CHAPTER 5
Text Generation

One of the most uncanny features of transformer-based language models is their ability to generate text that is almost indistinguishable from text written by humans. A famous example is OpenAI's GPT-2, which when given the prompt:[1]

> In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

was able to generate a compelling news article about talking unicorns:

> The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science. Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved. Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow. Pérez and the others then ventured further into the valley. "By the time we reached the top of one peak, the water looked blue, with some crystals on top," said Pérez. Pérez and his friends were astonished to see the unicorn herd. These creatures could be seen from the air without having to move too much to see them—they were so close they could touch their horns. While examining these bizarre creatures the scientists discovered that the creatures also spoke some fairly regular English …

1 This example comes from OpenAI's blog post on GPT-2.

What makes this example so remarkable is that it was generated without any explicit supervision! By simply learning to predict the next word in the text of millions of web pages, GPT-2 and its more powerful descendants like GPT-3 are able to acquire a broad set of skills and pattern recognition abilities that can be activated with different kinds of input prompts. Figure 5-1 shows how language models are sometimes exposed during pretraining to sequences of tasks where they need to predict the following tokens based on the context alone, like addition, unscrambling words, and translation. This allows them to transfer this knowledge effectively during fine-tuning or (if the model is large enough) at inference time. These tasks are not chosen ahead of time, but occur naturally in the huge corpora used to train billion-parameter language models.

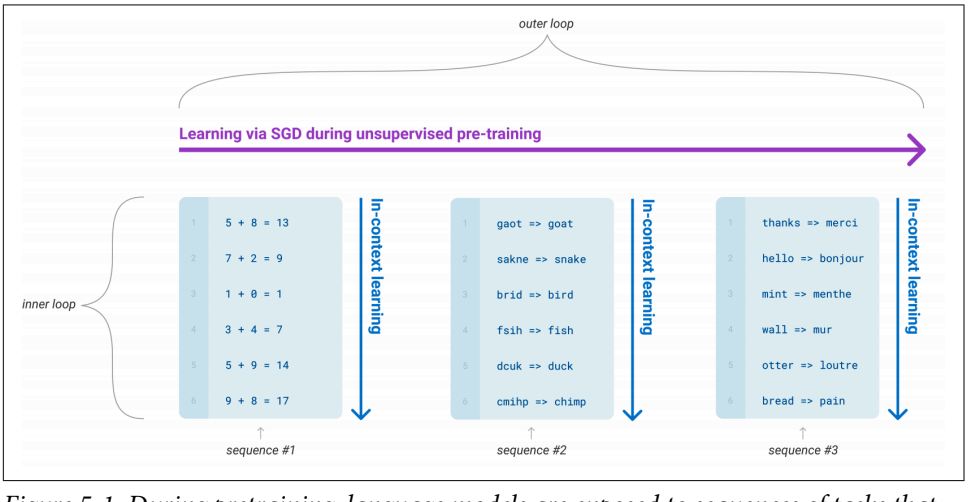


Figure 5-1. During pretraining, language models are exposed to sequences of tasks that can be adapted during inference (courtesy of Tom B. Brown)

The ability of transformers to generate realistic text has led to a diverse range of applications, like InferKit, Write With Transformer, AI Dungeon, and conversational agents like Google's Meena that can even tell corny jokes, as shown in Figure 5-2![2]

2 However, as Delip Rao points out, whether Meena *intends* to tell corny jokes is a subtle question.

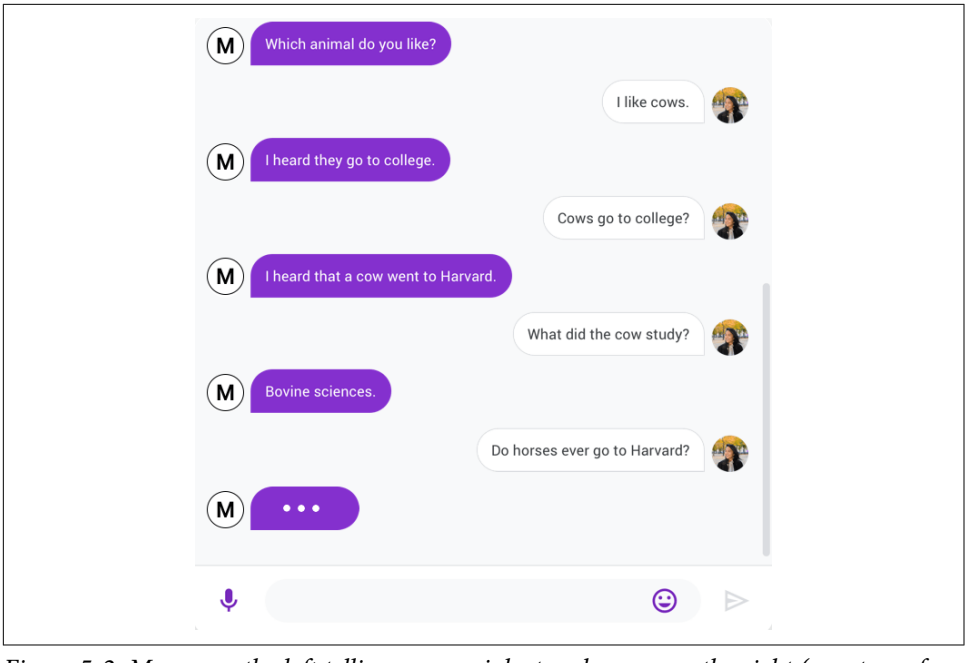


Figure 5-2. Meena on the left telling a corny joke to a human on the right (courtesy of Daniel Adiwardana and Thang Luong)

In this chapter we'll use GPT-2 to illustrate how text generation works for language models and explore how different decoding strategies impact the generated texts.

The Challenge with Generating Coherent Text

So far in this book, we have focused on tackling NLP tasks via a combination of pretraining and supervised fine-tuning. As we've seen, for task-specific heads like sequence or token classification, generating predictions is fairly straightforward; the model produces some logits and we either take the maximum value to get the predicted class, or apply a softmax function to obtain the predicted probabilities per class. By contrast, converting the model's probabilistic output to text requires a *decoding method*, which introduces a few challenges that are unique to text generation:

- The decoding is done *iteratively* and thus involves significantly more compute than simply passing inputs once through the forward pass of a model.
- The *quality* and *diversity* of the generated text depend on the choice of decoding method and associated hyperparameters.

To understand how this decoding process works, let's start by examining how GPT-2 is pretrained and subsequently applied to generate text.

Like other *autoregressive* or *causal language models*, GPT-2 is pretrained to estimate the probability $P(\mathbf{y}|\mathbf{x})$ of a sequence of tokens $\mathbf{y} = y_1, y_2, ...y_t$ occurring in the text, given some initial prompt or context sequence $\mathbf{x} = x_1, x_2, ...x_k$. Since it is impractical to acquire enough training data to estimate $P(\mathbf{y}|\mathbf{x})$ directly, it is common to use the chain rule of probability to factorize it as a product of *conditional* probabilities:

$$P(y_1, ..., y_t|\mathbf{x}) = \prod_{t=1}^{N} P(y_t|y_{<t}, \mathbf{x})$$

where $y_{<t}$ is a shorthand notation for the sequence $y_1, ..., y_{t-1}$. It is from these conditional probabilities that we pick up the intuition that autoregressive language modeling amounts to predicting each word given the preceding words in a sentence; this is exactly what the probability on the righthand side of the preceding equation describes. Notice that this pretraining objective is quite different from BERT's, which utilizes both *past* and *future* contexts to predict a *masked* token.

By now you may have guessed how we can adapt this next token prediction task to generate text sequences of arbitrary length. As shown in Figure 5-3, we start with a prompt like "Transformers are the" and use the model to predict the next token. Once we have determined the next token, we append it to the prompt and then use the new input sequence to generate another token. We do this until we have reached a special end-of-sequence token or a predefined maximum length.

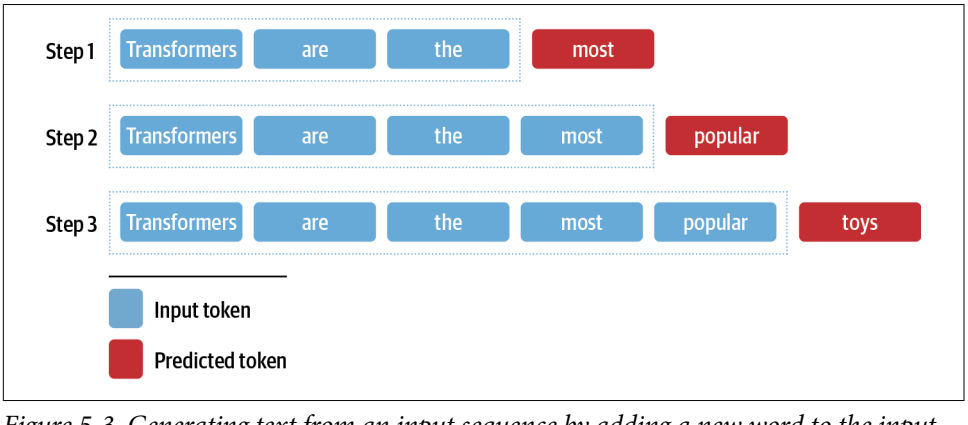


Figure 5-3. Generating text from an input sequence by adding a new word to the input at each step

Since the output sequence is *conditioned* on the choice of input prompt, this type of text generation is often called *conditional text generation*.

At the heart of this process lies a decoding method that determines which token is selected at each timestep. Since the language model head produces a logit $z_{t,i}$ per token in the vocabulary at each step, we can get the probability distribution over the next possible token w_i by taking the softmax:

$$P(y_t = w_i \mid y_{<t}, \mathbf{x}) = \text{softmax}(z_{t,i})$$

The goal of most decoding methods is to search for the most likely overall sequence by picking a $\hat{\mathbf{y}}$ such that:

$$\hat{\mathbf{y}} = \underset{\mathbf{y}}{\text{argmax}}\, P(\mathbf{y} \mid \mathbf{x})$$

Finding $\hat{\mathbf{y}}$ directly would involve evaluating every possible sequence with the language model. Since there does not exist an algorithm that can do this in a reasonable amount of time, we rely on approximations instead. In this chapter we'll explore a few of these approximations and gradually build up toward smarter and more complex algorithms that can be used to generate high-quality texts.

Greedy Search Decoding

The simplest decoding method to get discrete tokens from a model's continuous output is to greedily select the token with the highest probability at each timestep:

$$\hat{y}_t = \underset{y_t}{\text{argmax}}\, P(y_t \mid y_{<t}, \mathbf{x})$$

To see how greedy search works, let's start by loading the 1.5-billion-parameter version of GPT-2 with a language modeling head:[3]

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

device = "cuda" if torch.cuda.is_available() else "cpu"
model_name = "gpt2-xl"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name).to(device)
```

Now let's generate some text! Although 🤗 Transformers provides a `generate()` function for autoregressive models like GPT-2, we'll implement this decoding method

3 If you run out of memory on your machine, you can load a smaller GPT-2 version by replacing `model_name = "gpt-xl"` with `model_name = "gpt"`.

ourselves to see what goes on under the hood. To warm up, we'll take the same iterative approach shown in Figure 5-3: we'll use "Transformers are the" as the input prompt and run the decoding for eight timesteps. At each timestep, we pick out the model's logits for the last token in the prompt and wrap them with a softmax to get a probability distribution. We then pick the next token with the highest probability, add it to the input sequence, and run the process again. The following code does the job, and also stores the five most probable tokens at each timestep so we can visualize the alternatives:

```
import pandas as pd

input_txt = "Transformers are the"
input_ids = tokenizer(input_txt, return_tensors="pt")["input_ids"].to(device)
iterations = []
n_steps = 8
choices_per_step = 5

with torch.no_grad():
    for _ in range(n_steps):
        iteration = dict()
        iteration["Input"] = tokenizer.decode(input_ids[0])
        output = model(input_ids=input_ids)
        # Select logits of the first batch and the last token and apply softmax
        next_token_logits = output.logits[0, -1, :]
        next_token_probs = torch.softmax(next_token_logits, dim=-1)
        sorted_ids = torch.argsort(next_token_probs, dim=-1, descending=True)
        # Store tokens with highest probabilities
        for choice_idx in range(choices_per_step):
            token_id = sorted_ids[choice_idx]
            token_prob = next_token_probs[token_id].cpu().numpy()
            token_choice = (
                f"{tokenizer.decode(token_id)} ({100 * token_prob:.2f}%)"
            )
            iteration[f"Choice {choice_idx+1}"] = token_choice
        # Append predicted next token to input
        input_ids = torch.cat([input_ids, sorted_ids[None, 0, None]], dim=-1)
        iterations.append(iteration)

pd.DataFrame(iterations)
```

	Input	Choice 1	Choice 2	Choice 3	Choice 4	Choice 5
0	Transformers are the	most (8.53%)	only (4.96%)	best (4.65%)	Transformers (4.37%)	ultimate (2.16%)
1	Transformers are the most	popular (16.78%)	powerful (5.37%)	common (4.96%)	famous (3.72%)	successful (3.20%)
2	Transformers are the most popular	toy (10.63%)	toys (7.23%)	Transformers (6.60%)	of (5.46%)	and (3.76%)
3	Transformers are the most popular toy	line (34.38%)	in (18.20%)	of (11.71%)	brand (6.10%)	line (2.69%)

	Input	Choice 1	Choice 2	Choice 3	Choice 4	Choice 5
4	Transformers are the most popular toy line	in (46.28%)	of (15.09%)	, (4.94%)	on (4.40%)	ever (2.72%)
5	Transformers are the most popular toy line in	the (65.99%)	history (12.42%)	America (6.91%)	Japan (2.44%)	North (1.40%)
6	Transformers are the most popular toy line in the	world (69.26%)	United (4.55%)	history (4.29%)	US (4.23%)	U (2.30%)
7	Transformers are the most popular toy line in the world	, (39.73%)	. (30.64%)	and (9.87%)	with (2.32%)	today (1.74%)

With this simple method we were able to generate the sentence "Transformers are the most popular toy line in the world". Interestingly, this indicates that GPT-2 has internalized some knowledge about the Transformers media franchise, which was created by two toy companies (Hasbro and Takara Tomy). We can also see the other possible continuations at each step, which shows the iterative nature of text generation. Unlike other tasks such as sequence classification where a single forward pass suffices to generate the predictions, with text generation we need to decode the output tokens one at a time.

Implementing greedy search wasn't too hard, but we'll want to use the built-in `generate()` function from 🤗 Transformers to explore more sophisticated decoding methods. To reproduce our simple example, let's make sure sampling is switched off (it's off by default, unless the specific configuration of the model you are loading the checkpoint from states otherwise) and specify the `max_new_tokens` for the number of newly generated tokens:

```
input_ids = tokenizer(input_txt, return_tensors="pt")["input_ids"].to(device)
output = model.generate(input_ids, max_new_tokens=n_steps, do_sample=False)
print(tokenizer.decode(output[0]))
```

```
Transformers are the most popular toy line in the world,
```

Now let's try something a bit more interesting: can we reproduce the unicorn story from OpenAI? As we did previously, we'll encode the prompt with the tokenizer, and we'll specify a larger value for `max_length` to generate a longer sequence of text:

```
max_length = 128
input_txt = """In a shocking finding, scientist discovered \
a herd of unicorns living in a remote, previously unexplored \
valley, in the Andes Mountains. Even more surprising to the \
researchers was the fact that the unicorns spoke perfect English.\n\n
"""
input_ids = tokenizer(input_txt, return_tensors="pt")["input_ids"].to(device)
output_greedy = model.generate(input_ids, max_length=max_length,
                               do_sample=False)
print(tokenizer.decode(output_greedy[0]))
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.


The researchers, from the University of California, Davis, and the University of
Colorado, Boulder, were conducting a study on the Andean cloud forest, which is
home to the rare species of cloud forest trees.


The researchers were surprised to find that the unicorns were able to
communicate with each other, and even with humans.


The researchers were surprised to find that the unicorns were able
```

Well, the first few sentences are quite different from the OpenAI example and amusingly involve different universities being credited with the discovery! We can also see one of the main drawbacks with greedy search decoding: it tends to produce repetitive output sequences, which is certainly undesirable in a news article. This is a common problem with greedy search algorithms, which can fail to give you the optimal solution; in the context of decoding, they can miss word sequences whose overall probability is higher just because high-probability words happen to be preceded by low-probability ones.

Fortunately, we can do better—let's examine a popular method known as *beam search decoding*.

Although greedy search decoding is rarely used for text generation tasks that require diversity, it can be useful for producing short sequences like arithmetic where a deterministic and factually correct output is preferred.[4] For these tasks, you can condition GPT-2 by providing a few line-separated examples in the format `"5 + 8 => 13 \n 7 + 2 => 9 \n 1 + 0 =>"` as the input prompt.

Beam Search Decoding

Instead of decoding the token with the highest probability at each step, beam search keeps track of the top-b most probable next tokens, where b is referred to as the number of *beams* or *partial hypotheses*. The next set of beams are chosen by considering all possible next-token extensions of the existing set and selecting the b most likely extensions. The process is repeated until we reach the maximum length or an EOS

4 N.S. Keskar et al., "CTRL: A Conditional Transformer Language Model for Controllable Generation", (2019).

token, and the most likely sequence is selected by ranking the b beams according to their log probabilities. An example of beam search is shown in Figure 5-4.

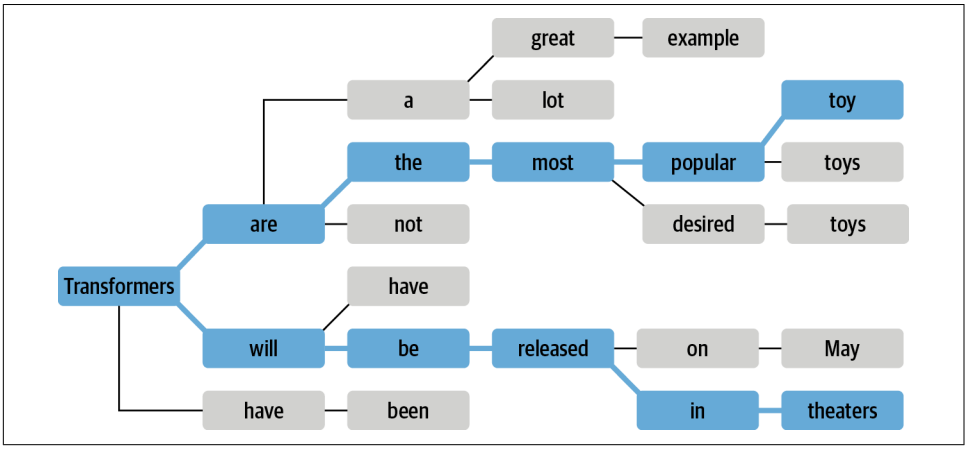


Figure 5-4. Beam search with two beams

Why do we score the sequences using log probabilities instead of the probabilities themselves? That calculating the overall probability of a sequence $P(y_1, y_2, ..., y_t | \mathbf{x})$ involves calculating a *product* of conditional probabilities $P(y_t | y_{<t}, \mathbf{x})$ is one reason. Since each conditional probability is typically a small number in the range $[0, 1]$, taking their product can lead to an overall probability that can easily underflow. This means that the computer can no longer precisely represent the result of the calculation. For example, suppose we have a sequence of $t = 1024$ tokens and generously assume that the probability for each token is 0.5. The overall probability for this sequence is an extremely small number:

```
0.5 ** 1024
```

```
5.562684646268003e-309
```

which leads to numerical instability as we run into underflow. We can avoid this by calculating a related term, the log probability. If we apply the logarithm to the joint and conditional probabilities, then with the help of the product rule for logarithms we get:

$$\log P(y_1, ...y_t | \mathbf{x}) = \sum_{t=1}^{N} \log P(y_t | y_{<t}, \mathbf{x})$$

In other words, the product of probabilities we saw earlier becomes a sum of log probabilities, which is much less likely to run into numerical instabilities. For example, calculating the log probability of the same example as before gives:

```
import numpy as np

sum([np.log(0.5)] * 1024)
```

```
-709.7827128933695
```

This is a number we can easily deal with, and this approach still works for much smaller numbers. Since we only want to compare relative probabilities, we can do this directly with log probabilities.

Let's calculate and compare the log probabilities of the texts generated by greedy and beam search to see if beam search can improve the overall probability. Since 🤗 Transformers models return the unnormalized logits for the next token given the input tokens, we first need to normalize the logits to create a probability distribution over the whole vocabulary for each token in the sequence. We then need to select only the token probabilities that were present in the sequence. The following function implements these steps:

```
import torch.nn.functional as F

def log_probs_from_logits(logits, labels):
    logp = F.log_softmax(logits, dim=-1)
    logp_label = torch.gather(logp, 2, labels.unsqueeze(2)).squeeze(-1)
    return logp_label
```

This gives us the log probability for a single token, so to get the total log probability of a sequence we just need to sum the log probabilities for each token:

```
def sequence_logprob(model, labels, input_len=0):
    with torch.no_grad():
        output = model(labels)
        log_probs = log_probs_from_logits(
            output.logits[:, :-1, :], labels[:, 1:])
        seq_log_prob = torch.sum(log_probs[:, input_len:])
    return seq_log_prob.cpu().numpy()
```

Note that we ignore the log probabilities of the input sequence because they are not generated by the model. We can also see that it is important to align the logits and the labels; since the model predicts the next token, we do not get a logit for the first label, and we don't need the last logit because we don't have a ground truth token for it.

Let's use these functions to first calculate the sequence log probability of the greedy decoder on the OpenAI prompt:

```
logp = sequence_logprob(model, output_greedy, input_len=len(input_ids[0]))
print(tokenizer.decode(output_greedy[0]))
print(f"\nlog-prob: {logp:.2f}")
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.
```

```
The researchers, from the University of California, Davis, and the University of
Colorado, Boulder, were conducting a study on the Andean cloud forest, which is
home to the rare species of cloud forest trees.


The researchers were surprised to find that the unicorns were able to
communicate with each other, and even with humans.


The researchers were surprised to find that the unicorns were able

log-prob: -87.43
```

Now let's compare this to a sequence that is generated with beam search. To activate beam search with the `generate()` function we just need to specify the number of beams with the `num_beams` parameter. The more beams we choose, the better the result potentially gets; however, the generation process becomes much slower since we generate parallel sequences for each beam:

```
output_beam = model.generate(input_ids, max_length=max_length, num_beams=5,
                             do_sample=False)
logp = sequence_logprob(model, output_beam, input_len=len(input_ids[0]))
print(tokenizer.decode(output_beam[0]))
print(f"\nlog-prob: {logp:.2f}")
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.


The discovery of the unicorns was made by a team of scientists from the
University of California, Santa Cruz, and the National Geographic Society.


