering system.

14.5 Incorporating the retriever: Using the question-answering model with the search engine

Next, we'll implement a rerank operation using the reader confidence score to rank the top answers. Here's an outline of the steps we'll go through in this exercise:

1. Query the outdoors index from a search collection tuned for high recall.

2. Pair our question with the top-*K* document results and infer answers and scores with the question-answering NLP inference pipeline.
3. Rerank the answer predictions by descending score.
4. Return the correct answer and top results using the parts created in steps 1 through 3.

See figure 14.4 for a refresher on this application flow.

14.5.1 Step 1: Querying the retriever

Our goal in the first retrieval stage is recall. Specifically, what are all the possibly relevant documents that may contain our answer? We rely on the already-tuned search collection to give us that recall so that we can pass quality documents into our reranking stage.

The following listing implements our `retriever` function, which can accept a question and return an initial list of relevant documents to consider as potential contexts for the answer.

Listing 14.15 Retriever function that searches for relevant answers

```
nlp = spacy.load("en_core_web_sm")  #1
nlp.remove_pipe("ner")

def get_query_from_question(question):
  words = [token.text for token in nlp(question)
           if not (token.lex.is_stop or token.lex.is_punct)]
  return " ".join(words)

def retriever(question):
  contexts = {"id": [], "question": [], "context": [], "url": []}
  query = get_query_from_question(question) #2
  request = {"query": query,
             "query_fields": ["body"],
             "return_fields": ["id", "url", "body"],
             "filters": [("post_type", "answer")],  #3
             "limit": 5}
  docs = outdoors_collection.search(**request)["docs"]
  for doc in docs:
    contexts["id"].append(doc["id"])
    contexts["url"].append(doc["url"])
    contexts["question"].append(question)
    contexts["context"].append(doc["body"])
  return pandas.DataFrame(contexts)

example_contexts = retriever('What are minimalist shoes?')
display_contexts(example_contexts)
```

#1 Uses the English spaCy NLP model
#2 Converts the question to a query by removing stop words and focusing on important parts of speech (see the notebook for the implementation)
#3 Only gets answer documents (not questions)

Response:

```
id     question                    context
18376  What are minimalist shoes?  Minimalist shoes or "barefoot" shoes are shoes..
18370  What are minimalist shoes?  There was actually a project done on the defin..
16427  What are minimalist shoes?  One summer job, I needed shoes to walk on a ro..
18375  What are minimalist shoes?  The answer to this question will vary on your...
13540  What are minimalist shoes?  Barefoot Shoes Also known as minimalist shoes,..
```

One problem we face when using the question as the query is noise. There are lots of documents that have the terms “who”, “what”, “when”, “where”, “why”, and “how”, as well as other stop words and less important parts of speech. Although BM25 may do a good job of deprioritizing these terms in a ranking function, we know those are not the key terms a user is searching for, so we remove them in the `get_query_from_question` function to reduce noise. We covered part of speech tagging with spaCy previously in chapters 5 and 13, so we won’t repeat the implementation here (you can find it in the notebook).

With a good set of documents returned from the search engine that may contain the answers to the user’s question, we can now pass those documents as contexts to the `reader` model.

14.5.2 Step 2: Inferring answers from the reader model

We now can use the `reader` model to infer answers to the question from each of the top *N* contexts. Listing 14.16 implements our generic `reader` interface, which accepts the output from the `retriever` from step 1. The model and pipeline loading for the `retriever` follow the same process as in listing 14.13, while the rest of the `reader` implementation specifically handles generating candidate answers (along with scores for each answer) from the passed-in contexts.

Listing 14.16 Reader function that incorporates our fine-tuned model

```
from transformers import pipeline

device = get_processor_device()
model_name = "data/question-answering/roberta-base-squad2-fine-tuned"
qa_nlp = pipeline("question-answering", model=model_name,  #1
                  tokenizer=model_name, device=device)  #1

def reader(contexts):
  answers = []
  for _, row in contexts.iterrows():  #2
    answer = qa_nlp({"question": row["question"],  #2
                     "context": row["context"]})  #2
    answer["id"] = row["id"]  #3
    answer["url"] = row["url"]  #3
    answers.append(answer)  #2
  return answers
```

#1 Creates a spaCy pipeline using our fine-tuned model
#2 Invokes the reader pipeline to extract a candidate answer from each context
#3 Returns additional metadata about where each answer was found

The `reader` returns an answer from each context based upon our fine-tuned model, along with the `id`, `url`, and `score` for the answer.