The scientists were conducting a study of the Andes Mountains when they
discovered a herd of unicorns living in a remote, previously unexplored valley,
in the Andes Mountains. Even more surprising to the researchers was the fact
that the unicorns spoke perfect English

log-prob: -55.23
```

We can see that we get a better log probability (higher is better) with beam search than we did with simple greedy decoding. However, we can see that beam search also suffers from repetitive text. One way to address this is to impose an n-gram penalty with the `no_repeat_ngram_size` parameter that tracks which n-grams have been seen and sets the next token probability to zero if it would produce a previously seen n-gram:

```
output_beam = model.generate(input_ids, max_length=max_length, num_beams=5,
                             do_sample=False, no_repeat_ngram_size=2)
logp = sequence_logprob(model, output_beam, input_len=len(input_ids[0]))
print(tokenizer.decode(output_beam[0]))
print(f"\nlog-prob: {logp:.2f}")
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.


The discovery was made by a team of scientists from the University of
California, Santa Cruz, and the National Geographic Society.

According to a press release, the scientists were conducting a survey of the
area when they came across the herd. They were surprised to find that they were
able to converse with the animals in English, even though they had never seen a
unicorn in person before. The researchers were

log-prob: -93.12
```

This isn't too bad! We've managed to stop the repetitions, and we can see that despite producing a lower score, the text remains coherent. Beam search with n-gram penalty is a good way to find a trade-off between focusing on high-probability tokens (with beam search) while reducing repetitions (with n-gram penalty), and it's commonly used in applications such as summarization or machine translation where factual correctness is important. When factual correctness is less important than the diversity of generated output, for instance in open-domain chitchat or story generation, another alternative to reduce repetitions while improving diversity is to use sampling. Let's round out our exploration of text generation by examining a few of the most common sampling methods.

Sampling Methods

The simplest sampling method is to randomly sample from the probability distribution of the model's outputs over the full vocabulary at each timestep:

$$P(y_t = w_i | y_{<t}, \mathbf{x}) = \text{softmax}(z_{t,i}) = \frac{\exp(z_{t,i})}{\sum_{j=1}^{|V|} \exp(z_{t,j})}$$

where $|V|$ denotes the cardinality of the vocabulary. We can easily control the diversity of the output by adding a temperature parameter T that rescales the logits before taking the softmax:

$$P(y_t = w_i | y_{<t}, \mathbf{x}) = \frac{\exp\left(z_{t,i}/T\right)}{\Sigma_{j=1}^{|V|} \exp\left(z_{t,j}/T\right)}$$

By tuning T we can control the shape of the probability distribution.[5] When $T \ll 1$, the distribution becomes peaked around the origin and the rare tokens are suppressed. On the other hand, when $T \gg 1$, the distribution flattens out and each token becomes equally likely. The effect of temperature on token probabilities is shown in Figure 5-5.

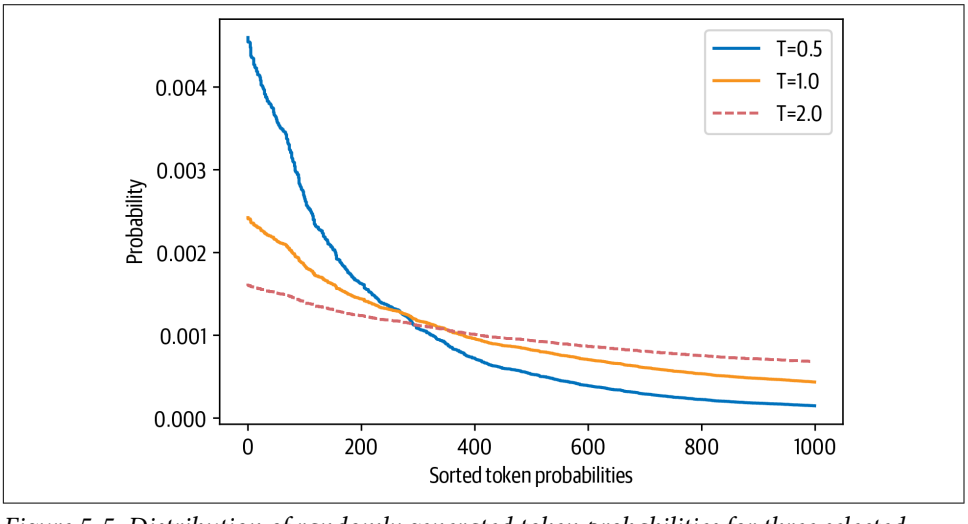


Figure 5-5. Distribution of randomly generated token probabilities for three selected temperatures

To see how we can use temperature to influence the generated text, let's sample with $T = 2$ by setting the `temperature` parameter in the `generate()` function (we'll explain the meaning of the `top_k` parameter in the next section):

```
output_temp = model.generate(input_ids, max_length=max_length, do_sample=True,
                             temperature=2.0, top_k=0)
print(tokenizer.decode(output_temp[0]))
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.


While the station aren protagonist receive Pengala nostalgiates tidbitRegarding
```

5 If you know some physics, you may recognize a striking resemblance to the Boltzmann distribution.

```
Jenny loclonju AgreementCON irrational ⁄rite Continent seaf A jer Turner
Dorbecue WILL Pumpkin mere Thatvernuildagain YoAniamond disse *
Runewitingkusstemprop});b zo coachinginventorymodules deflation press
Vaticanpres Wrestling chargesThingsctureddong Ty physician PET KimBi66 graz Oz
at aff da temporou MD6 radi iter
```

We can clearly see that a high temperature has produced mostly gibberish; by accentuating the rare tokens, we've caused the model to create strange grammar and quite a few made-up words! Let's see what happens if we cool down the temperature:

```
output_temp = model.generate(input_ids, max_length=max_length, do_sample=True,
                             temperature=0.5, top_k=0)
print(tokenizer.decode(output_temp[0]))
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.


The scientists were searching for the source of the mysterious sound, which was
making the animals laugh and cry.


The unicorns were living in a remote valley in the Andes mountains

'When we first heard the noise of the animals, we thought it was a lion or a
tiger,' said Luis Guzman, a researcher from the University of Buenos Aires,
Argentina.


'But when
```

This is significantly more coherent, and even includes a quote from yet another university being credited with the discovery! The main lesson we can draw from temperature is that it allows us to control the quality of the samples, but there's always a trade-off between coherence (low temperature) and diversity (high temperature) that one has to tune to the use case at hand.

Another way to adjust the trade-off between coherence and diversity is to truncate the distribution of the vocabulary. This allows us to adjust the diversity freely with the temperature, but in a more limited range that excludes words that would be too strange in the context (i.e., low-probability words). There are two main ways to do this: top-*k* and nucleus (or top-*p*) sampling. Let's take a look.

Top-k and Nucleus Sampling

Top-*k* and nucleus (top-*p*) sampling are two popular alternatives or extensions to using temperature. In both cases, the basic idea is to restrict the number of possible tokens we can sample from at each timestep. To see how this works, let's first visualize

the cumulative probability distribution of the model's outputs at $T = 1$ as seen in Figure 5-6.

Let's tease apart these plots, since they contain a lot of information. In the upper plot we can see a histogram of the token probabilities. It has a peak around 10^{-8} and a second, smaller peak around 10^{-4}, followed by a sharp drop with just a handful of tokens occurring with probability between 10^{-2} and 10^{-1}. Looking at this diagram, we can see that the probability of picking the token with the highest probability (the isolated bar at 10^{-1}) is 1 in 10.

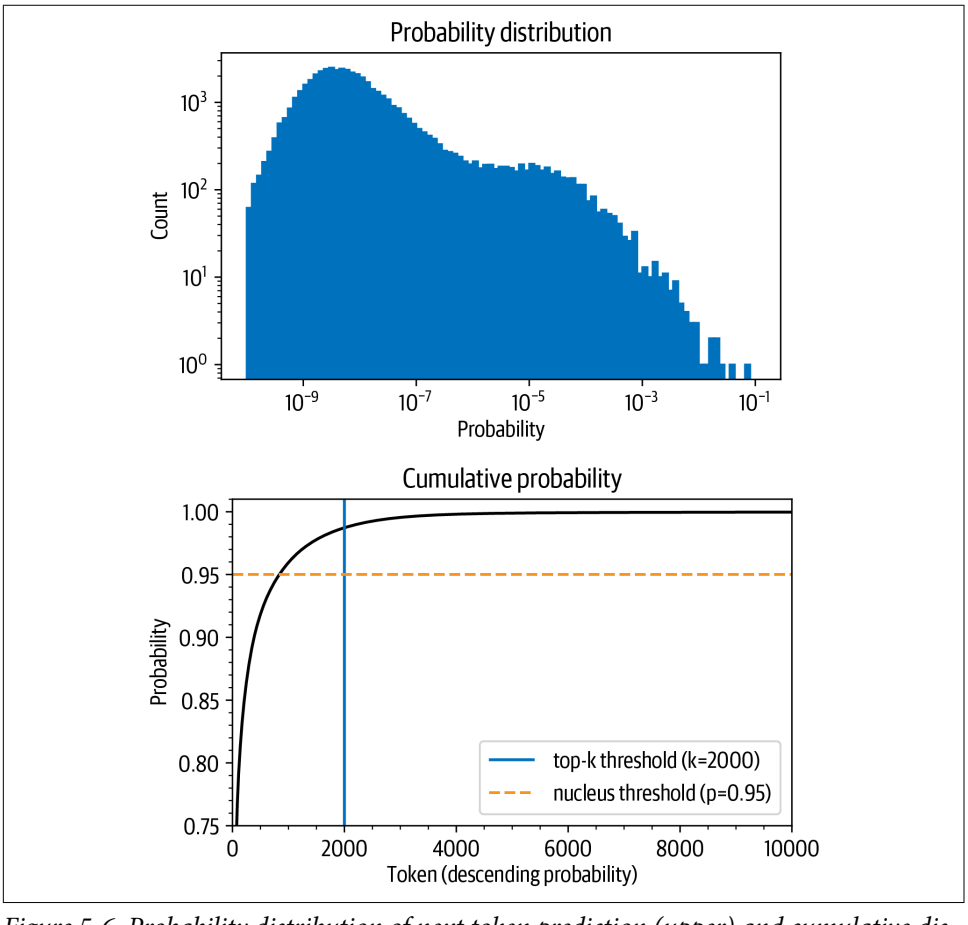


Figure 5-6. Probability distribution of next token prediction (upper) and cumulative distribution of descending token probabilities (lower)

In the lower plot, we've ordered the tokens by descending probability and calculated the cumulative sum of the first 10,000 tokens (in total, there are 50,257 tokens in GPT-2's vocabulary). The curved line represents the probability of picking any of the

preceding tokens. For example, there is roughly a 96% chance of picking any of the 1,000 tokens with the highest probability. We see that the probability rises quickly above 90% but saturates to close to 100% only after several thousand tokens. The plot shows that there is a 1 in 100 chance of not picking any of the tokens that are not even in the top 2,000.

Although these numbers might appear small at first sight, they become important because we sample once per token when generating text. So even if there is only a 1 in 100 or 1,000 chance, if we sample hundreds of times there is a significant chance of picking an unlikely token at some point—and picking such tokens when sampling can badly influence the quality of the generated text. For this reason, we generally want to avoid these very unlikely tokens. This is where top-*k* and top-*p* sampling come into play.

The idea behind top-*k* sampling is to avoid the low-probability choices by only sampling from the *k* tokens with the highest probability. This puts a fixed cut on the long tail of the distribution and ensures that we only sample from likely choices. Going back to Figure 5-6, top-*k* sampling is equivalent to defining a vertical line and sampling from the tokens on the left. Again, the `generate()` function provides an easy method to achieve this with the `top_k` argument:

```
output_topk = model.generate(input_ids, max_length=max_length, do_sample=True,
                             top_k=50)
print(tokenizer.decode(output_topk[0]))
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.


The wild unicorns roam the Andes Mountains in the region of Cajamarca, on the
border with Argentina (Picture: Alamy/Ecole Nationale Supérieure d'Histoire
Naturelle)

The researchers came across about 50 of the animals in the valley. They had
lived in such a remote and isolated area at that location for nearly a thousand
years that
```

This is arguably the most human-looking text we've generated so far. But how do we choose *k*? The value of *k* is chosen manually and is the same for each choice in the sequence, independent of the actual output distribution. We can find a good value for *k* by looking at some text quality metrics, which we will explore in the next chapter—but that fixed cutoff might not be very satisfactory.

An alternative is to use a *dynamic* cutoff. With nucleus or top-*p* sampling, instead of choosing a fixed cutoff value, we set a condition of when to cut off. This condition is when a certain probability mass in the selection is reached. Let's say we set that value

to 95%. We then order all tokens in descending order by probability and add one token after another from the top of the list until the sum of the probabilities of the selected tokens is 95%. Returning to Figure 5-6, the value for p defines a horizontal line on the cumulative sum of probabilities plot, and we sample only from tokens below the line. Depending on the output distribution, this could be just one (very likely) token or a hundred (more equally likely) tokens. At this point, you are probably not surprised that the `generate()` function also provides an argument to activate top-p sampling. Let's try it out:

```
output_topp = model.generate(input_ids, max_length=max_length, do_sample=True,
                             top_p=0.90)
print(tokenizer.decode(output_topp[0]))
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.


The scientists studied the DNA of the animals and came to the conclusion that
the herd are descendants of a prehistoric herd that lived in Argentina about
50,000 years ago.


According to the scientific analysis, the first humans who migrated to South
America migrated into the Andes Mountains from South Africa and Australia, after
the last ice age had ended.


Since their migration, the animals have been adapting to
```

Top-p sampling has also produced a coherent story, and this time with a new twist about migrations from Australia to South America. You can even combine the two sampling approaches to get the best of both worlds. Setting `top_k=50` and `top_p=0.9` corresponds to the rule of choosing tokens with a probability mass of 90%, from a pool of at most 50 tokens.

We can also apply beam search when we use sampling. Instead of selecting the next batch of candidate tokens greedily, we can sample them and build up the beams in the same way.

Which Decoding Method Is Best?

Unfortunately, there is no universally "best" decoding method. Which approach is best will depend on the nature of the task you are generating text for. If you want your model to perform a precise task like arithmetic or providing an answer to a specific question, then you should lower the temperature or use deterministic methods like greedy search in combination with beam search to guarantee getting the most likely answer. If you want the model to generate longer texts and even be a bit creative, then you should switch to sampling methods and increase the temperature or use a mix of top-*k* and nucleus sampling.

Conclusion

In this chapter we looked at text generation, which is a very different task from the NLU tasks we encountered previously. Generating text requires at least one forward pass per generated token, and even more if we use beam search. This makes text generation computationally demanding, and one needs the right infrastructure to run a text generation model at scale. In addition, a good decoding strategy that transforms the model's output probabilities into discrete tokens can improve the text quality. Finding the best decoding strategy requires some experimentation and a subjective evaluation of the generated texts.

In practice, however, we don't want to make these decisions based on gut feeling alone! Like with other NLP tasks, we should choose a model performance metric that reflects the problem we want to solve. Unsurprisingly, there are a wide range of choices, and we will encounter the most common ones in the next chapter, where we have a look at how to train and evaluate a model for text summarization. Or, if you can't wait to learn how to train a GPT-type model from scratch, you can skip right to Chapter 10, where we collect a large dataset of code and then train an autoregressive language model on it.

CHAPTER 6

Summarization

At one point or another, you've probably needed to summarize a document, be it a research article, a financial earnings report, or a thread of emails. If you think about it, this requires a range of abilities, such as understanding long passages, reasoning about the contents, and producing fluent text that incorporates the main topics from the original document. Moreover, accurately summarizing a news article is very different from summarizing a legal contract, so being able to do so requires a sophisticated degree of domain generalization. For these reasons, text summarization is a difficult task for neural language models, including transformers. Despite these challenges, text summarization offers the prospect for domain experts to significantly speed up their workflows and is used by enterprises to condense internal knowledge, summarize contracts, automatically generate content for social media releases, and more.

To help you understand the challenges involved, this chapter will explore how we can leverage pretrained transformers to summarize documents. Summarization is a classic sequence-to-sequence (seq2seq) task with an input text and a target text. As we saw in Chapter 1, this is where encoder-decoder transformers excel.

In this chapter we will build our own encoder-decoder model to condense dialogues between several people into a crisp summary. But before we get to that, let's begin by taking a look at one of the canonical datasets for summarization: the CNN/DailyMail corpus.

The CNN/DailyMail Dataset

The CNN/DailyMail dataset consists of around 300,000 pairs of news articles and their corresponding summaries, composed from the bullet points that CNN and the DailyMail attach to their articles. An important aspect of the dataset is that the

summaries are *abstractive* and not *extractive*, which means that they consist of new sentences instead of simple excerpts. The dataset is available on the Hub; we'll use version 3.0.0, which is a nonanonymized version set up for summarization. We can select versions in a similar manner as splits, we saw in Chapter 4, with a `version` keyword. So let's dive in and have a look at it:

```
from datasets import load_dataset

dataset = load_dataset("cnn_dailymail", version="3.0.0")
print(f"Features: {dataset['train'].column_names}")
```

```
Features: ['article', 'highlights', 'id']
```

The dataset has three columns: `article`, which contains the news articles, `highlights` with the summaries, and `id` to uniquely identify each article. Let's look at an excerpt from an article:

```
sample = dataset["train"][1]
print(f"""
Article (excerpt of 500 characters, total length: {len(sample["article"])}):
""")
print(sample["article"][:500])
print(f'\nSummary (length: {len(sample["highlights"])}):')
print(sample["highlights"])
```

```
Article (excerpt of 500 characters, total length: 3192):

(CNN) -- Usain Bolt rounded off the world championships Sunday by claiming his
third gold in Moscow as he anchored Jamaica to victory in the men's 4x100m
relay. The fastest man in the world charged clear of United States rival Justin
Gatlin as the Jamaican quartet of Nesta Carter, Kemar Bailey-Cole, Nickel
Ashmeade and Bolt won in 37.36 seconds. The U.S finished second in 37.56 seconds
with Canada taking the bronze after Britain were disqualified for a faulty
handover. The 26-year-old Bolt has n

Summary (length: 180):
Usain Bolt wins third gold of world championship .
Anchors Jamaica to 4x100m relay victory .
Eighth gold at the championships for Bolt .
Jamaica double up in women's 4x100m relay .
```

We see that the articles can be very long compared to the target summary; in this particular case the difference is 17-fold. Long articles pose a challenge to most transformer models since the context size is usually limited to 1,000 tokens or so, which is equivalent to a few paragraphs of text. The standard, yet crude way to deal with this for summarization is to simply truncate the texts beyond the model's context size. Obviously there could be important information for the summary toward the end of the text, but for now we need to live with this limitation of the model architectures.

Text Summarization Pipelines

Let's see how a few of the most popular transformer models for summarization perform by first looking qualitatively at the outputs for the preceding example. Although the model architectures we will be exploring have varying maximum input sizes, let's restrict the input text to 2,000 characters to have the same input for all models and thus make the outputs more comparable:

```
sample_text = dataset["train"][1]["article"][:2000]
# We'll collect the generated summaries of each model in a dictionary
summaries = {}
```

A convention in summarization is to separate the summary sentences by a newline. We could add a newline token after each full stop, but this simple heuristic would fail for strings like "U.S." or "U.N." The Natural Language Toolkit (NLTK) package includes a more sophisticated algorithm that can differentiate the end of a sentence from punctuation that occurs in abbreviations:

```
import nltk
from nltk.tokenize import sent_tokenize

nltk.download("punkt")
```

```
string = "The U.S. are a country. The U.N. is an organization."
sent_tokenize(string)
```

```
['The U.S. are a country.', 'The U.N. is an organization.']
```

In the following sections we will load several large models. If you run out of memory, you can either replace the large models with smaller checkpoints (e.g., "gpt", "t5-small") or skip this section and jump to "Evaluating PEGASUS on the CNN/DailyMail Dataset" on page 154.

Summarization Baseline

A common baseline for summarizing news articles is to simply take the first three sentences of the article. With NLTK's sentence tokenizer, we can easily implement such a baseline:

```
def three_sentence_summary(text):
    return "\n".join(sent_tokenize(text)[:3])
```

```
summaries["baseline"] = three_sentence_summary(sample_text)
```

GPT-2

We've already seen in Chapter 5 how GPT-2 can generate text given some prompt. One of the model's surprising features is that we can also use it to generate summaries by simply appending "TL;DR" at the end of the input text. The expression "TL;DR" (too long; didn't read) is often used on platforms like Reddit to indicate a short version of a long post. We will start our summarization experiment by re-creating the procedure of the original paper with the `pipeline()` function from 🤗 Transformers.[1] We create a text generation pipeline and load the large GPT-2 model:

```
from transformers import pipeline, set_seed

set_seed(42)
pipe = pipeline("text-generation", model="gpt2-xl")
gpt2_query = sample_text + "\nTL;DR:\n"
pipe_out = pipe(gpt2_query, max_length=512, clean_up_tokenization_spaces=True)
summaries["gpt2"] = "\n".join(
    sent_tokenize(pipe_out[0]["generated_text"][len(gpt2_query) :]))
```

Here we just store the summaries of the generated text by slicing off the input query and keep the result in a Python dictionary for later comparison.

T5

Next let's try the T5 transformer. As we saw in Chapter 3, the developers of this model performed a comprehensive study of transfer learning in NLP and found they could create a universal transformer architecture by formulating all tasks as text-to-text tasks. The T5 checkpoints are trained on a mixture of unsupervised data (to reconstruct masked words) and supervised data for several tasks, including summarization. These checkpoints can thus be directly used to perform summarization without fine-tuning by using the same prompts used during pretraining. In this framework, the input format for the model to summarize a document is `"summarize: <ARTICLE>"`, and for translation it looks like `"translate English to German: <TEXT>"`. As shown in Figure 6-1, this makes T5 extremely versatile and allows you to solve many tasks with a single model.

We can directly load T5 for summarization with the `pipeline()` function, which also takes care of formatting the inputs in the text-to-text format so we don't need to prepend them with `"summarize"`:

```
pipe = pipeline("summarization", model="t5-large")
pipe_out = pipe(sample_text)
summaries["t5"] = "\n".join(sent_tokenize(pipe_out[0]["summary_text"]))
```

1 A. Radford et al., "Language Models Are Unsupervised Multitask Learners", OpenAI (2019).

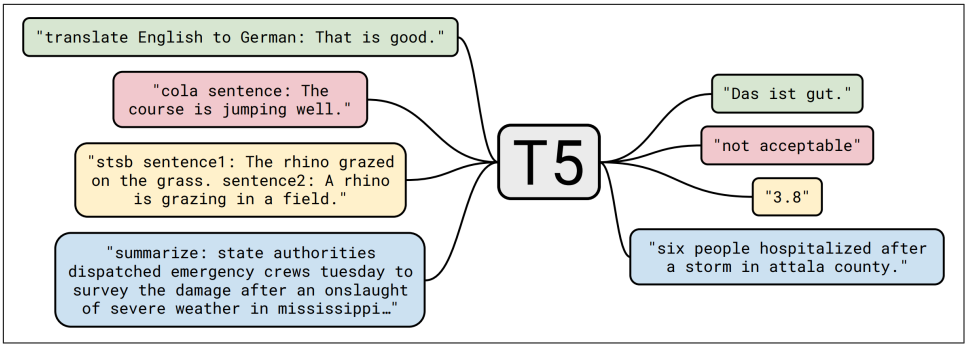


Figure 6-1. Diagram of T5's text-to-text framework (courtesy of Colin Raffel); besides translation and summarization, the CoLA (linguistic acceptability) and STSB (semantic similarity) tasks are shown

BART

BART also uses an encoder-decoder architecture and is trained to reconstruct corrupted inputs. It combines the pretraining schemes of BERT and GPT-2.[2] We'll use the `facebook/bart-large-ccn` checkpoint, which has been specifically fine-tuned on the CNN/DailyMail dataset:

```
pipe = pipeline("summarization", model="facebook/bart-large-cnn")
pipe_out = pipe(sample_text)
summaries["bart"] = "\n".join(sent_tokenize(pipe_out[0]["summary_text"]))
```

PEGASUS

Like BART, PEGASUS is an encoder-decoder transformer.[3] As shown in Figure 6-2, its pretraining objective is to predict masked sentences in multisentence texts. The authors argue that the closer the pretraining objective is to the downstream task, the more effective it is. With the aim of finding a pretraining objective that is closer to summarization than general language modeling, they automatically identified, in a very large corpus, sentences containing most of the content of their surrounding paragraphs (using summarization evaluation metrics as a heuristic for content overlap) and pretrained the PEGASUS model to reconstruct these sentences, thereby obtaining a state-of-the-art model for text summarization.

2 M. Lewis et al., "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension", (2019).

3 J. Zhang et al., "PEGASUS: Pre-Training with Extracted Gap-Sentences for Abstractive Summarization", (2019).

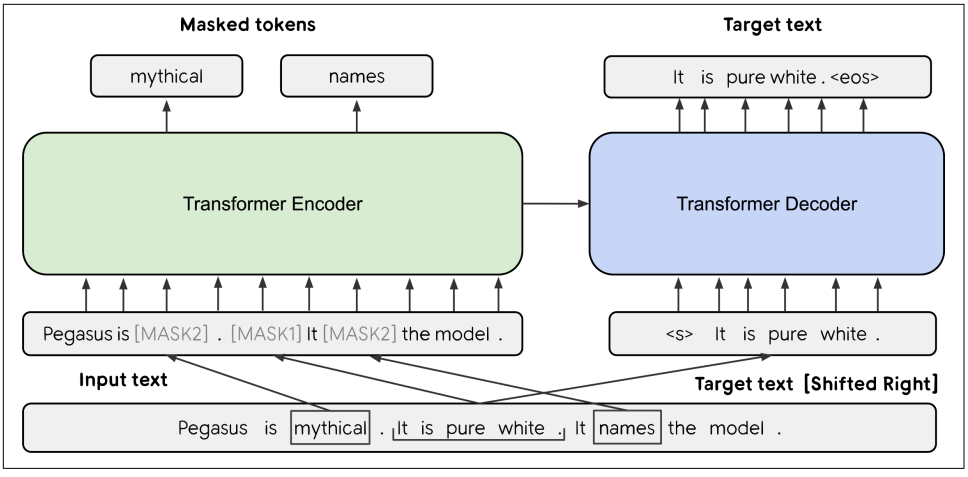


Figure 6-2. Diagram of PEGASUS architecture (courtesy of Jingqing Zhang et al.)

This model has a special token for newlines, which is why we don't need the `sent_tokenize()` function:

```
pipe = pipeline("summarization", model="google/pegasus-cnn_dailymail")
pipe_out = pipe(sample_text)
summaries["pegasus"] = pipe_out[0]["summary_text"].replace(" .<n>", ".\n")
```

Comparing Different Summaries

Now that we have generated summaries with four different models, let's compare the results. Keep in mind that one model has not been trained on the dataset at all (GPT-2), one model has been fine-tuned on this task among others (T5), and two models have exclusively been fine-tuned on this task (BART and PEGASUS). Let's have a look at the summaries these models have generated:

```
print("GROUND TRUTH")
print(dataset["train"][1]["highlights"])
print("")

for model_name in summaries:
    print(model_name.upper())
    print(summaries[model_name])
    print("")
```

```
GROUND TRUTH
Usain Bolt wins third gold of world championship .
Anchors Jamaica to 4x100m relay victory .
Eighth gold at the championships for Bolt .
Jamaica double up in women's 4x100m relay .

BASELINE
```

```
(CNN) -- Usain Bolt rounded off the world championships Sunday by claiming his
third gold in Moscow as he anchored Jamaica to victory in the men's 4x100m
relay.
The fastest man in the world charged clear of United States rival Justin Gatlin
as the Jamaican quartet of Nesta Carter, Kemar Bailey-Cole, Nickel Ashmeade and
Bolt won in 37.36 seconds.
The U.S finished second in 37.56 seconds with Canada taking the bronze after
Britain were disqualified for a faulty handover.

GPT2
Nesta, the fastest man in the world.
Gatlin, the most successful Olympian ever.
Kemar, a Jamaican legend.
Shelly-Ann, the fastest woman ever.
Bolt, the world's greatest athlete.
The team sport of pole vaulting

T5
usain bolt wins his third gold medal of the world championships in the men's
4x100m relay .
the 26-year-old anchored Jamaica to victory in the event in the Russian capital
.
he has now collected eight gold medals at the championships, equaling the record
.

BART
Usain Bolt wins his third gold of the world championships in Moscow.
Bolt anchors Jamaica to victory in the men's 4x100m relay.
The 26-year-old has now won eight gold medals at world championships.
Jamaica's women also win gold in the relay, beating France in the process.

PEGASUS
Usain Bolt wins third gold of world championships.
Anchors Jamaica to victory in men's 4x100m relay.
Eighth gold at the championships for Bolt.
Jamaica also win women's 4x100m relay .
```

The first thing we notice by looking at the model outputs is that the summary generated by GPT-2 is quite different from the others. Instead of giving a summary of the text, it summarizes the characters. Often the GPT-2 model "hallucinates" or invents facts, since it was not explicitly trained to generate truthful summaries. For example, at the time of writing, Nesta is not the fastest man in the world, but sits in ninth place. Comparing the other three model summaries against the ground truth, we see that there is remarkable overlap, with PEGASUS's output bearing the most striking resemblance.

Now that we have inspected a few models, let's try to decide which one we would use in a production setting. All four models seem to provide qualitatively reasonable results, and we could generate a few more examples to help us decide. However, this is not a systematic way of determining the best model! Ideally, we would define a

metric, measure it for all models on some benchmark dataset, and choose the one with the best performance. But how do you define a metric for text generation? The standard metrics that we've seen, like accuracy, recall, and precision, are not easy to apply to this task. For each "gold standard" summary written by a human, dozens of other summaries with synonyms, paraphrases, or a slightly different way of formulating the facts could be just as acceptable.

In the next section we will look at some common metrics that have been developed for measuring the quality of generated text.

Measuring the Quality of Generated Text

Good evaluation metrics are important, since we use them to measure the performance of models not only when we train them but also later, in production. If we have bad metrics we might be blind to model degradation, and if they are misaligned with the business goals we might not create any value.

Measuring performance on a text generation task is not as easy as with standard classification tasks such as sentiment analysis or named entity recognition. Take the example of translation; given a sentence like "I love dogs!" in English and translating it to Spanish there can be multiple valid possibilities, like "¡Me encantan los perros!" or "¡Me gustan los perros!" Simply checking for an exact match to a reference translation is not optimal; even humans would fare badly on such a metric because we all write text slightly differently from each other (and even from ourselves, depending on the time of the day or year!). Fortunately, there are alternatives.

Two of the most common metrics used to evaluate generated text are BLEU and ROUGE. Let's take a look at how they're defined.

BLEU

The idea of BLEU is simple:[4] instead of looking at how many of the tokens in the generated texts are perfectly aligned with the reference text tokens, we look at words or *n*-grams. BLEU is a precision-based metric, which means that when we compare the two texts we count the number of words in the generation that occur in the reference and divide it by the length of the reference.

However, there is an issue with this vanilla precision. Assume the generated text just repeats the same word over and over again, and this word also appears in the reference. If it is repeated as many times as the length of the reference text, then we get

4 K. Papineni et al., "BLEU: A Method for Automatic Evaluation of Machine Translation," *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics* (July 2002): 311–318, http://dx.doi.org/10.3115/1073083.1073135.

perfect precision! For this reason, the authors of the BLEU paper introduced a slight modification: a word is only counted as many times as it occurs in the reference. To illustrate this point, suppose we have the reference text "the cat is on the mat" and the generated text "the the the the the the".

From this simple example, we can calculate the precision values as follows:

$$p_{vanilla} = \frac{6}{6}$$

$$p_{mod} = \frac{2}{6}$$

and we can see that the simple correction has produced a much more reasonable value. Now let's extend this by not only looking at single words, but *n*-grams as well. Let's assume we have one generated sentence, *snt*, that we want to compare against a reference sentence, *snt′*. We extract all possible *n*-grams of degree *n* and do the accounting to get the precision p_n:

$$p_n = \frac{\Sigma_{n\text{-}gram \in snt} Count_{clip}(n\text{-}gram)}{\Sigma_{n\text{-}gram \in snt'} Count(n\text{-}gram)}$$

In order to avoid rewarding repetitive generations, the count in the numerator is clipped. What this means is that the occurrence count of an *n*-gram is capped at how many times it appears in the reference sentence. Also note that the definition of a sentence is not very strict in this equation, and if you had a generated text spanning multiple sentences you would treat it as one sentence.

In general we have more than one sample in the test set we want to evaluate, so we need to slightly extend the equation by summing over all samples in the corpus *C*:

$$p_n = \frac{\Sigma_{snt \in C} \Sigma_{n\text{-}gram \in snt} Count_{clip}(n\text{-}gram)}{\Sigma_{snt' \in C} \Sigma_{n\text{-}gram \in snt'} Count(n\text{-}gram)}$$

We're almost there. Since we are not looking at recall, all generated sequences that are short but precise have a benefit compared to sentences that are longer. Therefore, the precision score favors short generations. To compensate for that the authors of BLEU introduced an additional term, the *brevity penalty*:

$$BR = \min\left(1, e^{1 - \ell_{ref}/\ell_{gen}}\right)$$

By taking the minimum, we ensure that this penalty never exceeds 1 and the exponential term becomes exponentially small when the length of the generated text l_{gen} is smaller than the reference text l_{ref}. At this point you might ask, why don't we just use something like an F_1-score to account for recall as well? The answer is that often in translation datasets there are multiple reference sentences instead of just one, so if we also measured recall we would incentivize translations that used all the words from all the references. Therefore, it's preferable to look for high precision in the translation and make sure the translation and reference have a similar length.

Finally, we can put everything together and get the equation for the BLEU score:

$$\text{BLEU-}N = BR \times \left(\prod_{n=1}^{N} p_n \right)^{1/N}$$

The last term is the geometric mean of the modified precision up to n-gram N. In practice, the BLEU-4 score is often reported. However, you can probably already see that this metric has many limitations; for instance, it doesn't take synonyms into account, and many steps in the derivation seem like ad hoc and rather fragile heuristics. You can find a wonderful exposition of BLEU's flaws in Rachel Tatman's blog post "Evaluating Text Output in NLP: BLEU at Your Own Risk".

In general, the field of text generation is still looking for better evaluation metrics, and finding ways to overcome the limits of metrics like BLEU is an active area of research. Another weakness of the BLEU metric is that it expects the text to already be tokenized. This can lead to varying results if the exact same method for text tokenization is not used. The SacreBLEU metric addresses this issue by internalizing the tokenization step; for this reason, it is the preferred metric for benchmarking.

We've now worked through some theory, but what we really want to do is calculate the score for some generated text. Does that mean we need to implement all this logic in Python? Fear not, 🤗 Datasets also provides metrics! Loading a metric works just like loading a dataset:

```
from datasets import load_metric

bleu_metric = load_metric("sacrebleu")
```

The `bleu_metric` object is an instance of the `Metric` class, and works like an aggregator: you can add single instances with `add()` or whole batches via `add_batch()`. Once you have added all the samples you need to evaluate, you then call `compute()` and the metric is calculated. This returns a dictionary with several values, such as the precision for each n-gram, the length penalty, as well as the final BLEU score. Let's look at the example from before:

```
import pandas as pd
import numpy as np

bleu_metric.add(
    prediction="the the the the the the", reference=["the cat is on the mat"])
results = bleu_metric.compute(smooth_method="floor", smooth_value=0)
results["precisions"] = [np.round(p, 2) for p in results["precisions"]]
pd.DataFrame.from_dict(results, orient="index", columns=["Value"])
```

	Value
score	0.0
counts	[2, 0, 0, 0]
totals	[6, 5, 4, 3]
precisions	[33.33, 0.0, 0.0, 0.0]
bp	1.0
sys_len	6
ref_len	6

The BLEU score also works if there are multiple reference translations. This is why `reference` is passed as a list. To make the metric smoother for zero counts in the *n*-grams, BLEU integrates methods to modify the precision calculation. One method is to add a constant to the numerator. That way, a missing *n*-gram does not cause the score to automatically go to zero. For the purpose of explaining the values, we turn it off by setting `smooth_value=0`.

We can see the precision of the 1-gram is indeed 2/6, whereas the precisions for the 2/3/4-grams are all 0. (For more information about the individual metrics, like counts and bp, see the SacreBLEU repository.) This means the geometric mean is zero, and thus also the BLEU score. Let's look at another example where the prediction is almost correct:

```
bleu_metric.add(
    prediction="the cat is on mat", reference=["the cat is on the mat"])
results = bleu_metric.compute(smooth_method="floor", smooth_value=0)
results["precisions"] = [np.round(p, 2) for p in results["precisions"]]
pd.DataFrame.from_dict(results, orient="index", columns=["Value"])
```

	Value
score	57.893007
counts	[5, 3, 2, 1]
totals	[5, 4, 3, 2]
precisions	[100.0, 75.0, 66.67, 50.0]

	Value
bp	0.818731
sys_len	5
ref_len	6

We observe that the precision scores are much better. The 1-grams in the prediction all match, and only in the precision scores do we see that something is off. For the 4-gram there are only two candidates, `["the", "cat", "is", "on"]` and `["cat", "is", "on", "mat"]`, where the last one does not match, hence the precision of 0.5.

The BLEU score is widely used for evaluating text, especially in machine translation, since precise translations are usually favored over translations that include all possible and appropriate words.

There are other applications, such as summarization, where the situation is different. There, we want all the important information in the generated text, so we favor high recall. This is where the ROUGE score is usually used.

ROUGE

The ROUGE score was specifically developed for applications like summarization where high recall is more important than just precision.[5] The approach is very similar to the BLEU score in that we look at different *n*-grams and compare their occurrences in the generated text and the reference texts. The difference is that with ROUGE we check how many *n*-grams in the reference text also occur in the generated text. For BLEU we looked at how many *n*-grams in the generated text appear in the reference, so we can reuse the precision formula with the minor modification that we count the (unclipped) occurrence of reference *n*-grams in the generated text in the numerator:

$$\text{ROUGE-}N = \frac{\sum_{\text{snt'} \in C} \sum_{n\text{-}gram \in snt'} Count_{match}(n\text{-}gram)}{\sum_{\text{snt'} \in C} \sum_{n\text{-}gram \in snt'} Count(n\text{-}gram)}$$

This was the original proposal for ROUGE. Subsequently, researchers have found that fully removing precision can have strong negative effects. Going back to the BLEU formula without the clipped counting, we can measure precision as well, and we can then combine both precision and recall ROUGE scores in the harmonic mean to get an F_1-score. This score is the metric that is nowadays commonly reported for ROUGE.

5 C-Y. Lin, "ROUGE: A Package for Automatic Evaluation of Summaries," *Text Summarization Branches Out* (July 2004), *https://aclanthology.org/W04-1013.pdf*.

There is a separate score in ROUGE to measure the longest common substring (LCS), called ROUGE-L. The LCS can be calculated for any pair of strings. For example, the LCS for "abab" and "abc" would be "ab", and its the length would be 2. If we want to compare this value between two samples we need to somehow normalize it because otherwise a longer text would be at an advantage. To achieve this, the inventor of ROUGE came up with an *F*-score-like scheme where the LCS is normalized with the length of the reference and generated text, then the two normalized scores are mixed together:

$$R_{LCS} = \frac{LCS(X, Y)}{m}$$

$$P_{LCS} = \frac{LCS(X, Y)}{n}$$

$$F_{LCS} = \frac{\left(1 + \beta^2\right) R_{LCS} P_{LCS}}{R_{LCS} + \beta P_{LCS}}, \text{where } \beta = P_{LCS} / R_{LCS}$$

That way the LCS score is properly normalized and can be compared across samples. In the 🤗 Datasets implementation, two variations of ROUGE are calculated: one calculates the score per sentence and averages it for the summaries (ROUGE-L), and the other calculates it directly over the whole summary (ROUGE-Lsum).

We can load the metric as follows:

```
rouge_metric = load_metric("rouge")
```

We already generated a set of summaries with GPT-2 and the other models, and now we have a metric to compare the summaries systematically. Let's apply the ROUGE score to all the summaries generated by the models:

```
reference = dataset["train"][1]["highlights"]
records = []
rouge_names = ["rouge1", "rouge2", "rougeL", "rougeLsum"]

for model_name in summaries:
    rouge_metric.add(prediction=summaries[model_name], reference=reference)
    score = rouge_metric.compute()
    rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names)
    records.append(rouge_dict)
pd.DataFrame.from_records(records, index=summaries.keys())
```

	rouge1	rouge2	rougeL	rougeLsum
baseline	0.303571	0.090909	0.214286	0.232143
gpt2	0.187500	0.000000	0.125000	0.187500

	rouge1	rouge2	rougeL	rougeLsum
t5	0.486486	0.222222	0.378378	0.486486
bart	0.582278	0.207792	0.455696	0.506329
pegasus	0.866667	0.655172	0.800000	0.833333

The ROUGE metric in the 🤗 Datasets library also calculates confidence intervals (by default, the 5th and 95th percentiles). The average value is stored in the attribute `mid` and the interval can be retrieved with `low` and `high`.

These results are obviously not very reliable as we only looked at a single sample, but we can compare the quality of the summary for that one example. The table confirms our observation that of the models we considered, GPT-2 performs worst. This is not surprising since it is the only model of the group that was not explicitly trained to summarize. It is striking, however, that the simple first-three-sentence baseline doesn't fare too poorly compared to the transformer models that have on the order of a billion parameters! PEGASUS and BART are the best models overall (higher ROUGE scores are better), but T5 is slightly better on ROUGE-1 and the LCS scores. These results place T5 and PEGASUS as the best models, but again these results should be treated with caution as we only evaluated the models on a single example. Looking at the results in the PEGASUS paper, we would expect the PEGASUS to outperform T5 on the CNN/DailyMail dataset.

Let's see if we can reproduce those results with PEGASUS.

Evaluating PEGASUS on the CNN/DailyMail Dataset

We now have all the pieces in place to evaluate the model properly: we have a dataset with a test set from CNN/DailyMail, we have a metric with ROUGE, and we have a summarization model. We just need to put the pieces together. Let's first evaluate the performance of the three-sentence baseline:

```
def evaluate_summaries_baseline(dataset, metric,
                                column_text="article",
                                column_summary="highlights"):
    summaries = [three_sentence_summary(text) for text in dataset[column_text]]
    metric.add_batch(predictions=summaries,
                     references=dataset[column_summary])
    score = metric.compute()
    return score
```

Now we'll apply the function to a subset of the data. Since the test fraction of the CNN/DailyMail dataset consists of roughly 10,000 samples, generating summaries for all these articles takes a lot of time. Recall from Chapter 5 that every generated token

requires a forward pass through the model; generating just 100 tokens for each sample will thus require 1 million forward passes, and if we use beam search this number is multiplied by the number of beams. For the purpose of keeping the calculations relatively fast, we'll subsample the test set and run the evaluation on 1,000 samples instead. This should give us a much more stable score estimation while completing in less than one hour on a single GPU for the PEGASUS model:

```
test_sampled = dataset["test"].shuffle(seed=42).select(range(1000))

score = evaluate_summaries_baseline(test_sampled, rouge_metric)
rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names)
pd.DataFrame.from_dict(rouge_dict, orient="index", columns=["baseline"]).T
```

	rouge1	rouge2	rougeL	rougeLsum
baseline	0.396061	0.173995	0.245815	0.361158

The scores are mostly worse than on the previous example, but still better than those achieved by GPT-2! Now let's implement the same evaluation function for evaluating the PEGASUS model:

```
from tqdm import tqdm
import torch

device = "cuda" if torch.cuda.is_available() else "cpu"

def chunks(list_of_elements, batch_size):
    """Yield successive batch-sized chunks from list_of_elements."""
    for i in range(0, len(list_of_elements), batch_size):
        yield list_of_elements[i : i + batch_size]

def evaluate_summaries_pegasus(dataset, metric, model, tokenizer,
                               batch_size=16, device=device,
                               column_text="article",
                               column_summary="highlights"):
    article_batches = list(chunks(dataset[column_text], batch_size))
    target_batches = list(chunks(dataset[column_summary], batch_size))

    for article_batch, target_batch in tqdm(
        zip(article_batches, target_batches), total=len(article_batches)):

        inputs = tokenizer(article_batch, max_length=1024,  truncation=True,
                        padding="max_length", return_tensors="pt")

        summaries = model.generate(input_ids=inputs["input_ids"].to(device),
                         attention_mask=inputs["attention_mask"].to(device),
                         length_penalty=0.8, num_beams=8, max_length=128)

        decoded_summaries = [tokenizer.decode(s, skip_special_tokens=True,
                                clean_up_tokenization_spaces=True)
               for s in summaries]
```

```
            decoded_summaries = [d.replace("<n>", " ") for d in decoded_summaries]
            metric.add_batch(predictions=decoded_summaries, references=target_batch)

        score = metric.compute()
        return score
```

Let's unpack this evaluation code a bit. First we split the dataset into smaller batches that we can process simultaneously. Then for each batch we tokenize the input articles and feed them to the `generate()` function to produce the summaries using beam search. We use the same generation parameters as proposed in the paper. The new parameter for length penalty ensures that the model does not generate sequences that are too long. Finally, we decode the generated texts, replace the `<n>` token, and add the decoded texts with the references to the metric. At the end, we compute and return the ROUGE scores. Let's now load the model again with the `AutoModelForSeq2SeqLM` class, used for seq2seq generation tasks, and evaluate it:

```
from transformers import AutoModelForSeq2SeqLM, AutoTokenizer

model_ckpt = "google/pegasus-cnn_dailymail"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
model = AutoModelForSeq2SeqLM.from_pretrained(model_ckpt).to(device)
score = evaluate_summaries_pegasus(test_sampled, rouge_metric,
                                   model, tokenizer, batch_size=8)
rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names)
pd.DataFrame(rouge_dict, index=["pegasus"])
```

	rouge1	rouge2	rougeL	rougeLsum
pegasus	0.434381	0.210883	0.307195	0.373231

These numbers are very close to the published results. One thing to note here is that the loss and per-token accuracy are decoupled to some degree from the ROUGE scores. The loss is independent of the decoding strategy, whereas the ROUGE score is strongly coupled.

Since ROUGE and BLEU correlate better with human judgment than loss or accuracy, we should focus on them and carefully explore and choose the decoding strategy when building text generation models. These metrics are far from perfect, however, and one should always consider human judgments as well.

Now that we're equipped with an evaluation function, it's time to train our own model for summarization.

Training a Summarization Model

We've worked through a lot of details on text summarization and evaluation, so let's put this to use to train a custom text summarization model! For our application, we'll use the SAMSum dataset developed by Samsung, which consists of a collection of dialogues along with brief summaries. In an enterprise setting, these dialogues might represent the interactions between a customer and the support center, so generating accurate summaries can help improve customer service and detect common patterns among customer requests. Let's load it and look at an example:

```
dataset_samsum = load_dataset("samsum")
split_lengths = [len(dataset_samsum[split])for split in dataset_samsum]

print(f"Split lengths: {split_lengths}")
print(f"Features: {dataset_samsum['train'].column_names}")
print("\nDialogue:")
print(dataset_samsum["test"][0]["dialogue"])
print("\nSummary:")
print(dataset_samsum["test"][0]["summary"])
```

```
Split lengths: [14732, 819, 818]
Features: ['id', 'dialogue', 'summary']

Dialogue:
Hannah: Hey, do you have Betty's number?
Amanda: Lemme check
Hannah: <file_gif>
Amanda: Sorry, can't find it.
Amanda: Ask Larry
Amanda: He called her last time we were at the park together
Hannah: I don't know him well
Hannah: <file_gif>
Amanda: Don't be shy, he's very nice
Hannah: If you say so..
Hannah: I'd rather you texted him
Amanda: Just text him 🙂
Hannah: Urgh.. Alright
Hannah: Bye
Amanda: Bye bye

Summary:
Hannah needs Betty's number but Amanda doesn't have it. She needs to contact
Larry.
```

The dialogues look like what you would expect from a chat via SMS or WhatsApp, including emojis and placeholders for GIFs. The `dialogue` field contains the full text and the `summary` the summarized dialogue. Could a model that was fine-tuned on the CNN/DailyMail dataset deal with that? Let's find out!

Evaluating PEGASUS on SAMSum

First we'll run the same summarization pipeline with PEGASUS to see what the output looks like. We can reuse the code we used for the CNN/DailyMail summary generation:

```
pipe_out = pipe(dataset_samsum["test"][0]["dialogue"])
print("Summary:")
print(pipe_out[0]["summary_text"].replace(" .<n>", ".\n"))
```

```
Summary:
Amanda: Ask Larry Amanda: He called her last time we were at the park together.
Hannah: I'd rather you texted him.
Amanda: Just text him .
```

We can see that the model mostly tries to summarize by extracting the key sentences from the dialogue. This probably worked relatively well on the CNN/DailyMail dataset, but the summaries in SAMSum are more abstract. Let's confirm this by running the full ROUGE evaluation on the test set:

```
score = evaluate_summaries_pegasus(dataset_samsum["test"], rouge_metric, model,
                                   tokenizer, column_text="dialogue",
                                   column_summary="summary", batch_size=8)

rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names)
pd.DataFrame(rouge_dict, index=["pegasus"])
```

	rouge1	rouge2	rougeL	rougeLsum
pegasus	0.296168	0.087803	0.229604	0.229514

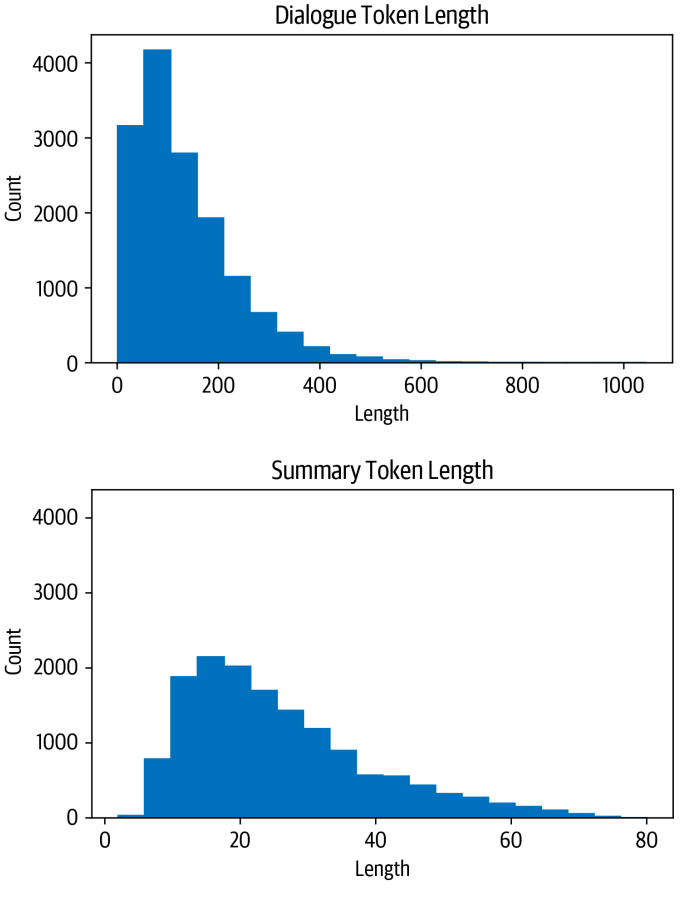
Well, the results aren't great, but this is not unexpected since we've moved quite a bit away from the CNN/DailyMail data distribution. Nevertheless, setting up the evaluation pipeline before training has two advantages: we can directly measure the success of training with the metric and we have a good baseline. Fine-tuning the model on our dataset should result in an immediate improvement in the ROUGE metric, and if that is not the case we'll know something is wrong with our training loop.

Fine-Tuning PEGASUS

Before we process the data for training, let's have a quick look at the length distribution of the input and outputs:

```
d_len = [len(tokenizer.encode(s)) for s in dataset_samsum["train"]["dialogue"]]
s_len = [len(tokenizer.encode(s)) for s in dataset_samsum["train"]["summary"]]

fig, axes = plt.subplots(1, 2, figsize=(10, 3.5), sharey=True)
axes[0].hist(d_len, bins=20, color="C0", edgecolor="C0")
axes[0].set_title("Dialogue Token Length")
axes[0].set_xlabel("Length")
```

```
axes[0].set_ylabel("Count")
axes[1].hist(s_len, bins=20, color="C0", edgecolor="C0")
axes[1].set_title("Summary Token Length")
axes[1].set_xlabel("Length")
plt.tight_layout()
plt.show()
```



We see that most dialogues are much shorter than the CNN/DailyMail articles, with 100–200 tokens per dialogue. Similarly, the summaries are much shorter, with around 20–40 tokens (the average length of a tweet).

Let's keep those observations in mind as we build the data collator for the `Trainer`. First we need to tokenize the dataset. For now, we'll set the maximum lengths to 1024 and 128 for the dialogues and summaries, respectively:

```
def convert_examples_to_features(example_batch):
    input_encodings = tokenizer(example_batch["dialogue"], max_length=1024,
                                truncation=True)
```

```
        with tokenizer.as_target_tokenizer():
            target_encodings = tokenizer(example_batch["summary"], max_length=128,
                                         truncation=True)

        return {"input_ids": input_encodings["input_ids"],
                "attention_mask": input_encodings["attention_mask"],
                "labels": target_encodings["input_ids"]}

    dataset_samsum_pt = dataset_samsum.map(convert_examples_to_features,
                                           batched=True)
    columns = ["input_ids", "labels", "attention_mask"]
    dataset_samsum_pt.set_format(type="torch", columns=columns)
```

A new thing in the use of the tokenization step is the `tokenizer.as_target_tokenizer()` context. Some models require special tokens in the decoder inputs, so it's important to differentiate between the tokenization of encoder and decoder inputs. In the `with` statement (called a *context manager*), the tokenizer knows that it is tokenizing for the decoder and can process sequences accordingly.

Now, we need to create the data collator. This function is called in the `Trainer` just before the batch is fed through the model. In most cases we can use the default collator, which collects all the tensors from the batch and simply stacks them. For the summarization task we need to not only stack the inputs but also prepare the targets on the decoder side. PEGASUS is an encoder-decoder transformer and thus has the classic seq2seq architecture. In a seq2seq setup, a common approach is to apply "teacher forcing" in the decoder. With this strategy, the decoder receives input tokens (like in decoder-only models such as GPT-2) that consists of the labels shifted by one in addition to the encoder output; so, when making the prediction for the next token the decoder gets the ground truth shifted by one as an input, as illustrated in the following table:

	decoder_input	label
step		
1	[PAD]	Transformers
2	[PAD, Transformers]	are
3	[PAD, Transformers, are]	awesome
4	[PAD, Transformers, are, awesome]	for
5	[PAD, Transformers, are, awesome, for]	text
6	[PAD, Transformers, are, awesome, for, text]	summarization

We shift it by one so that the decoder only sees the previous ground truth labels and not the current or future ones. Shifting alone suffices since the decoder has masked self-attention that masks all inputs at present and in the future.

So, when we prepare our batch, we set up the decoder inputs by shifting the labels to the right by one. After that, we make sure the padding tokens in the labels are ignored by the loss function by setting them to –100. We actually don't have to do this manually, though, since the `DataCollatorForSeq2Seq` comes to the rescue and takes care of all these steps for us:

```
from transformers import DataCollatorForSeq2Seq

seq2seq_data_collator = DataCollatorForSeq2Seq(tokenizer, model=model)
```

Then, as usual, we set up a the `TrainingArguments` for training:

```
from transformers import TrainingArguments, Trainer

training_args = TrainingArguments(
    output_dir='pegasus-samsum', num_train_epochs=1, warmup_steps=500,
    per_device_train_batch_size=1, per_device_eval_batch_size=1,
    weight_decay=0.01, logging_steps=10, push_to_hub=True,
    evaluation_strategy='steps', eval_steps=500, save_steps=1e6,
    gradient_accumulation_steps=16)
```

One thing that is different from the previous settings is that new argument, `gradient_accumulation_steps`. Since the model is quite big, we had to set the batch size to 1. However, a batch size that is too small can hurt convergence. To resolve that issue, we can use a nifty technique called *gradient accumulation*. As the name suggests, instead of calculating the gradients of the full batch all at once, we make smaller batches and aggregate the gradients. When we have aggregated enough gradients, we run the optimization step. Naturally this is a bit slower than doing it in one pass, but it saves us a lot of GPU memory.

Let's now make sure that we are logged in to Hugging Face so we can push the model to the Hub after training:

```
from huggingface_hub import notebook_login

notebook_login()
```

We have now everything we need to initialize the trainer with the model, tokenizer, training arguments, and data collator, as well as the training and evaluation sets:

```
trainer = Trainer(model=model, args=training_args,
                  tokenizer=tokenizer, data_collator=seq2seq_data_collator,
                  train_dataset=dataset_samsum_pt["train"],
                  eval_dataset=dataset_samsum_pt["validation"])
```

We are ready for training. After training, we can directly run the evaluation function on the test set to see how well the model performs:

```
trainer.train()
score = evaluate_summaries_pegasus(
    dataset_samsum["test"], rouge_metric, trainer.model, tokenizer,
    batch_size=2, column_text="dialogue", column_summary="summary")
```

```
rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names)
pd.DataFrame(rouge_dict, index=[f"pegasus"])
```

	rouge1	rouge2	rougeL	rougeLsum
pegasus	0.427614	0.200571	0.340648	0.340738

We see that the ROUGE scores improved considerably over the model without fine-tuning, so even though the previous model was also trained for summarization, it was not well adapted for the new domain. Let's push our model to the Hub:

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

In the next section we'll use the model to generate a few summaries for us.

You can also evaluate the generations as part of the training loop: use the extension of `TrainingArguments` called `Seq2SeqTrainingArguments` and specify `predict_with_generate=True`. Pass it to the dedicated `Trainer` called `Seq2SeqTrainer`, which then uses the `generate()` function instead of the model's forward pass to create predictions for evaluation. Give it a try!

Generating Dialogue Summaries

Looking at the losses and ROUGE scores, it seems the model is showing a significant improvement over the original model trained on CNN/DailyMail only. Let's see what a summary generated on a sample from the test set looks like:

```
gen_kwargs = {"length_penalty": 0.8, "num_beams":8, "max_length": 128}
sample_text = dataset_samsum["test"][0]["dialogue"]
reference = dataset_samsum["test"][0]["summary"]
pipe = pipeline("summarization", model="transformersbook/pegasus-samsum")

print("Dialogue:")
print(sample_text)
print("\nReference Summary:")
print(reference)
print("\nModel Summary:")
print(pipe(sample_text, **gen_kwargs)[0]["summary_text"])
```

```
Dialogue:
Hannah: Hey, do you have Betty's number?
Amanda: Lemme check
Hannah: <file_gif>
Amanda: Sorry, can't find it.
Amanda: Ask Larry
Amanda: He called her last time we were at the park together
Hannah: I don't know him well
Hannah: <file_gif>
```

```
Amanda: Don't be shy, he's very nice
Hannah: If you say so..
Hannah: I'd rather you texted him
Amanda: Just text him 🙂
Hannah: Urgh.. Alright
Hannah: Bye
Amanda: Bye bye

Reference Summary:
Hannah needs Betty's number but Amanda doesn't have it. She needs to contact
Larry.

Model Summary:
Amanda can't find Betty's number. Larry called Betty last time they were at the
park together. Hannah wants Amanda to text Larry instead of calling Betty.
```

That looks much more like the reference summary. It seems the model has learned to synthesize the dialogue into a summary without just extracting passages. Now, the ultimate test: how well does the model work on a custom input?

```
custom_dialogue = """\
Thom: Hi guys, have you heard of transformers?
Lewis: Yes, I used them recently!
Leandro: Indeed, there is a great library by Hugging Face.
Thom: I know, I helped build it ;)
Lewis: Cool, maybe we should write a book about it. What do you think?
Leandro: Great idea, how hard can it be?!
Thom: I am in!
Lewis: Awesome, let's do it together!
"""
print(pipe(custom_dialogue, **gen_kwargs)[0]["summary_text"])
```

```
Thom, Lewis and Leandro are going to write a book about transformers. Thom
helped build a library by Hugging Face. They are going to do it together.
```

The generated summary of the custom dialogue makes sense. It summarizes well that all the people in the discussion want to write the book together and does not simply extract single sentences. For example, it synthesizes the third and fourth lines into a logical combination.

Conclusion

Text summarization poses some unique challenges compared to other tasks that can be framed as classification tasks, like sentiment analysis, named entity recognition, or question answering. Conventional metrics such as accuracy do not reflect the quality of the generated text. As we saw, the BLEU and ROUGE metrics can better evaluate generated texts; however, human judgment remains the best measure.

A common question when working with summarization models is how we can summarize documents where the texts are longer than the model's context length.

Unfortunately, there is no single strategy to solve this problem, and to date this is still an open and active research question. For example, recent work by OpenAI showed how to scale summarization by applying it recursively to long documents and using human feedback in the loop.[6]

In the next chapter we'll look at question answering, which is the task of providing an answer to a question based on a text passage. In contrast to summarization, with this task there exist good strategies to deal with long or many documents, and we'll show you how to scale question answering to thousands of documents.

6 J. Wu et al., "Recursively Summarizing Books with Human Feedback", (2021).

CHAPTER 7
Question Answering

Whether you're a researcher, analyst, or data scientist, chances are that at some point you've needed to wade through oceans of documents to find the information you're looking for. To make matters worse, you're constantly reminded by Google and Bing that there exist better ways to search! For instance, if we search for "When did Marie Curie win her first Nobel Prize?" on Google, we immediately get the correct answer of "1903," as illustrated in Figure 7-1.

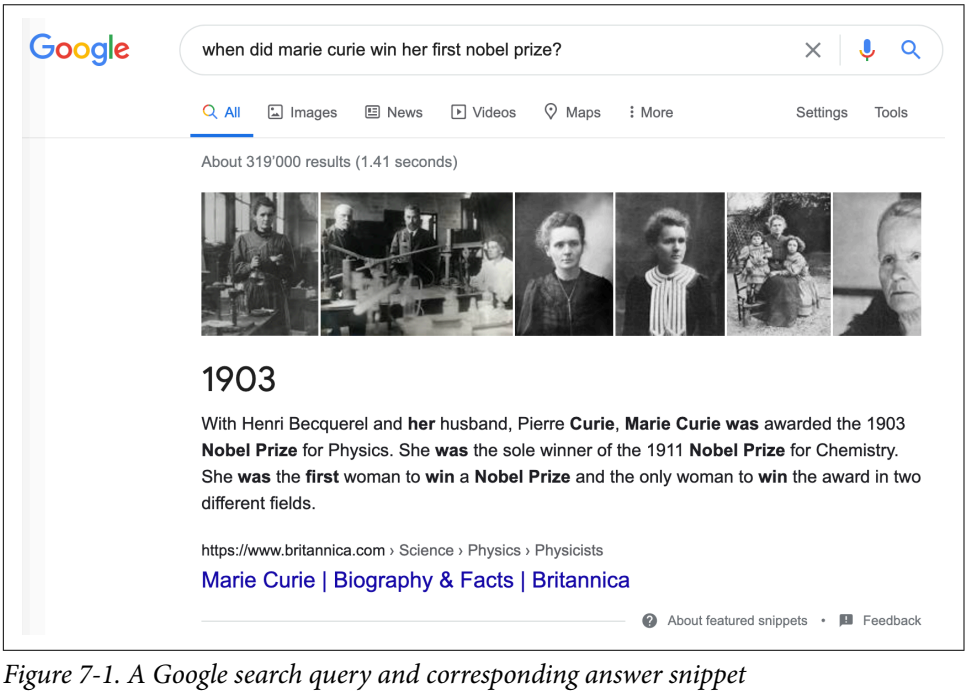


Figure 7-1. A Google search query and corresponding answer snippet

In this example, Google first retrieved around 319,000 documents that were relevant to the query, and then performed an additional processing step to extract the answer snippet with the corresponding passage and web page. It's not hard to see why these answer snippets are useful. For example, if we search for a trickier question like "Which guitar tuning is the best?" Google doesn't provide an answer, and instead we have to click on one of the web pages returned by the search engine to find it ourselves.[1]

The general approach behind this technology is called *question answering* (QA). There are many flavors of QA, but the most common is *extractive QA*, which involves questions whose answer can be identified as a span of text in a document, where the document might be a web page, legal contract, or news article. The two-stage process of first retrieving relevant documents and then extracting answers from them is also the basis for many modern QA systems, including semantic search engines, intelligent assistants, and automated information extractors. In this chapter, we'll apply this process to tackle a common problem facing ecommerce websites: helping consumers answer specific queries to evaluate a product. We'll see that customer reviews can be used as a rich and challenging source of information for QA, and along the way we'll learn how transformers act as powerful *reading comprehension* models that can extract meaning from text. Let's begin by fleshing out the use case.

This chapter focuses on extractive QA, but other forms of QA may be more suitable for your use case. For example, *community QA* involves gathering question-answer pairs that are generated by users on forums like Stack Overflow, and then using semantic similarity search to find the closest matching answer to a new question. There is also *long-form QA*, which aims to generate complex paragraph-length answers to open-ended questions like "Why is the sky blue?" Remarkably, it is also possible to do QA over tables, and transformer models like TAPAS can even perform aggregations to produce the final answer!

Building a Review-Based QA System

If you've ever purchased a product online, you probably relied on customer reviews to help inform your decision. These reviews can often help answer specific questions like "Does this guitar come with a strap?" or "Can I use this camera at night?" that may be hard to answer from the product description alone. However, popular products can have hundreds to thousands of reviews, so it can be a major drag to find one that is relevant. One alternative is to post your question on the community QA

1 Although, in this particular case, everyone agrees that Drop C is the best guitar tuning.

platforms provided by websites like Amazon, but it usually takes days to get an answer (if you get one at all). Wouldn't it be nice if we could get an immediate answer, like in the Google example from Figure 7-1? Let's see if we can do this using transformers!

The Dataset

To build our QA system we'll use the SubjQA dataset,[2] which consists of more than 10,000 customer reviews in English about products and services in six domains: TripAdvisor, Restaurants, Movies, Books, Electronics, and Grocery. As illustrated in Figure 7-2, each review is associated with a question that can be answered using one or more sentences from the review.[3]

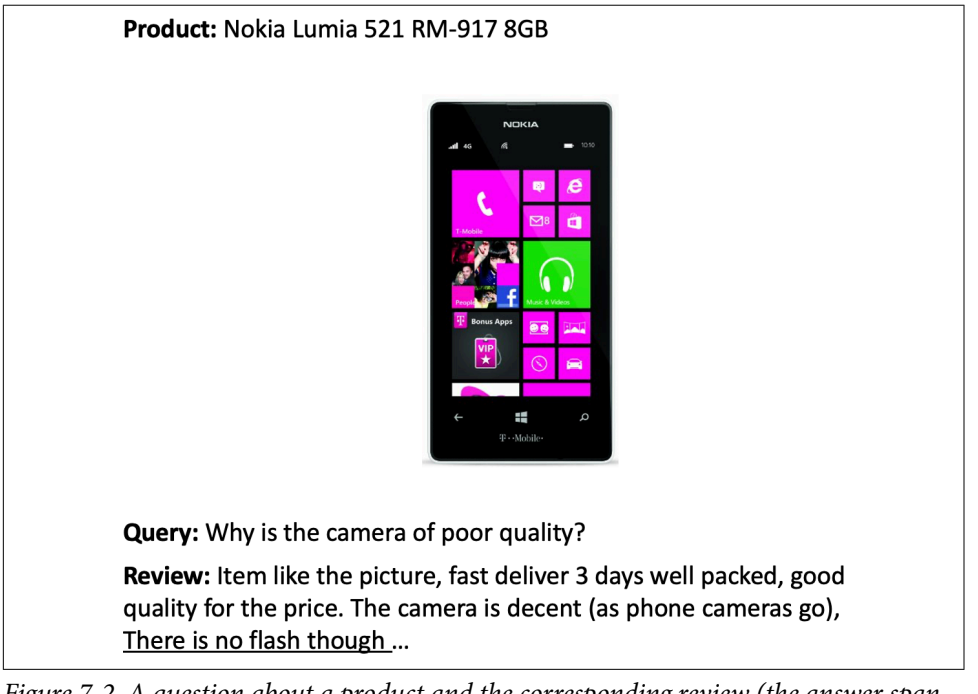


Figure 7-2. A question about a product and the corresponding review (the answer span is underlined)

The interesting aspect of this dataset is that most of the questions and answers are *subjective*; that is, they depend on the personal experience of the users. The example in Figure 7-2 shows why this feature makes the task potentially more difficult than

2 J. Bjerva et al., "SubjQA: A Dataset for Subjectivity and Review Comprehension", (2020).

3 As we'll soon see, there are also *unanswerable* questions that are designed to produce more robust models.

finding answers to factual questions like "What is the currency of the United Kingdom?" First, the query is about "poor quality," which is subjective and depends on the user's definition of quality. Second, important parts of the query do not appear in the review at all, which means it cannot be answered with shortcuts like keyword search or paraphrasing the input question. These features make SubjQA a realistic dataset to benchmark our review-based QA models on, since user-generated content like that shown in Figure 7-2 resembles what we might encounter in the wild.

QA systems are usually categorized by the *domain* of data that they have access to when responding to a query. *Closed-domain* QA deals with questions about a narrow topic (e.g., a single product category), while *open-domain* QA deals with questions about almost anything (e.g., Amazon's whole product catalog). In general, closed-domain QA involves searching through fewer documents than the open-domain case.

To get started, let's download the dataset from the Hugging Face Hub. As we did in Chapter 4, we can use the `get_dataset_config_names()` function to find out which subsets are available:

```
from datasets import get_dataset_config_names

domains = get_dataset_config_names("subjqa")
domains
```

```
['books', 'electronics', 'grocery', 'movies', 'restaurants', 'tripadvisor']
```

For our use case, we'll focus on building a QA system for the Electronics domain. To download the `electronics` subset, we just need to pass this value to the `name` argument of the `load_dataset()` function:

```
from datasets import load_dataset

subjqa = load_dataset("subjqa", name="electronics")
```

Like other question answering datasets on the Hub, SubjQA stores the answers to each question as a nested dictionary. For example, if we inspect one of the rows in the `answers` column:

```
print(subjqa["train"]["answers"][1])
```

```
{'text': ['Bass is weak as expected', 'Bass is weak as expected, even with EQ
adjusted up'], 'answer_start': [1302, 1302], 'answer_subj_level': [1, 1],
'ans_subj_score': [0.5083333253860474, 0.5083333253860474], 'is_ans_subjective':
[True, True]}
```

we can see that the answers are stored in a `text` field, while the starting character indices are provided in `answer_start`. To explore the dataset more easily, we'll flatten

these nested columns with the `flatten()` method and convert each split to a Pandas `DataFrame` as follows:

```
import pandas as pd

dfs = {split: dset.to_pandas() for split, dset in subjqa.flatten().items()}

for split, df in dfs.items():
    print(f"Number of questions in {split}: {df['id'].nunique()}")
```

```
Number of questions in train: 1295
Number of questions in test: 358
Number of questions in validation: 255
```

Notice that the dataset is relatively small, with only 1,908 examples in total. This simulates a real-world scenario, since getting domain experts to label extractive QA datasets is labor-intensive and expensive. For example, the CUAD dataset for extractive QA on legal contracts is estimated to have a value of $2 million to account for the legal expertise needed to annotate its 13,000 examples![4]

There are quite a few columns in the SubjQA dataset, but the most interesting ones for building our QA system are shown in Table 7-1.

Table 7-1. Column names and their descriptions from the SubjQA dataset

Column name	Description
`title`	The Amazon Standard Identification Number (ASIN) associated with each product
`question`	The question
`answers.answer_text`	The span of text in the review labeled by the annotator
`answers.answer_start`	The start character index of the answer span
`context`	The customer review

Let's focus on these columns and take a look at a few of the training examples. We can use the `sample()` method to select a random sample:

```
qa_cols = ["title", "question", "answers.text",
           "answers.answer_start", "context"]
sample_df = dfs["train"][qa_cols].sample(2, random_state=7)
sample_df
```

4 D. Hendrycks et al., "CUAD: An Expert-Annotated NLP Dataset for Legal Contract Review", (2021).

title	question	answers.text	answers.answer_start	context
B005DKZTMG	Does the keyboard lightweight?	[this keyboard is compact]	[215]	I really like this keyboard. I give it 4 stars because it doesn't have a CAPS LOCK key so I never know if my caps are on. But for the price, it really suffices as a wireless keyboard. I have very large hands and this keyboard is compact, but I have no complaints.
B00AAIPT76	How is the battery?	[]	[]	I bought this after the first spare gopro battery I bought wouldn't hold a charge. I have very realistic expectations of this sort of product, I am skeptical of amazing stories of charge time and battery life but I do expect the batteries to hold a charge for a couple of weeks at least and for the charger to work like a charger. In this I was not disappointed. I am a river rafter and found that the gopro burns through power in a hurry so this purchase solved that issue. the batteries held a charge, on shorter trips the extra two batteries were enough and on longer trips I could use my friends JOOS Orange to recharge them.I just bought a newtrent xtreme powerpak and expect to be able to charge these with that so I will not run out of power again.

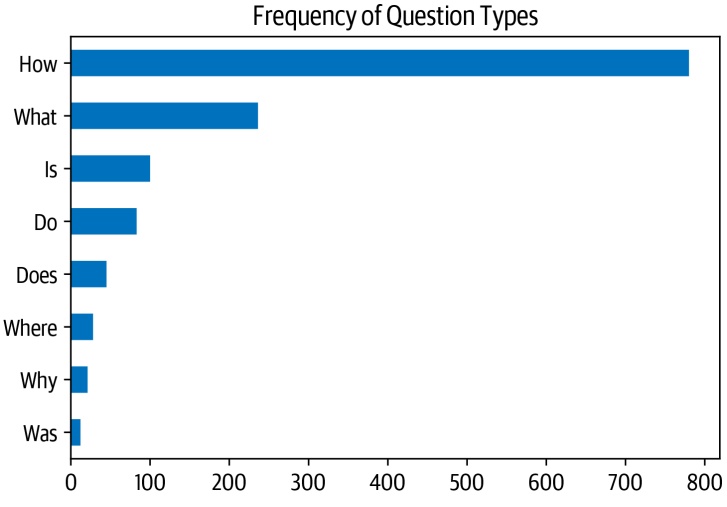
From these examples we can make a few observations. First, the questions are not grammatically correct, which is quite common in the FAQ sections of ecommerce websites. Second, an empty `answers.text` entry denotes "unanswerable" questions whose answer cannot be found in the review. Finally, we can use the start index and length of the answer span to slice out the span of text in the review that corresponds to the answer:

```
start_idx = sample_df["answers.answer_start"].iloc[0][0]
end_idx = start_idx + len(sample_df["answers.text"].iloc[0][0])
sample_df["context"].iloc[0][start_idx:end_idx]
```

```
'this keyboard is compact'
```

Next, let's get a feel for what types of questions are in the training set by counting the questions that begin with a few common starting words:

```
counts = {}
question_types = ["What", "How", "Is", "Does", "Do", "Was", "Where", "Why"]

for q in question_types:
    counts[q] = dfs["train"]["question"].str.startswith(q).value_counts()[True]

pd.Series(counts).sort_values().plot.barh()
plt.title("Frequency of Question Types")
plt.show()
```



We can see that questions beginning with "How", "What", and "Is" are the most common ones, so let's have a look at some examples:

```
for question_type in ["How", "What", "Is"]:
    for question in (
        dfs["train"][dfs["train"].question.str.startswith(question_type)]
        .sample(n=3, random_state=42)['question']):
        print(question)
```

```
How is the camera?
How do you like the control?
How fast is the charger?
What is direction?
What is the quality of the construction of the bag?
What is your impression of the product?
Is this how zoom works?
Is sound clear?
Is it a wireless keyboard?
```

The Stanford Question Answering Dataset

The *(question, review, [answer sentences])* format of SubjQA is commonly used in extractive QA datasets, and was pioneered in the Stanford Question Answering Dataset (SQuAD).[5] This is a famous dataset that is often used to test the ability of machines to read a passage of text and answer questions about it. The dataset was created by sampling several hundred English articles from Wikipedia, partitioning each article into paragraphs, and then asking crowdworkers to generate a set of questions

5 P. Rajpurkar et al., "SQuAD: 100,000+ Questions for Machine Comprehension of Text", (2016).

and answers for each paragraph. In the first version of SQuAD, each answer to a question was guaranteed to exist in the corresponding passage. But it wasn't long before sequence models started performing better than humans at extracting the correct span of text with the answer. To make the task more difficult, SQuAD 2.0 was created by augmenting SQuAD 1.1 with a set of adversarial questions that are relevant to a given passage but cannot be answered from the text alone.[6] The state of the art as of this book's writing is shown in Figure 7-3, with most models since 2019 surpassing human performance.

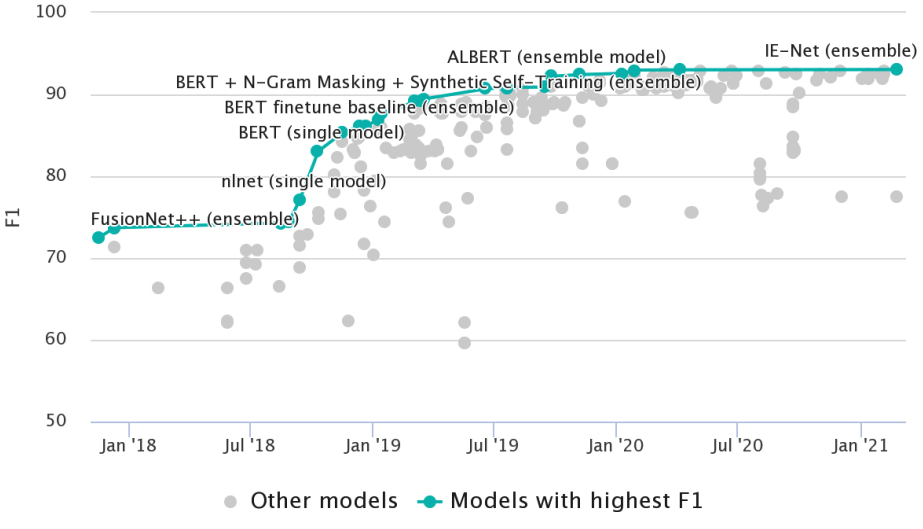


Figure 7-3. Progress on the SQuAD 2.0 benchmark (image from Papers with Code)

However, this superhuman performance does not appear to reflect genuine reading comprehension, since answers to the "unanswerable" questions can usually be identified through patterns in the passages like antonyms. To address these problems Google released the Natural Questions (NQ) dataset,[7] which involves fact-seeking questions obtained from Google Search users. The answers in NQ are much longer than in SQuAD and present a more challenging benchmark.

Now that we've explored our dataset a bit, let's dive into understanding how transformers can extract answers from text.

6 P. Rajpurkar, R. Jia, and P. Liang, "Know What You Don't Know: Unanswerable Questions for SQuAD", (2018).

7 T. Kwiatkowski et al., "Natural Questions: A Benchmark for Question Answering Research," *Transactions of the Association for Computational Linguistics* 7 (March 2019): 452–466, *http://dx.doi.org/10.1162/tacl_a_00276*.

Extracting Answers from Text

The first thing we'll need for our QA system is to find a way to identify a potential answer as a span of text in a customer review. For example, if a we have a question like "Is it waterproof?" and the review passage is "This watch is waterproof at 30m depth", then the model should output "waterproof at 30m". To do this we'll need to understand how to:

- Frame the supervised learning problem.
- Tokenize and encode text for QA tasks.
- Deal with long passages that exceed a model's maximum context size.

Let's start by taking a look at how to frame the problem.

Span classification

The most common way to extract answers from text is by framing the problem as a *span classification* task, where the start and end tokens of an answer span act as the labels that a model needs to predict. This process is illustrated in Figure 7-4.

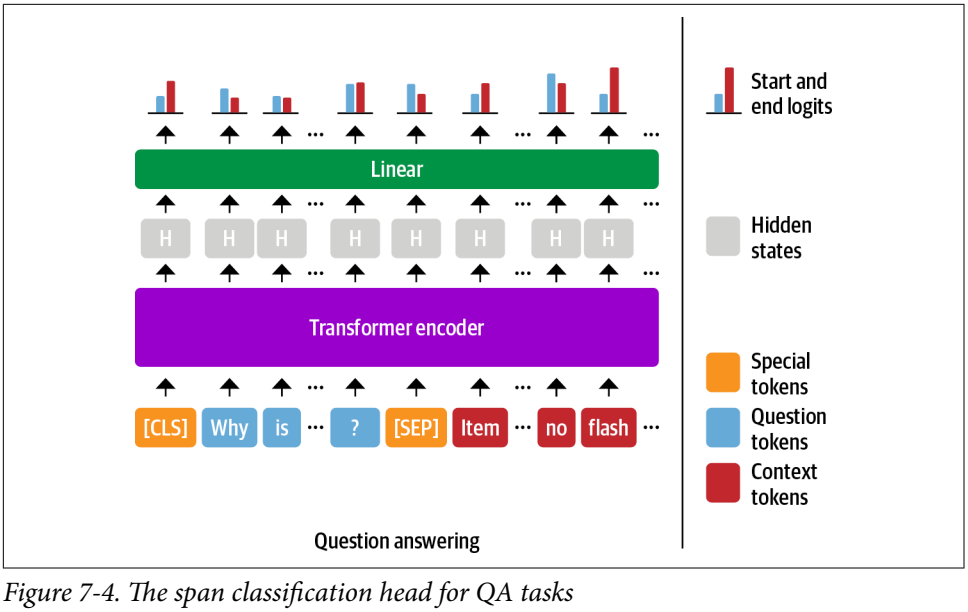


Figure 7-4. The span classification head for QA tasks

Since our training set is relatively small, with only 1,295 examples, a good strategy is to start with a language model that has already been fine-tuned on a large-scale QA dataset like SQuAD. In general, these models have strong reading comprehension capabilities and serve as a good baseline upon which to build a more accurate system. This is a somewhat different approach to that taken in previous chapters, where we

typically started with a pretrained model and fine-tuned the task-specific head ourselves. For example, in Chapter 2, we had to fine-tune the classification head because the number of classes was tied to the dataset at hand. For extractive QA, we can actually start with a fine-tuned model since the structure of the labels remains the same across datasets.

You can find a list of extractive QA models by navigating to the Hugging Face Hub and searching for "squad" on the Models tab (Figure 7-5).

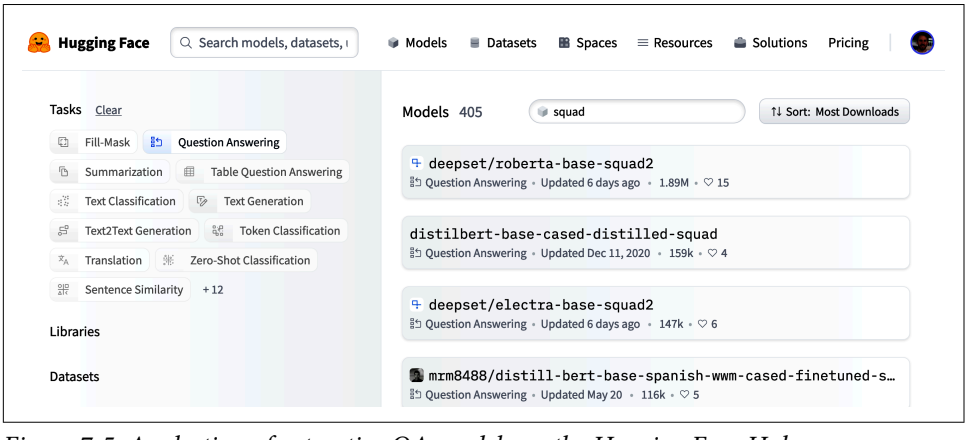


Figure 7-5. A selection of extractive QA models on the Hugging Face Hub

As you can see, at the time of writing, there are more than 350 QA models to choose from—so which one should you pick? In general, the answer depends on various factors like whether your corpus is mono- or multilingual and the constraints of running the model in a production environment. Table 7-2 lists a few models that provide a good foundation to build on.

Table 7-2. Baseline transformer models that are fine-tuned on SQuAD 2.0

Transformer	Description	Number of parameters	F_1-score on SQuAD 2.0
MiniLM	A distilled version of BERT-base that preserves 99% of the performance while being twice as fast	66M	79.5
RoBERTa-base	RoBERTa models have better performance than their BERT counterparts and can be fine-tuned on most QA datasets using a single GPU	125M	83.0
ALBERT-XXL	State-of-the-art performance on SQuAD 2.0, but computationally intensive and difficult to deploy	235M	88.1
XLM-RoBERTa-large	Multilingual model for 100 languages with strong zero-shot performance	570M	83.8

For the purposes of this chapter, we'll use a fine-tuned MiniLM model since it is fast to train and will allow us to quickly iterate on the techniques that we'll be exploring.[8] As usual, the first thing we need is a tokenizer to encode our texts, so let's take a look at how this works for QA tasks.

Tokenizing text for QA

To encode our texts, we'll load the MiniLM model checkpoint from the Hugging Face Hub as usual:

```
from transformers import AutoTokenizer

model_ckpt = "deepset/minilm-uncased-squad2"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
```

To see the model in action, let's first try to extract an answer from a short passage of text. In extractive QA tasks, the inputs are provided as (question, context) pairs, so we pass them both to the tokenizer as follows:

```
question = "How much music can this hold?"
context = """An MP3 is about 1 MB/minute, so about 6000 hours depending on \
file size."""
inputs = tokenizer(question, context, return_tensors="pt")
```

Here we've returned PyTorch `Tensor` objects, since we'll need them to run the forward pass through the model. If we view the tokenized inputs as a table:

input_ids	101	2129	2172	2189	2064	2023	...	5834	2006	5371	2946	1012	102
token_type_ids	0	0	0	0	0	0	...	1	1	1	1	1	1
attention_mask	1	1	1	1	1	1	...	1	1	1	1	1	1

we can see the familiar `input_ids` and `attention_mask` tensors, while the `token_type_ids` tensor indicates which part of the inputs corresponds to the question and context (a 0 indicates a question token, a 1 indicates a context token).[9]

To understand how the tokenizer formats the inputs for QA tasks, let's decode the `input_ids` tensor:

```
print(tokenizer.decode(inputs["input_ids"][0]))
```

8 W. Wang et al., "MINILM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers", (2020).

9 Note that the `token_type_ids` are not present in all transformer models. In the case of BERT-like models such as MiniLM, the `token_type_ids` are also used during pretraining to incorporate the next sentence prediction task.

```
[CLS] how much music can this hold? [SEP] an mp3 is about 1 mb / minute, so
about 6000 hours depending on file size. [SEP]
```

We see that for each QA example, the inputs take the format:

```
[CLS] question tokens [SEP] context tokens [SEP]
```

where the location of the first `[SEP]` token is determined by the `token_type_ids`. Now that our text is tokenized, we just need to instantiate the model with a QA head and run the inputs through the forward pass:

```
import torch
from transformers import AutoModelForQuestionAnswering

model = AutoModelForQuestionAnswering.from_pretrained(model_ckpt)

with torch.no_grad():
    outputs = model(**inputs)
print(outputs)
```

```
QuestionAnsweringModelOutput(loss=None, start_logits=tensor([[-0.9862, -4.7750,
         -5.4025, -5.2378, -5.2863, -5.5117, -4.9819, -6.1880,
         -0.9862,  0.2596, -0.2144, -1.7136,  3.7806,  4.8561, -1.0546, -3.9097,
         -1.7374, -4.5944, -1.4278,  3.9949,  5.0390, -0.2018, -3.0193, -4.8549,
         -2.3107, -3.5110, -3.5713, -0.9862]]), end_logits=tensor([[-0.9623,
         -5.4733, -5.0326, -5.1639, -5.4278, -5.5151, -5.1749, -4.6233,
         -0.9623, -3.7855, -0.8715, -3.7745, -3.0161, -1.1780,  0.1758, -2.7365,
          4.8934,  0.3046, -3.1761, -3.2762,  0.8937,  5.6606, -0.3623, -4.9554,
         -3.2531, -0.0914,  1.6211, -0.9623]]), hidden_states=None,
attentions=None)
```

Here we can see that we get a `QuestionAnsweringModelOutput` object as the output of the QA head. As illustrated in Figure 7-4, the QA head corresponds to a linear layer that takes the hidden states from the encoder and computes the logits for the start and end spans.[10] This means that we treat QA as a form of token classification, similar to what we encountered for named entity recognition in Chapter 4. To convert the outputs into an answer span, we first need to get the logits for the start and end tokens:

```
start_logits = outputs.start_logits
end_logits = outputs.end_logits
```

If we compare the shapes of these logits to the input IDs:

```
print(f"Input IDs shape: {inputs.input_ids.size()}")
print(f"Start logits shape: {start_logits.size()}")
print(f"End logits shape: {end_logits.size()}")
```

10 See Chapter 2 for details on how these hidden states can be extracted.