14.5.3 Step 3: Reranking the answers

Listing 14.17 shows a straightforward function that reranks the answers by simply sorting them by the score (the probability mass function outputs) from the `reader` model. The top answer is the most likely and is therefore shown first. You can show one answer, or you can show all that are returned by the reader. Indeed, sometimes it might be useful to give the question-asker multiple options and let them decide. This increases the odds of showing a correct answer, but it also takes up more space in the browser or application presenting the answers, so it may require a user experience trade-off.

Listing 14.17 The reranker sorts on scores from the reader

```
def reranker(answers):
  return sorted(answers, key=lambda k: k["score"], reverse=True)
```

We should note that your reranker could be more sophisticated, potentially incorporating multiple conditional models or even attempting to combine multiple answers together (such as overlapping answers from multiple contexts). For our purposes, we'll just rely on the top score.

14.5.4 Step 4: Returning results by combining the retriever, reader, and reranker

We're now ready to assemble all the components of our question-answering (QA) system. The hard part is done, so we can put them in one function, aptly named `ask`, which will accept a query and print out the answer.

Listing 14.18 QA function combining retriever, reader, and reranker

```
def ask(question):
  documents = retriever(question)
  answers = reader(documents)
  reranked = reranker(answers)
  print_answer(question, reranked)

ask('What is the best mosquito repellent?')
ask('How many miles can a person hike day?')
ask('How much water does a person need per day?')
```

Response:

```
What is the best mosquito repellent?
1116   DEET (0.606)
1056   thiamine (0.362)
569    Free-standing bug nets (0.158)
1076   Insect repellent is not 100% effective (0.057)
829    bear-spray (0.05)

How many miles can a person hike day?
17651 20-25 (0.324)
19609 12 miles (0.164)
19558 13 (0.073)
13030 25-35 (0.065)
4536 13 miles (0.022)

How much water does a person need per day?
1629 3 liters (0.46)
193 MINIMUM a gallon (0.235)
20634 0.4 to 0.6 L/day (0.207)
11679 4 litres (0.084)
11687 carry water (0.037)
```

These results look pretty good. Note that in some cases multiple contexts could return the same answer. Generally, this would be a strong signal of a correct answer, so it may be a signal to consider integrating into your reranking.

It's amazing to see the quality of results possible using these out-of-the-box models with minimal retraining. Kudos to the NLP community for making these open source tools, techniques, models, and datasets freely available and straightforward to use!

Congratulations, you've successfully implemented an end-to-end question-answering system that extracts answers from search results. You generated a silver set of answers, saw how to improve them into a golden set, loaded and fine-tuned a question-answering reader model, and implemented the retriever-reader pattern, using your trained model and the search engine.

With LLMs, we can do much more than just extract answers from search results, however. LLM's can be fine-tuned to perform abstractive question answering to generate answers not seen in search results but synthesized from multiple sources. They can also be trained to summarize search results for users or even synthesize brand-new content (text, images, etc.) in response to user input. Many LLMs are trained on so much data across such a widespread amount of human knowledge (such as the majority of the known internet), that they can often perform a wide variety of tasks like this well out of the box. These foundation models, which we'll cover in the next chapter, are paving the way for the next evolution of both AI and AI-powered search.

Summary

- An extractive question-answering system generally follows the retriever-reader pattern, where possible contexts (documents) are found by the retriever and are then analyzed using the reader model to extract the most likely answer.
- A search engine serves as a great retriever, since it is specifically designed to take a query and return ranked documents that are likely to serve as relevant context for the query.
- A reader model analyzes spans of text to predict the most likely beginning and ending of the answer within each context, scoring all options to extract the most likely answer.
- Curating a training dataset is time-intensive, but you can generate a silver set of training data automatically using a pretrained model. You can then tweak the answers in the silver set to save significant effort compared to creating the entire golden training dataset manually.
- You can fine-tune a pretrained model to your specific dataset using a training, testing, and holdout validation dataset and optimizing for a loss minimization function.

Previous chapter
‹ 13 Semantic search with dense vectors

Next chapter
15 Foundation models and emerging search paradigms ›