```
Input IDs shape: torch.Size([1, 28])
Start logits shape: torch.Size([1, 28])
End logits shape: torch.Size([1, 28])
```

we see that there are two logits (a start and end) associated with each input token. As illustrated in Figure 7-6, larger, positive logits correspond to more likely candidates for the start and end tokens. In this example we can see that the model assigns the highest start token logits to the numbers "1" and "6000", which makes sense since our question is asking about a quantity. Similarly, we see that the end tokens with the highest logits are "minute" and "hours".

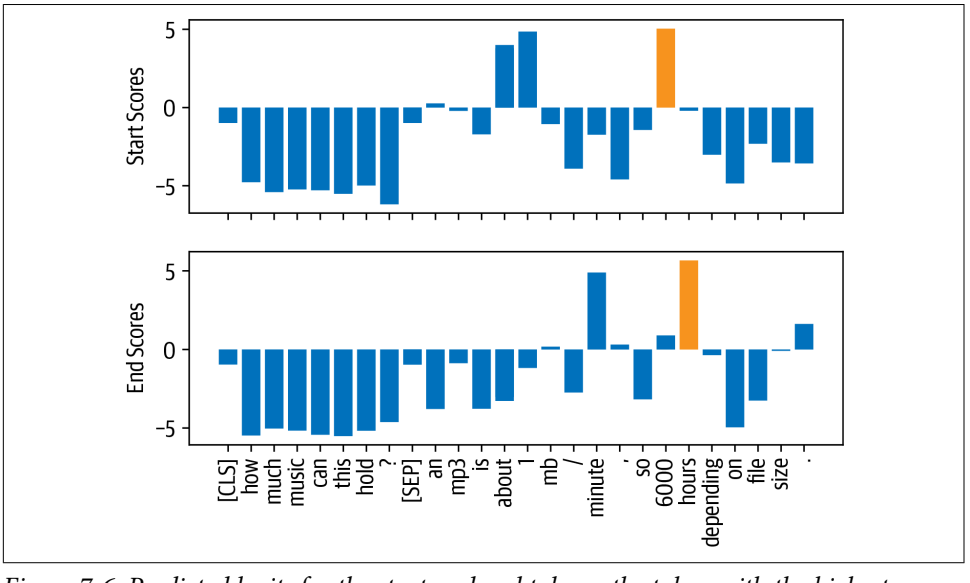


Figure 7-6. Predicted logits for the start and end tokens; the token with the highest score is colored in orange

To get the final answer, we can compute the argmax over the start and end token logits and then slice the span from the inputs. The following code performs these steps and decodes the result so we can print the resulting text:

```
import torch

start_idx = torch.argmax(start_logits)
end_idx = torch.argmax(end_logits) + 1
answer_span = inputs["input_ids"][0][start_idx:end_idx]
answer = tokenizer.decode(answer_span)
print(f"Question: {question}")
print(f"Answer: {answer}")
```

```
Question: How much music can this hold?
Answer: 6000 hours
```

Great, it worked! In 🤗 Transformers, all of these preprocessing and postprocessing steps are conveniently wrapped in a dedicated pipeline. We can instantiate the pipeline by passing our tokenizer and fine-tuned model as follows:

```
from transformers import pipeline

pipe = pipeline("question-answering", model=model, tokenizer=tokenizer)
pipe(question=question, context=context, topk=3)
```

```
[{'score': 0.26516005396842957,
  'start': 38,
  'end': 48,
  'answer': '6000 hours'},
 {'score': 0.2208300083875656,
  'start': 16,
  'end': 48,
  'answer': '1 MB/minute, so about 6000 hours'},
 {'score': 0.10253632068634033,
  'start': 16,
  'end': 27,
  'answer': '1 MB/minute'}]
```

In addition to the answer, the pipeline also returns the model's probability estimate in the `score` field (obtained by taking a softmax over the logits). This is handy when we want to compare multiple answers within a single context. We've also shown that we can have the model predict multiple answers by specifying the `topk` parameter. Sometimes, it is possible to have questions for which no answer is possible, like the empty `answers.answer_start` examples in SubjQA. In these cases the model will assign a high start and end score to the `[CLS]` token, and the pipeline maps this output to an empty string:

```
pipe(question="Why is there no data?", context=context,
     handle_impossible_answer=True)
```

```
{'score': 0.9068416357040405, 'start': 0, 'end': 0, 'answer': ''}
```

In our simple example, we obtained the start and end indices by taking the argmax of the corresponding logits. However, this heuristic can produce out-of-scope answers by selecting tokens that belong to the question instead of the context. In practice, the pipeline computes the best combination of start and end indices subject to various constraints such as being in-scope, requiring the start indices to precede the end indices, and so on.

Dealing with long passages

One subtlety faced by reading comprehension models is that the context often contains more tokens than the maximum sequence length of the model (which is usually a few hundred tokens at most). As illustrated in Figure 7-7, a decent portion of the SubjQA training set contains question-context pairs that won't fit within MiniLM's context size of 512 tokens.

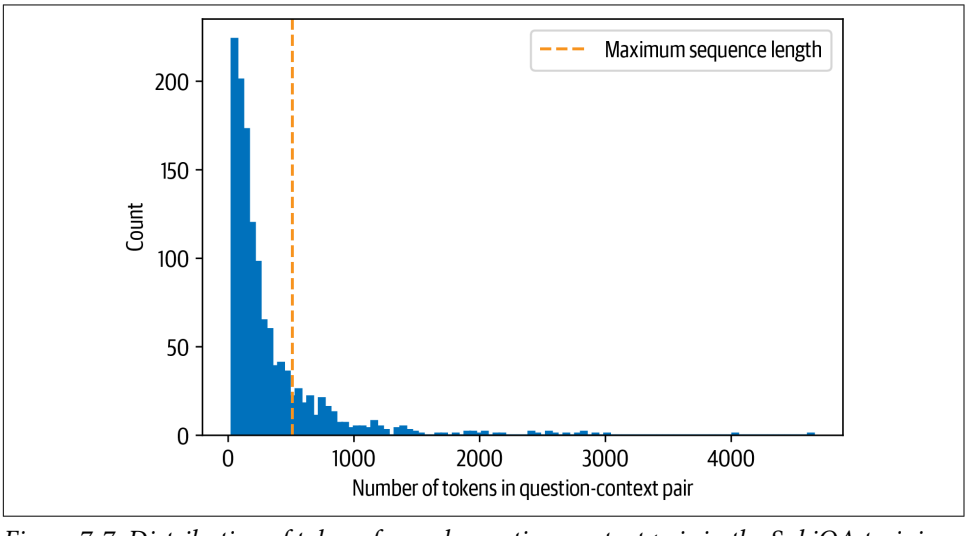


Figure 7-7. Distribution of tokens for each question-context pair in the SubjQA training set

For other tasks, like text classification, we simply truncated long texts under the assumption that enough information was contained in the embedding of the `[CLS]` token to generate accurate predictions. For QA, however, this strategy is problematic because the answer to a question could lie near the end of the context and thus would be removed by truncation. As illustrated in Figure 7-8, the standard way to deal with this is to apply a *sliding window* across the inputs, where each window contains a passage of tokens that fit in the model's context.

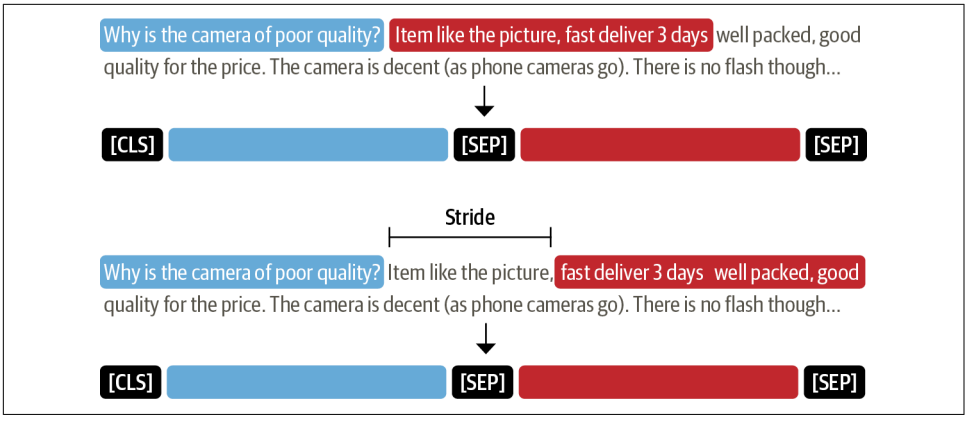


Figure 7-8. How the sliding window creates multiple question-context pairs for long documents—the first bar corresponds to the question, while the second bar is the context captured in each window

In 🤗 Transformers, we can set `return_overflowing_tokens=True` in the tokenizer to enable the sliding window. The size of the sliding window is controlled by the `max_seq_length` argument, and the size of the stride is controlled by `doc_stride`. Let's grab the first example from our training set and define a small window to illustrate how this works:

```
example = dfs["train"].iloc[0][["question", "context"]]
tokenized_example = tokenizer(example["question"], example["context"],
                              return_overflowing_tokens=True, max_length=100,
                              stride=25)
```

In this case we now get a list of `input_ids`, one for each window. Let's check the number of tokens we have in each window:

```
for idx, window in enumerate(tokenized_example["input_ids"]):
    print(f"Window #{idx} has {len(window)} tokens")
```

```
Window #0 has 100 tokens
Window #1 has 88 tokens
```

Finally, we can see where two windows overlap by decoding the inputs:

```
for window in tokenized_example["input_ids"]:
    print(f"{tokenizer.decode(window)} \n")
```

```
[CLS] how is the bass? [SEP] i have had koss headphones in the past, pro 4aa and
qz - 99. the koss portapro is portable and has great bass response. the work
great with my android phone and can be " rolled up " to be carried in my
motorcycle jacket or computer bag without getting crunched. they are very light
and don't feel heavy or bear down on your ears even after listening to music
with them on all day. the sound is [SEP]

[CLS] how is the bass? [SEP] and don't feel heavy or bear down on your ears even
```

```
after listening to music with them on all day. the sound is night and day better
than any ear - bud could be and are almost as good as the pro 4aa. they are "
open air " headphones so you cannot match the bass to the sealed types, but it
comes close. for $ 32, you cannot go wrong. [SEP]
```

Now that we have some intuition about how QA models can extract answers from text, let's look at the other components we need to build an end-to-end QA pipeline.

Using Haystack to Build a QA Pipeline

In our simple answer extraction example, we provided both the question and the context to the model. However, in reality our system's users will only provide a question about a product, so we need some way of selecting relevant passages from among all the reviews in our corpus. One way to do this would be to concatenate all the reviews of a given product together and feed them to the model as a single, long context. Although simple, the drawback of this approach is that the context can become extremely long and thereby introduce an unacceptable latency for our users' queries. For example, let's suppose that on average, each product has 30 reviews and each review takes 100 milliseconds to process. If we need to process all the reviews to get an answer, this would result in an average latency of 3 seconds per user query—much too long for ecommerce websites!

To handle this, modern QA systems are typically based on the *retriever-reader* architecture, which has two main components:

Retriever
: Responsible for retrieving relevant documents for a given query. Retrievers are usually categorized as *sparse* or *dense*. Sparse retrievers use word frequencies to represent each document and query as a sparse vector.[11] The relevance of a query and a document is then determined by computing an inner product of the vectors. On the other hand, dense retrievers use encoders like transformers to represent the query and document as contextualized embeddings (which are dense vectors). These embeddings encode semantic meaning, and allow dense retrievers to improve search accuracy by understanding the content of the query.

Reader
: Responsible for extracting an answer from the documents provided by the retriever. The reader is usually a reading comprehension model, although at the end of the chapter we'll see examples of models that can generate free-form answers.

11 A vector is sparse if most of its elements are zero.

As illustrated in Figure 7-9, there can also be other components that apply post-processing to the documents fetched by the retriever or to the answers extracted by the reader. For example, the retrieved documents may need reranking to eliminate noisy or irrelevant ones that can confuse the reader. Similarly, postprocessing of the reader's answers is often needed when the correct answer comes from various passages in a long document.

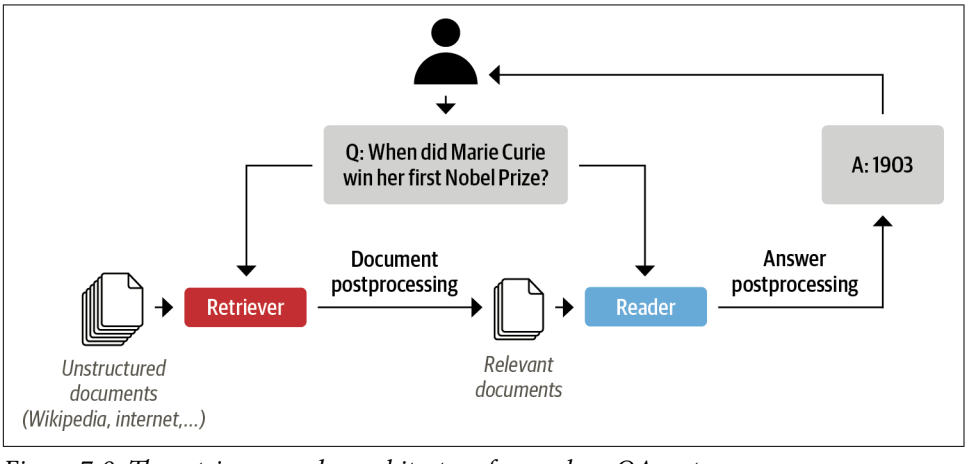


Figure 7-9. The retriever-reader architecture for modern QA systems

To build our QA system, we'll use the *Haystack* library developed by deepset, a German company focused on NLP. Haystack is based on the retriever-reader architecture, abstracts much of the complexity involved in building these systems, and integrates tightly with 🤗 Transformers.

In addition to the retriever and reader, there are two more components involved when building a QA pipeline with Haystack:

Document store
: A document-oriented database that stores documents and metadata which are provided to the retriever at query time

Pipeline
: Combines all the components of a QA system to enable custom query flows, merging documents from multiple retrievers, and more

In this section we'll look at how we can use these components to quickly build a prototype QA pipeline. Later, we'll examine how we can improve its performance.

This chapter was written using version 0.9.0 of the Haystack library. In version 0.10.0, the pipeline and evaluation APIs were redesigned to make it easier to inspect whether the retriever or reader are impacting performance. To see what this chapter's code looks like with the new API, check out the GitHub repository.

Initializing a document store

In Haystack, there are various document stores to choose from and each one can be paired with a dedicated set of retrievers. This is illustrated in Table 7-3, where the compatibility of sparse (TF-IDF, BM25) and dense (Embedding, DPR) retrievers is shown for each of the available document stores. We'll explain what all these acronyms mean later in this chapter.

Table 7-3. Compatibility of Haystack retrievers and document stores

	In memory	Elasticsearch	FAISS	Milvus
TF-IDF	Yes	Yes	No	No
BM25	No	Yes	No	No
Embedding	Yes	Yes	Yes	Yes
DPR	Yes	Yes	Yes	Yes

Since we'll be exploring both sparse and dense retrievers in this chapter, we'll use the `ElasticsearchDocumentStore`, which is compatible with both retriever types. Elasticsearch is a search engine that is capable of handling a diverse range of data types, including textual, numerical, geospatial, structured, and unstructured. Its ability to store huge volumes of data and quickly filter it with full-text search features makes it especially well suited for developing QA systems. It also has the advantage of being the industry standard for infrastructure analytics, so there's a good chance your company already has a cluster that you can work with.

To initialize the document store, we first need to download and install Elasticsearch. By following Elasticsearch's guide,[12] we can grab the latest release for Linux with `wget` and unpack it with the `tar` shell command:

```
url = """https://artifacts.elastic.co/downloads/elasticsearch/\
elasticsearch-7.9.2-linux-x86_64.tar.gz"""
!wget -nc -q {url}
!tar -xzf elasticsearch-7.9.2-linux-x86_64.tar.gz
```

Next we need to start the Elasticsearch server. Since we're running all the code in this book within Jupyter notebooks, we'll need to use Python's `Popen()` function to spawn

12 The guide also provides installation instructions for macOS and Windows.

a new process. While we're at it, let's also run the subprocess in the background using the `chown` shell command:

```
import os
from subprocess import Popen, PIPE, STDOUT

# Run Elasticsearch as a background process
!chown -R daemon:daemon elasticsearch-7.9.2
es_server = Popen(args=['elasticsearch-7.9.2/bin/elasticsearch'],
                  stdout=PIPE, stderr=STDOUT, preexec_fn=lambda: os.setuid(1))
# Wait until Elasticsearch has started
!sleep 30
```

In the `Popen()` function, the `args` specify the program we wish to execute, while `stdout=PIPE` creates a new pipe for the standard output and `stderr=STDOUT` collects the errors in the same pipe. The `preexec_fn` argument specifies the ID of the subprocess we wish to use. By default, Elasticsearch runs locally on port 9200, so we can test the connection by sending an HTTP request to `localhost`:

```
!curl -X GET "localhost:9200/?pretty"
```

```
{
  "name" : "96938eee37cd",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "ABGDdvbbRWmMb9Umz79HbA",
  "version" : {
    "number" : "7.9.2",
    "build_flavor" : "default",
    "build_type" : "docker",
    "build_hash" : "d34da0ea4a966c4e49417f2da2f244e3e97b4e6e",
    "build_date" : "2020-09-23T00:45:33.626720Z",
    "build_snapshot" : false,
    "lucene_version" : "8.6.2",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

Now that our Elasticsearch server is up and running, the next thing to do is instantiate the document store:

```
from haystack.document_store.elasticsearch import ElasticsearchDocumentStore

# Return the document embedding for later use with dense retriever
document_store = ElasticsearchDocumentStore(return_embedding=True)
```

By default, `ElasticsearchDocumentStore` creates two indices on Elasticsearch: one called `document` for (you guessed it) storing documents, and another called `label` for storing the annotated answer spans. For now, we'll just populate the `document` index

with the SubjQA reviews, and Haystack's document stores expect a list of dictionaries with `text` and `meta` keys as follows:

```
{
    "text": "<the-context>",
    "meta": {
        "field_01": "<additional-metadata>",
        "field_02": "<additional-metadata>",
        ...
    }
}
```

The fields in `meta` can be used for applying filters during retrieval. For our purposes we'll include the `item_id` and `q_review_id` columns of SubjQA so we can filter by product and question ID, along with the corresponding training split. We can then loop through the examples in each `DataFrame` and add them to the index with the `write_documents()` method as follows:

```
for split, df in dfs.items():
    # Exclude duplicate reviews
    docs = [{"text": row["context"],
             "meta":{"item_id": row["title"], "question_id": row["id"],
                     "split": split}}
        for _,row in df.drop_duplicates(subset="context").iterrows()]
    document_store.write_documents(docs, index="document")

print(f"Loaded {document_store.get_document_count()} documents")
```

```
Loaded 1615 documents
```

Great, we've loaded all our reviews into an index! To search the index we'll need a retriever, so let's look at how we can initialize one for Elasticsearch.

Initializing a retriever

The Elasticsearch document store can be paired with any of the Haystack retrievers, so let's start by using a sparse retriever based on BM25 (short for "Best Match 25"). BM25 is an improved version of the classic Term Frequency-Inverse Document Frequency (TF-IDF) algorithm and represents the question and context as sparse vectors that can be searched efficiently on Elasticsearch. The BM25 score measures how much matched text is about a search query and improves on TF-IDF by saturating TF values quickly and normalizing the document length so that short documents are favored over long ones.[13]

13 For an in-depth explanation of document scoring with TF-IDF and BM25 see Chapter 23 of *Speech and Language Processing*, 3rd edition, by D. Jurafsky and J.H. Martin (Prentice Hall).

In Haystack, the BM25 retriever is used by default in `ElasticsearchRetriever`, so let's initialize this class by specifying the document store we wish to search over:

```
from haystack.retriever.sparse import ElasticsearchRetriever

es_retriever = ElasticsearchRetriever(document_store=document_store)
```

Next, let's look at a simple query for a single electronics product in the training set. For review-based QA systems like ours, it's important to restrict the queries to a single item because otherwise the retriever would source reviews about products that are not related to a user's query. For example, asking "Is the camera quality any good?" without a product filter could return reviews about phones, when the user might be asking about a specific laptop camera instead. By themselves, the ASIN values in our dataset are a bit cryptic, but we can decipher them with online tools like *amazon ASIN* or by simply appending the value of `item_id` to the *www.amazon.com/dp/* URL. The following item ID corresponds to one of Amazon's Fire tablets, so let's use the retriever's `retrieve()` method to ask if it's any good for reading with:

```
item_id = "B0074BW614"
query = "Is it good for reading?"
retrieved_docs = es_retriever.retrieve(
    query=query, top_k=3, filters={"item_id":[item_id], "split":["train"]})
```

Here we've specified how many documents to return with the `top_k` argument and applied a filter on both the `item_id` and `split` keys that were included in the `meta` field of our documents. Each element of `retrieved_docs` is a Haystack `Document` object that is used to represent documents and includes the retriever's query score along with other metadata. Let's have a look at one of the retrieved documents:

```
print(retrieved_docs[0])
```

```
{'text': 'This is a gift to myself.  I have been a kindle user for 4 years and
this is my third one.  I never thought I would want a fire for I mainly use it
for book reading.  I decided to try the fire for when I travel I take my laptop,
my phone and my iPod classic.  I love my iPod but watching movies on the plane
with it can be challenging because it is so small. Laptops battery life is not
as good as the Kindle.  So the Fire combines for me what I needed all three to
do. So far so good.', 'score': 6.243799, 'probability': 0.6857824513476455,
'question': None, 'meta': {'item_id': 'B0074BW614', 'question_id':
'868e311275e26dbafe5af70774a300f3', 'split': 'train'}, 'embedding': None, 'id':
'252e83e25d52df7311d597dc89eef9f6'}
```

In addition to the document's text, we can see the `score` that Elasticsearch computed for its relevance to the query (larger scores imply a better match). Under the hood, Elasticsearch relies on Lucene for indexing and search, so by default it uses Lucene's *practical scoring function*. You can find the nitty-gritty details behind the scoring function in the Elasticsearch documentation, but in brief terms it first filters the candidate documents by applying a Boolean test (does the document match the query?),

and then applies a similarity metric that's based on representing both the document and the query as vectors.

Now that we have a way to retrieve relevant documents, the next thing we need is a way to extract answers from them. This is where the reader comes in, so let's take a look at how we can load our MiniLM model in Haystack.

Initializing a reader

In Haystack, there are two types of readers one can use to extract answers from a given context:

`FARMReader`
: Based on deepset's *FARM* framework for fine-tuning and deploying transformers. Compatible with models trained using 🤗 Transformers and can load models directly from the Hugging Face Hub.

`TransformersReader`
: Based on the QA pipeline from 🤗 Transformers. Suitable for running inference only.

Although both readers handle a model's weights in the same way, there are some differences in the way the predictions are converted to produce answers:

- In 🤗 Transformers, the QA pipeline normalizes the start and end logits with a softmax in each passage. This means that it is only meaningful to compare answer scores between answers extracted from the same passage, where the probabilities sum to 1. For example, an answer score of 0.9 from one passage is not necessarily better than a score of 0.8 in another. In FARM, the logits are not normalized, so inter-passage answers can be compared more easily.
- The `TransformersReader` sometimes predicts the same answer twice, but with different scores. This can happen in long contexts if the answer lies across two overlapping windows. In FARM, these duplicates are removed.

Since we will be fine-tuning the reader later in the chapter, we'll use the `FARMReader`. As with 🤗 Transformers, to load the model we just need to specify the MiniLM checkpoint on the Hugging Face Hub along with some QA-specific arguments:

```
from haystack.reader.farm import FARMReader

model_ckpt = "deepset/minilm-uncased-squad2"
max_seq_length, doc_stride = 384, 128
reader = FARMReader(model_name_or_path=model_ckpt, progress_bar=False,
                    max_seq_len=max_seq_length, doc_stride=doc_stride,
                    return_no_answer=True)
```

It is also possible to fine-tune a reading comprehension model directly in 🤗 Transformers and then load it in `TransformersReader` to run inference. For details on how to do the fine-tuning step, see the question answering tutorial in the library's documentation.

In `FARMReader`, the behavior of the sliding window is controlled by the same `max_seq_length` and `doc_stride` arguments that we saw for the tokenizer. Here we've used the values from the MiniLM paper. To confirm, let's now test the reader on our simple example from earlier:

```
print(reader.predict_on_texts(question=question, texts=[context], top_k=1))
```

```
{'query': 'How much music can this hold?', 'no_ans_gap': 12.648084878921509,
'answers': [{'answer': '6000 hours', 'score': 10.69961929321289, 'probability':
0.3988136053085327, 'context': 'An MP3 is about 1 MB/minute, so about 6000 hours
depending on file size.', 'offset_start': 38, 'offset_end': 48,
'offset_start_in_doc': 38, 'offset_end_in_doc': 48, 'document_id':
'e344757014e804eff50faa3ecf1c9c75'}]}
```

Great, the reader appears to be working as expected—so next, let's tie together all our components using one of Haystack's pipelines.

Putting it all together

Haystack provides a `Pipeline` abstraction that allows us to combine retrievers, readers, and other components together as a graph that can be easily customized for each use case. There are also predefined pipelines analogous to those in 🤗 Transformers, but specialized for QA systems. In our case, we're interested in extracting answers, so we'll use the `ExtractiveQAPipeline`, which takes a single retriever-reader pair as its arguments:

```
from haystack.pipeline import ExtractiveQAPipeline

pipe = ExtractiveQAPipeline(reader, es_retriever)
```

Each `Pipeline` has a `run()` method that specifies how the query flow should be executed. For the `ExtractiveQAPipeline` we just need to pass the `query`, the number of documents to retrieve with `top_k_retriever`, and the number of answers to extract from these documents with `top_k_reader`. In our case, we also need to specify a filter over the item ID, which can be done using the `filters` argument as we did with the retriever earlier. Let's run a simple example using our question about the Amazon Fire tablet again, but this time returning the extracted answers:

```
n_answers = 3
preds = pipe.run(query=query, top_k_retriever=3, top_k_reader=n_answers,
                 filters={"item_id": [item_id], "split":["train"]})

print(f"Question: {preds['query']} \n")
```

```
for idx in range(n_answers):
    print(f"Answer {idx+1}: {preds['answers'][idx]['answer']}")
    print(f"Review snippet: ...{preds['answers'][idx]['context']}...")
    print("\n\n")
```

```
Question: Is it good for reading?

Answer 1: I mainly use it for book reading
Review snippet: ... is my third one.  I never thought I would want a fire for I
mainly use it for book reading.  I decided to try the fire for when I travel I
take my la...


Answer 2: the larger screen compared to the Kindle makes for easier reading
Review snippet: ...ght enough that I can hold it to read, but the larger screen
compared to the Kindle makes for easier reading. I love the color, something I
never thou...


Answer 3: it is great for reading books when no light is available
Review snippet: ...ecoming addicted to hers! Our son LOVES it and it is great
for reading books when no light is available. Amazing sound but I suggest good
headphones t...
```

Great, we now have an end-to-end QA system for Amazon product reviews! This is a good start, but notice that the second and third answers are closer to what the question is actually asking. To do better, we'll need some metrics to quantify the performance of the retriever and reader. We'll take a look at that next.

Improving Our QA Pipeline

Although much of the recent research on QA has focused on improving reading comprehension models, in practice it doesn't matter how good your reader is if the retriever can't find the relevant documents in the first place! In particular, the retriever sets an upper bound on the performance of the whole QA system, so it's important to make sure it's doing a good job. With this in mind, let's start by introducing some common metrics to evaluate the retriever so that we can compare the performance of sparse and dense representations.

Evaluating the Retriever

A common metric for evaluating retrievers is *recall*, which measures the fraction of all relevant documents that are retrieved. In this context, "relevant" simply means whether the answer is present in a passage of text or not, so given a set of questions, we can compute recall by counting the number of times an answer appears in the top k documents returned by the retriever.

In Haystack, there are two ways to evaluate retrievers:

- Use the retriever's in-built `eval()` method. This can be used for both open- and closed-domain QA, but not for datasets like SubjQA where each document is paired with a single product and we need to filter by product ID for every query.
- Build a custom `Pipeline` that combines a retriever with the `EvalRetriever` class. This enables the implementation of custom metrics and query flows.

A complementary metric to recall is *mean average precision* (mAP), which rewards retrievers that can place the correct answers higher up in the document ranking.

Since we need to evaluate the recall per product and then aggregate across all products, we'll opt for the second approach. Each node in the `Pipeline` graph represents a class that takes some inputs and produces some outputs via a `run()` method:

```
class PipelineNode:
    def __init__(self):
        self.outgoing_edges = 1

    def run(self, **kwargs):
        ...
        return (outputs, "outgoing_edge_name")
```

Here `kwargs` corresponds to the outputs from the previous node in the graph, which is manipulated within the `run()` method to return a tuple of the outputs for the next node, along with a name for the outgoing edge. The only other requirement is to include an `outgoing_edges` attribute that indicates the number of outputs from the node (in most cases `outgoing_edges=1`, unless you have branches in the pipeline that route the inputs according to some criterion).

In our case, we need a node to evaluate the retriever, so we'll use the `EvalRetriever` class whose `run()` method keeps track of which documents have answers that match the ground truth. With this class we can then build up a `Pipeline` graph by adding the evaluation node after a node that represents the retriever itself:

```
from haystack.pipeline import Pipeline
from haystack.eval import EvalDocuments

class EvalRetrieverPipeline:
    def __init__(self, retriever):
        self.retriever = retriever
        self.eval_retriever = EvalDocuments()
        pipe = Pipeline()
        pipe.add_node(component=self.retriever, name="ESRetriever",
```

```
                    inputs=["Query"])
        pipe.add_node(component=self.eval_retriever, name="EvalRetriever",
                    inputs=["ESRetriever"])
        self.pipeline = pipe


pipe = EvalRetrieverPipeline(es_retriever)
```

Notice that each node is given a `name` and a list of `inputs`. In most cases, each node has a single outgoing edge, so we just need to include the name of the previous node in `inputs`.

Now that we have our evaluation pipeline, we need to pass some queries and their corresponding answers. To do this, we'll add the answers to a dedicated `label` index on our document store. Haystack provides a `Label` object that represents the answer spans and their metadata in a standardized fashion. To populate the `label` index, we'll first create a list of `Label` objects by looping over each question in the test set and extracting the matching answers and additional metadata:

```
from haystack import Label

labels = []
for i, row in dfs["test"].iterrows():
    # Metadata used for filtering in the Retriever
    meta = {"item_id": row["title"], "question_id": row["id"]}
    # Populate labels for questions with answers
    if len(row["answers.text"]):
        for answer in row["answers.text"]:
            label = Label(
                question=row["question"], answer=answer, id=i, origin=row["id"],
                meta=meta, is_correct_answer=True, is_correct_document=True,
                no_answer=False)
            labels.append(label)
    # Populate labels for questions without answers
    else:
        label = Label(
            question=row["question"], answer="", id=i, origin=row["id"],
            meta=meta, is_correct_answer=True, is_correct_document=True,
            no_answer=True)
        labels.append(label)
```

If we peek at one of these labels:

```
print(labels[0])
```

```
{'id': 'e28f5e62-85e8-41b2-8a34-fbff63b7a466', 'created_at': None, 'updated_at':
None, 'question': 'What is the tonal balance of these headphones?', 'answer': 'I
have been a headphone fanatic for thirty years', 'is_correct_answer': True,
'is_correct_document': True, 'origin': 'd0781d13200014aa25860e44da9d5ea7',
'document_id': None, 'offset_start_in_doc': None, 'no_answer': False,
'model_id': None, 'meta': {'item_id': 'B00001WRSJ', 'question_id':
'd0781d13200014aa25860e44da9d5ea7'}}
```

we can see the question-answer pair, along with an `origin` field that contains the unique question ID so we can filter the document store per question. We've also added the product ID to the `meta` field so we can filter the labels by product. Now that we have our labels, we can write them to the `label` index on Elasticsearch as follows:

```
document_store.write_labels(labels, index="label")
print(f"""Loaded {document_store.get_label_count(index="label")} \
question-answer pairs""")
```

```
Loaded 358 question-answer pairs
```

Next, we need to build up a mapping between our question IDs and corresponding answers that we can pass to the pipeline. To get all the labels, we can use the `get_all_labels_aggregated()` method from the document store that will aggregate all question-answer pairs associated with a unique ID. This method returns a list of `MultiLabel` objects, but in our case we only get one element since we're filtering by question ID. We can build up a list of aggregated labels as follows:

```
labels_agg = document_store.get_all_labels_aggregated(
    index="label",
    open_domain=True,
    aggregate_by_meta=["item_id"]
)
print(len(labels_agg))
```

```
330
```

By peeking at one of these labels we can see that all the answers associated with a given question are aggregated together in a `multiple_answers` field:

```
print(labels_agg[109])
```

```
{'question': 'How does the fan work?', 'multiple_answers': ['the fan is really
really good', "the fan itself isn't super loud. There is an adjustable dial to
change fan speed"], 'is_correct_answer': True, 'is_correct_document': True,
'origin': '5a9b7616541f700f103d21f8ad41bc4b', 'multiple_document_ids': [None,
None], 'multiple_offset_start_in_docs': [None, None], 'no_answer': False,
'model_id': None, 'meta': {'item_id': 'B002MU1ZRS'}}
```

We now have all the ingredients for evaluating the retriever, so let's define a function that feeds each question-answer pair associated with each product to the evaluation pipeline and tracks the correct retrievals in our `pipe` object:

```
def run_pipeline(pipeline, top_k_retriever=10, top_k_reader=4):
    for l in labels_agg:
        _ = pipeline.pipeline.run(
            query=l.question,
            top_k_retriever=top_k_retriever,
            top_k_reader=top_k_reader,
            top_k_eval_documents=top_k_retriever,
            labels=l,
            filters={"item_id": [l.meta["item_id"]], "split": ["test"]})
```

```
run_pipeline(pipe, top_k_retriever=3)
print(f"Recall@3: {pipe.eval_retriever.recall:.2f}")
```

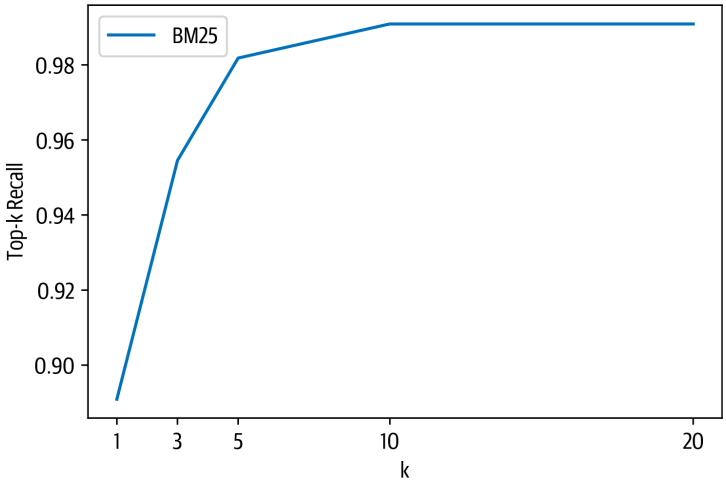
```
Recall@3: 0.95
```

Great, it works! Notice that we picked a specific value for `top_k_retriever` to specify the number of documents to retrieve. In general, increasing this parameter will improve the recall, but at the expense of providing more documents to the reader and slowing down the end-to-end pipeline. To guide our decision on which value to pick, we'll create a function that loops over several k values and compute the recall across the whole test set for each k:

```
def evaluate_retriever(retriever, topk_values = [1,3,5,10,20]):
    topk_results = {}

    for topk in topk_values:
        # Create Pipeline
        p = EvalRetrieverPipeline(retriever)
        # Loop over each question-answers pair in test set
        run_pipeline(p, top_k_retriever=topk)
        # Get metrics
        topk_results[topk] = {"recall": p.eval_retriever.recall}

    return pd.DataFrame.from_dict(topk_results, orient="index")


es_topk_df = evaluate_retriever(es_retriever)
```

If we plot the results, we can see how the recall improves as we increase k:

```
def plot_retriever_eval(dfs, retriever_names):
    fig, ax = plt.subplots()
    for df, retriever_name in zip(dfs, retriever_names):
        df.plot(y="recall", ax=ax, label=retriever_name)
    plt.xticks(df.index)
    plt.ylabel("Top-k Recall")
    plt.xlabel("k")
    plt.show()

plot_retriever_eval([es_topk_df], ["BM25"])
```



From the plot, we can see that there's an inflection point around $k = 5$ and we get almost perfect recall from $k = 10$ onwards. Let's now take a look at retrieving documents with dense vector techniques.

Dense Passage Retrieval

We've seen that we get almost perfect recall when our sparse retriever returns $k = 10$ documents, but can we do better at smaller values of k? The advantage of doing so is that we can pass fewer documents to the reader and thereby reduce the overall latency of our QA pipeline. A well-known limitation of sparse retrievers like BM25 is that they can fail to capture the relevant documents if the user query contains terms that don't match exactly those of the review. One promising alternative is to use dense embeddings to represent the question and document, and the current state of the art is an architecture known as *Dense Passage Retrieval* (DPR).[14] The main idea behind DPR is to use two BERT models as encoders for the question and the passage. As illustrated in Figure 7-10, these encoders map the input text into a d-dimensional vector representation of the `[CLS]` token.

14 V. Karpukhin et al., "Dense Passage Retrieval for Open-Domain Question Answering", (2020).

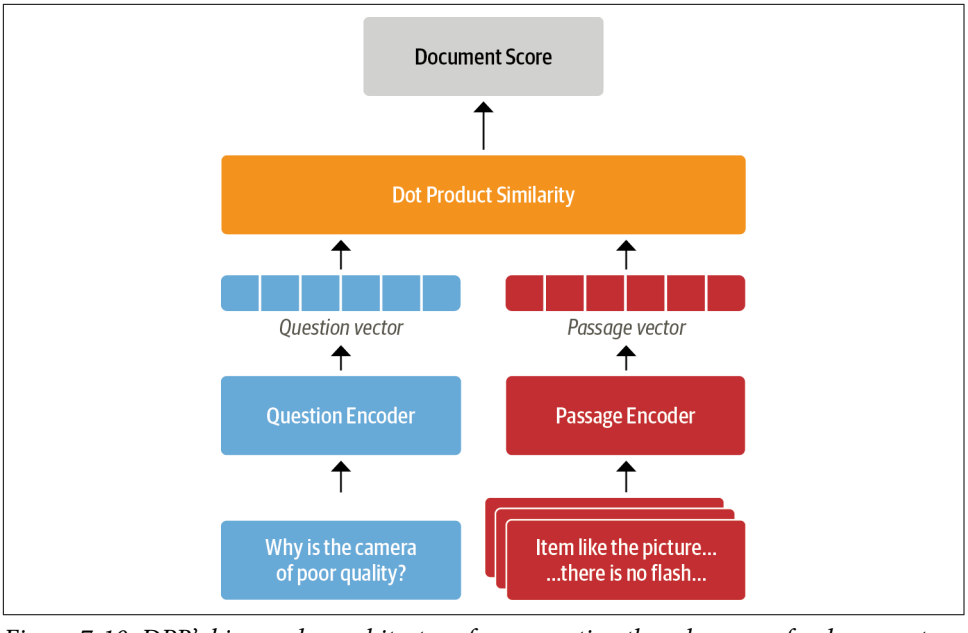


Figure 7-10. DPR's bi-encoder architecture for computing the relevance of a document and query

In Haystack, we can initialize a retriever for DPR in a similar way to what we did for BM25. In addition to specifying the document store, we also need to pick the BERT encoders for the question and passage. These encoders are trained by giving them questions with relevant (positive) passages and irrelevant (negative) passages, where the goal is to learn that relevant question-passage pairs have a higher similarity. For our use case, we'll use encoders that have been fine-tuned on the NQ corpus in this way:

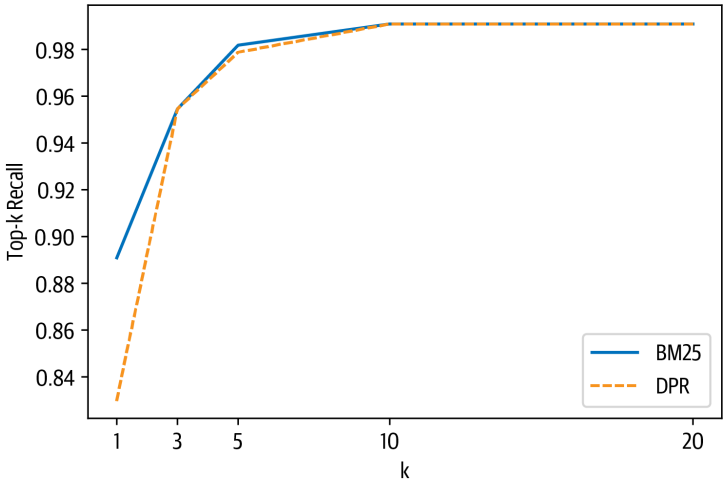
```
from haystack.retriever.dense import DensePassageRetriever

dpr_retriever = DensePassageRetriever(document_store=document_store,
    query_embedding_model="facebook/dpr-question_encoder-single-nq-base",
    passage_embedding_model="facebook/dpr-ctx_encoder-single-nq-base",
    embed_title=False)
```

Here we've also set `embed_title=False` since concatenating the document's title (i.e., `item_id`) doesn't provide any additional information because we filter per product. Once we've initialized the dense retriever, the next step is to iterate over all the indexed documents in our Elasticsearch index and apply the encoders to update the embedding representation. This can be done as follows:

```
document_store.update_embeddings(retriever=dpr_retriever)
```

We're now set to go! We can evaluate the dense retriever in the same way we did for BM25 and compare the top-k recall:

```
dpr_topk_df = evaluate_retriever(dpr_retriever)
plot_retriever_eval([es_topk_df, dpr_topk_df], ["BM25", "DPR"])
```



Here we can see that DPR does not provide a boost in recall over BM25 and saturates around $k = 3$.

Performing similarity search of the embeddings can be sped up by using Facebook's FAISS library as the document store. Similarly, the performance of the DPR retriever can be improved by fine-tuning on the target domain. If you'd like to learn how to fine-tune DPR, check out the Haystack tutorial.

Now that we've explored the evaluation of the retriever, let's turn to evaluating the reader.

Evaluating the Reader

In extractive QA, there are two main metrics that are used for evaluating readers:

Exact Match (EM)
: A binary metric that gives EM = 1 if the characters in the predicted and ground truth answers match exactly, and EM = 0 otherwise. If no answer is expected, the model gets EM = 0 if it predicts any text at all.

F_1-score
: Measures the harmonic mean of the precision and recall.

Let's see how these metrics work by importing some helper functions from FARM and applying them to a simple example:

```
from farm.evaluation.squad_evaluation import compute_f1, compute_exact

pred = "about 6000 hours"
label = "6000 hours"
print(f"EM: {compute_exact(label, pred)}")
print(f"F1: {compute_f1(label, pred)}")
```

```
EM: 0
F1: 0.8
```

Under the hood, these functions first normalize the prediction and label by removing punctuation, fixing whitespace, and converting to lowercase. The normalized strings are then tokenized as a bag-of-words, before finally computing the metric at the token level. From this simple example we can see that EM is a much stricter metric than the F_1-score: adding a single token to the prediction gives an EM of zero. On the other hand, the F_1-score can fail to catch truly incorrect answers. For example, if our predicted answer span is "about 6000 dollars", then we get:

```
pred = "about 6000 dollars"
print(f"EM: {compute_exact(label, pred)}")
print(f"F1: {compute_f1(label, pred)}")
```

```
EM: 0
F1: 0.4
```

Relying on just the F_1-score is thus misleading, and tracking both metrics is a good strategy to balance the trade-off between underestimating (EM) and overestimating (F_1-score) model performance.

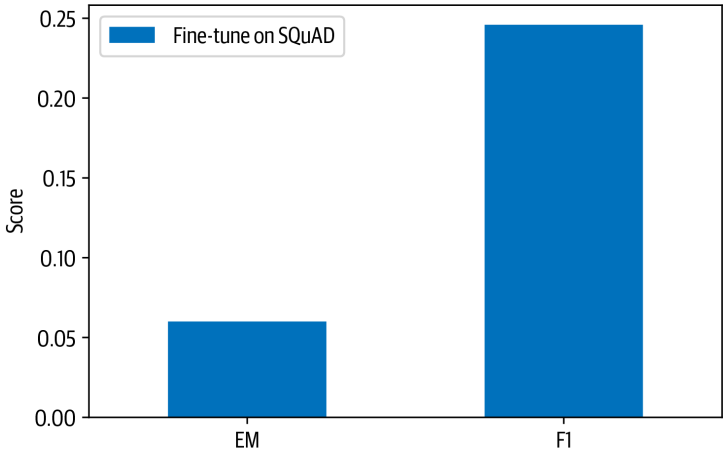
Now in general, there are multiple valid answers per question, so these metrics are calculated for each question-answer pair in the evaluation set, and the best score is selected over all possible answers. The overall EM and F_1 scores for the model are then obtained by averaging over the individual scores of each question-answer pair.

To evaluate the reader we'll create a new pipeline with two nodes: a reader node and a node to evaluate the reader. We'll use the `EvalReader` class that takes the predictions from the reader and computes the corresponding EM and F_1 scores. To compare with the SQuAD evaluation, we'll take the best answers for each query with the `top_1_em` and `top_1_f1` metrics that are stored in `EvalAnswers`:

```
from haystack.eval import EvalAnswers

def evaluate_reader(reader):
    score_keys = ['top_1_em', 'top_1_f1']
    eval_reader = EvalAnswers(skip_incorrect_retrieval=False)
    pipe = Pipeline()
    pipe.add_node(component=reader, name="QAReader", inputs=["Query"])
    pipe.add_node(component=eval_reader, name="EvalReader", inputs=["QAReader"])

    for l in labels_agg:
        doc = document_store.query(l.question,
                                   filters={"question_id":[l.origin]})
        _ = pipe.run(query=l.question, documents=doc, labels=l)

    return {k:v for k,v in eval_reader.__dict__.items() if k in score_keys}

reader_eval = {}
reader_eval["Fine-tune on SQuAD"] = evaluate_reader(reader)
```

Notice that we specified `skip_incorrect_retrieval=False`. This is to ensure that the retriever always passes the context to the reader (as in the SQuAD evaluation). Now that we've run every question through the reader, let's print the scores:

```
def plot_reader_eval(reader_eval):
    fig, ax = plt.subplots()
    df = pd.DataFrame.from_dict(reader_eval)
    df.plot(kind="bar", ylabel="Score", rot=0, ax=ax)
    ax.set_xticklabels(["EM", "F1"])
    plt.legend(loc='upper left')
    plt.show()

plot_reader_eval(reader_eval)
```



OK, it seems that the fine-tuned model performs significantly worse on SubjQA than on SQuAD 2.0, where MiniLM achieves EM and F_1 scores of 76.1 and 79.5, respectively. One reason for the performance drop is that customer reviews are quite different from the Wikipedia articles the SQuAD 2.0 dataset is generated from, and the language they use is often informal. Another factor is likely the inherent subjectivity of our dataset, where both questions and answers differ from the factual information contained in Wikipedia. Let's look at how to fine-tune a model on a dataset to get better results with domain adaptation.

Domain Adaptation

Although models that are fine-tuned on SQuAD will often generalize well to other domains, we've seen that for SubjQA the EM and F_1 scores of our model were much worse than for SQuAD. This failure to generalize has also been observed in other extractive QA datasets and is understood as evidence that transformer models are particularly adept at overfitting to SQuAD.[15] The most straightforward way to improve the reader is by fine-tuning our MiniLM model further on the SubjQA training set. The `FARMReader` has a `train()` method that is designed for this purpose and expects the data to be in SQuAD JSON format, where all the question-answer pairs are grouped together for each item as illustrated in Figure 7-11.

15 D. Yogatama et al., "Learning and Evaluating General Linguistic Intelligence", (2019).

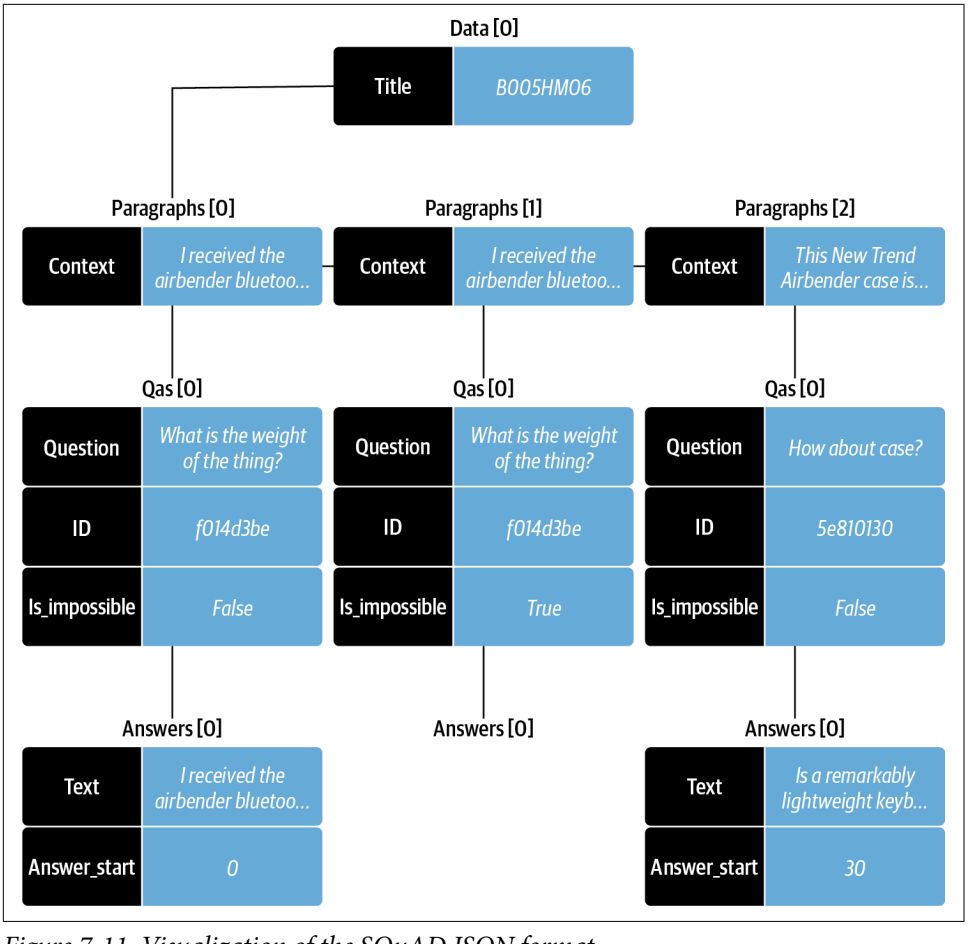


Figure 7-11. Visualization of the SQuAD JSON format

This is quite a complex data format, so we'll need a few functions and some Pandas magic to help us do the conversion. The first thing we need to do is implement a function that can create the `paragraphs` array associated with each product ID. Each element in this array contains a single context (i.e., review) and a `qas` array of question-answer pairs. Here's a function that builds up the `paragraphs` array:

```
def create_paragraphs(df):
    paragraphs = []
    id2context = dict(zip(df["review_id"], df["context"]))
    for review_id, review in id2context.items():
        qas = []
        # Filter for all question-answer pairs about a specific context
        review_df = df.query(f"review_id == '{review_id}'")
        id2question = dict(zip(review_df["id"], review_df["question"]))
        # Build up the qas array
```

```
        for qid, question in id2question.items():
            # Filter for a single question ID
            question_df = df.query(f"id == '{qid}'").to_dict(orient="list")
            ans_start_idxs = question_df["answers.answer_start"][0].tolist()
            ans_text = question_df["answers.text"][0].tolist()
            # Fill answerable questions
            if len(ans_start_idxs):
                answers = [
                    {"text": text, "answer_start": answer_start}
                    for text, answer_start in zip(ans_text, ans_start_idxs)]
                is_impossible = False
            else:
                answers = []
                is_impossible = True
            # Add question-answer pairs to qas
            qas.append({"question": question, "id": qid,
                        "is_impossible": is_impossible, "answers": answers})
        # Add context and question-answer pairs to paragraphs
        paragraphs.append({"qas": qas, "context": review})
    return paragraphs
```

Now, when we apply to the rows of a `DataFrame` associated with a single product ID, we get the SQuAD format:

```
product = dfs["train"].query("title == 'B00001P4ZH'")
create_paragraphs(product)
```

```
[{'qas': [{'question': 'How is the bass?',
    'id': '2543d296da9766d8d17d040ecc781699',
    'is_impossible': True,
    'answers': []}],
  'context': 'I have had Koss headphones ...',
    'id': 'd476830bf9282e2b9033e2bb44bbb995',
    'is_impossible': False,
    'answers': [{'text': 'Bass is weak as expected', 'answer_start': 1302},
     {'text': 'Bass is weak as expected, even with EQ adjusted up',
      'answer_start': 1302}]}],
  'context': 'To anyone who hasn\'t tried all ...'},
 {'qas': [{'question': 'How is the bass?',
    'id': '455575557886d6dfeea5aa19577e5de4',
    'is_impossible': False,
    'answers': [{'text': 'The only fault in the sound is the bass',
      'answer_start': 650}]}],
  'context': "I have had many sub-$100 headphones ..."}]
```

The final step is to then apply this function to each product ID in the `DataFrame` of each split. The following `convert_to_squad()` function does this trick and stores the result in an *electronics-{split}.json* file:

```
import json

def convert_to_squad(dfs):
    for split, df in dfs.items():
```

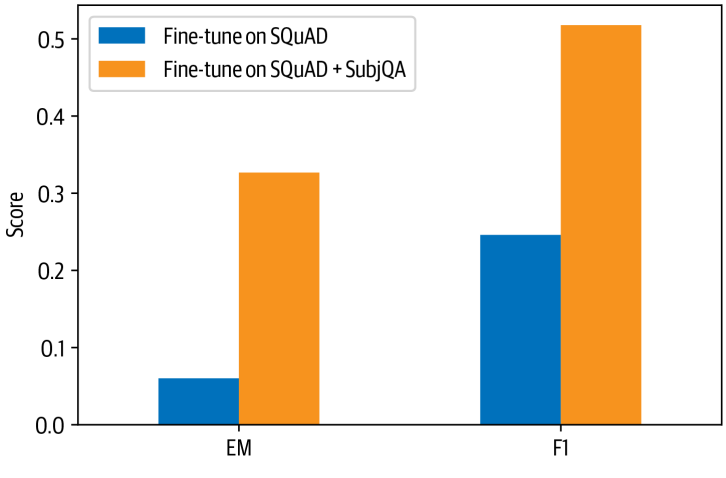
```
        subjqa_data = {}
        # Create `paragraphs` for each product ID
        groups = (df.groupby("title").apply(create_paragraphs)
            .to_frame(name="paragraphs").reset_index())
        subjqa_data["data"] = groups.to_dict(orient="records")
        # Save the result to disk
        with open(f"electronics-{split}.json", "w+", encoding="utf-8") as f:
            json.dump(subjqa_data, f)

convert_to_squad(dfs)
```

Now that we have the splits in the right format, let's fine-tune our reader by specifying the locations of the train and dev splits, along with where to save the fine-tuned model:

```
train_filename = "electronics-train.json"
dev_filename = "electronics-validation.json"

reader.train(data_dir=".", use_gpu=True, n_epochs=1, batch_size=16,
             train_filename=train_filename, dev_filename=dev_filename)
```

With the reader fine-tuned, let's now compare its performance on the test set against our baseline model:

```
reader_eval["Fine-tune on SQuAD + SubjQA"] = evaluate_reader(reader)
plot_reader_eval(reader_eval)
```



Wow, domain adaptation has increased our EM score by a factor of six and more than doubled the F_1-score! At this point, you might be wondering why we didn't just fine-tune a pretrained language model directly on the SubjQA training set. One reason is that we only have 1,295 training examples in SubjQA while SQuAD has over 100,000, so we might run into challenges with overfitting. Nevertheless, let's take a look at what naive fine-tuning produces. For a fair comparison, we'll use the same language model

that was used for fine-tuning our baseline on SQuAD. As before, we'll load up the model with the `FARMReader`:

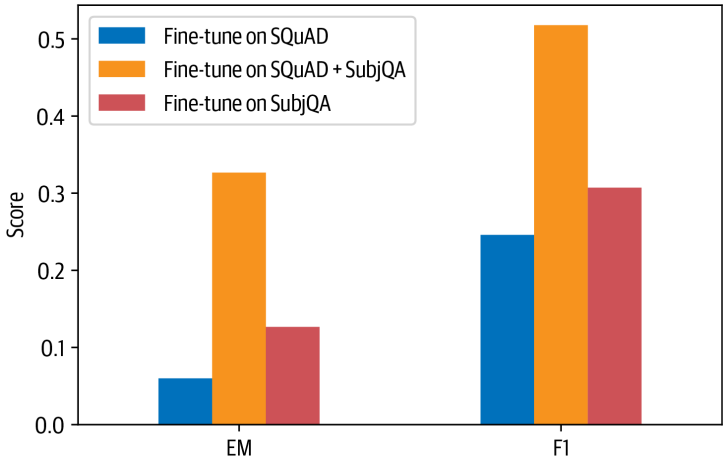
```
minilm_ckpt = "microsoft/MiniLM-L12-H384-uncased"
minilm_reader = FARMReader(model_name_or_path=minilm_ckpt, progress_bar=False,
                           max_seq_len=max_seq_length, doc_stride=doc_stride,
                           return_no_answer=True)
```

Next, we fine-tune for one epoch:

```
minilm_reader.train(data_dir=".", use_gpu=True, n_epochs=1, batch_size=16,
            train_filename=train_filename, dev_filename=dev_filename)
```

and include the evaluation on the test set:

```
reader_eval["Fine-tune on SubjQA"] = evaluate_reader(minilm_reader)
plot_reader_eval(reader_eval)
```



We can see that fine-tuning the language model directly on SubjQA results in considerably worse performance than fine-tuning on SQuAD and SubjQA.

When dealing with small datasets, it is best practice to use cross-validation when evaluating transformers as they can be prone to overfitting. You can find an example of how to perform cross-validation with SQuAD-formatted datasets in the FARM repository.

Evaluating the Whole QA Pipeline

Now that we've seen how to evaluate the reader and retriever components individually, let's tie them together to measure the overall performance of our pipeline. To do so, we'll need to augment our retriever pipeline with nodes for the reader and its

evaluation. We've seen that we get almost perfect recall at $k = 10$, so we can fix this value and assess the impact this has on the reader's performance (since it will now receive multiple contexts per query compared to the SQuAD-style evaluation):

```
# Initialize retriever pipeline
pipe = EvalRetrieverPipeline(es_retriever)
# Add nodes for reader
eval_reader = EvalAnswers()
pipe.pipeline.add_node(component=reader, name="QAReader",
                       inputs=["EvalRetriever"])
pipe.pipeline.add_node(component=eval_reader, name="EvalReader",
                       inputs=["QAReader"])
# Evaluate!
run_pipeline(pipe)
# Extract metrics from reader
reader_eval["QA Pipeline (top-1)"] = {
    k:v for k,v in eval_reader.__dict__.items()
    if k in ["top_1_em", "top_1_f1"]}
```

We can then compare the top 1 EM and F_1 scores for the model to predict an answer in the documents returned by the retriever in Figure 7-12.

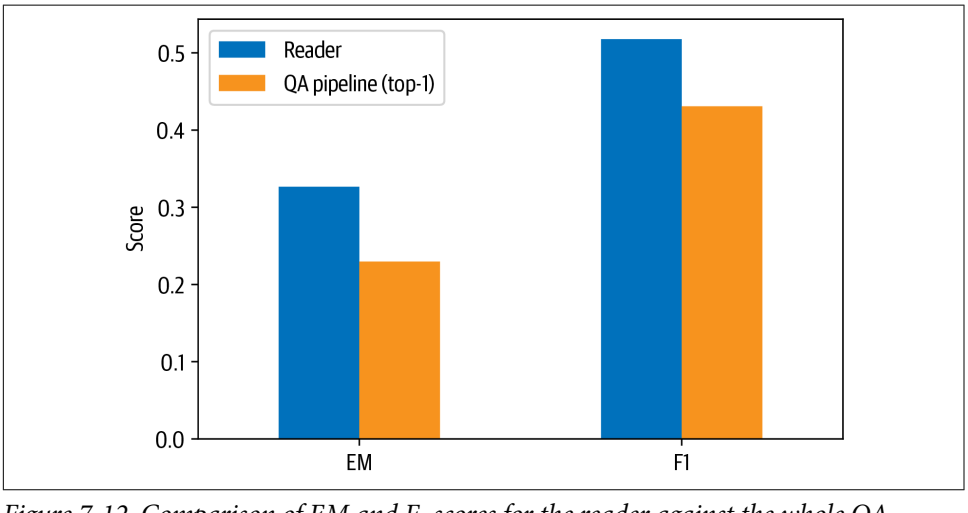


Figure 7-12. Comparison of EM and F_1 scores for the reader against the whole QA pipeline

From this plot we can see the effect that the retriever has on the overall performance. In particular, there is an overall degradation compared to matching the question-context pairs, as is done in the SQuAD-style evaluation. This can be circumvented by increasing the number of possible answers that the reader is allowed to predict.

Until now we have only extracted answer spans from the context, but in general it could be that bits and pieces of the answer are scattered throughout the document

and we would like our model to synthesize these fragments into a single coherent answer. Let's have a look at how we can use generative QA to succeed at this task.

Going Beyond Extractive QA

One interesting alternative to extracting answers as spans of text in a document is to generate them with a pretrained language model. This approach is often referred to as *abstractive* or *generative QA* and has the potential to produce better-phrased answers that synthesize evidence across multiple passages. Although less mature than extractive QA, this is a fast-moving field of research, so chances are that these approaches will be widely adopted in industry by the time you are reading this! In this section we'll briefly touch on the current state of the art: *retrieval-augmented generation* (RAG).[16]

RAG extends the classic retriever-reader architecture that we've seen in this chapter by swapping the reader for a *generator* and using DPR as the retriever. The generator is a pretrained sequence-to-sequence transformer like T5 or BART that receives latent vectors of documents from DPR and then iteratively generates an answer based on the query and these documents. Since DPR and the generator are differentiable, the whole process can be fine-tuned end-to-end as illustrated in Figure 7-13.

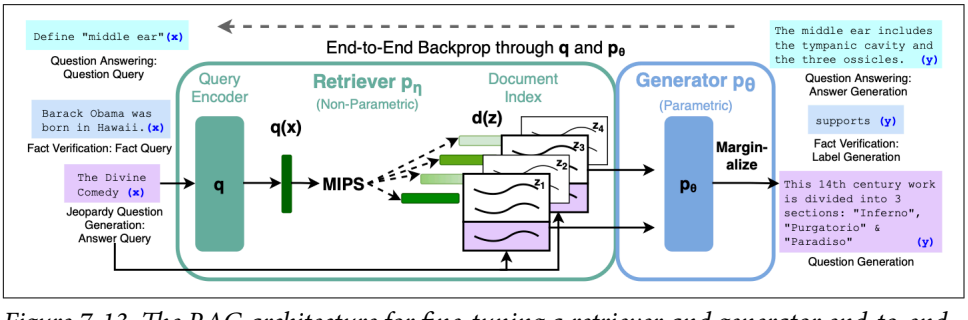


Figure 7-13. The RAG architecture for fine-tuning a retriever and generator end-to-end (courtesy of Ethan Perez)

To show RAG in action we'll use the `DPRetriever` from earlier, so we just need to instantiate a generator. There are two types of RAG models to choose from:

RAG-Sequence
: Uses the same retrieved document to generate the complete answer. In particular, the top k documents from the retriever are fed to the generator, which produces an output sequence for each document, and the result is marginalized to obtain the best answer.

16 P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks", (2020).

RAG-Token
: Can use a different document to generate each token in the answer. This allows the generator to synthesize evidence from multiple documents.

Since RAG-Token models tend to perform better than RAG-Sequence ones, we'll use the token model that was fine-tuned on NQ as our generator. Instantiating a generator in Haystack is similar to instantiating the reader, but instead of specifying the `max_seq_length` and `doc_stride` parameters for a sliding window over the contexts, we specify hyperparameters that control the text generation:

```
from haystack.generator.transformers import RAGenerator

generator = RAGenerator(model_name_or_path="facebook/rag-token-nq",
                        embed_title=False, num_beams=5)
```

Here `num_beams` specifies the number of beams to use in beam search (text generation is covered at length in Chapter 5). As we did with the DPR retriever, we don't embed the document titles since our corpus is always filtered per product ID.

The next thing to do is tie together the retriever and generator using Haystack's `GenerativeQAPipeline`:

```
from haystack.pipeline import GenerativeQAPipeline

pipe = GenerativeQAPipeline(generator=generator, retriever=dpr_retriever)
```

In RAG, both the query encoder and the generator are trained end-to-end, while the context encoder is frozen. In Haystack, the `GenerativeQAPipeline` uses the query encoder from `RAGenerator` and the context encoder from `DensePassageRetriever`.

Let's now give RAG a spin by feeding in some queries about the Amazon Fire tablet from before. To simplify the querying, we'll write a simple function that takes the query and prints out the top answers:

```
def generate_answers(query, top_k_generator=3):
    preds = pipe.run(query=query, top_k_generator=top_k_generator,
                     top_k_retriever=5, filters={"item_id":["B0074BW614"]})
    print(f"Question: {preds['query']} \n")
    for idx in range(top_k_generator):
        print(f"Answer {idx+1}: {preds['answers'][idx]['answer']}")
```

OK, now we're ready to give it a test:

```
generate_answers(query)
```

```
Question: Is it good for reading?

Answer 1:  the screen is absolutely beautiful
```

```
Answer 2:  the Screen is absolutely beautiful
Answer 3:  Kindle fire
```

This result isn't too bad for an answer, but it does suggest that the subjective nature of the question is confusing the generator. Let's try with something a bit more factual:

```
generate_answers("What is the main drawback?")
```

```
Question: What is the main drawback?

Answer 1:  the price
Answer 2:  no flash support
Answer 3:  the cost
```

This is more sensible! To get better results we could fine-tune RAG end-to-end on SubjQA; we'll leave this as an exercise, but if you're interested in exploring it there are scripts in the 🤗 Transformers repository to help you get started.

Conclusion

Well, that was a whirlwind tour of QA, and you probably have many more questions that you'd like answered (pun intended!). In this chapter, we discussed two approaches to QA (extractive and generative) and examined two different retrieval algorithms (BM25 and DPR). Along the way, we saw that domain adaptation can be a simple technique to boost the performance of our QA system by a significant margin, and we looked at a few of the most common metrics that are used for evaluating such systems. Although we focused on closed-domain QA (i.e., a single domain of electronic products), the techniques in this chapter can easily be generalized to the open-domain case; we recommend reading Cloudera's excellent Fast Forward QA series to see what's involved.

Deploying QA systems in the wild can be a tricky business to get right, and our experience is that a significant part of the value comes from first providing end users with useful search capabilities, followed by an extractive component. In this respect, the reader can be used in novel ways beyond answering on-demand user queries. For example, researchers at Grid Dynamics were able to use their reader to automatically extract a set of pros and cons for each product in a client's catalog. They also showed that a reader can be used to extract named entities in a zero-shot fashion by creating queries like "What kind of camera?" Given its infancy and subtle failure modes, we recommend exploring generative QA only once the other two approaches have been exhausted. This "hierarchy of needs" for tackling QA problems is illustrated in Figure 7-14.

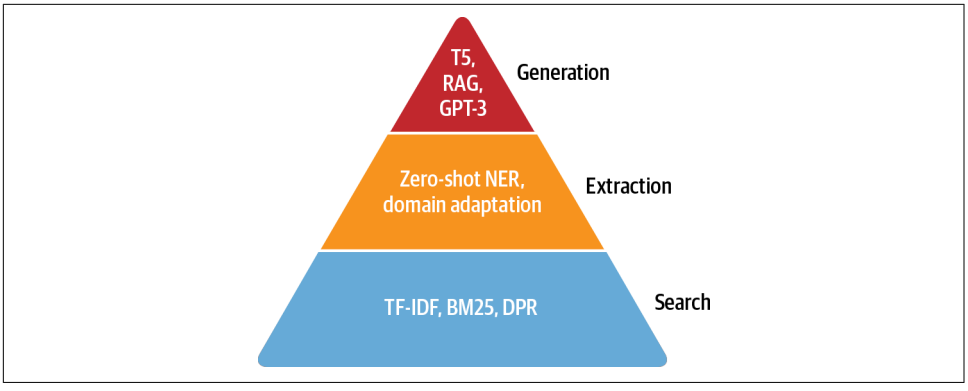


Figure 7-14. The QA hierarchy of needs

Looking ahead, one exciting research area is *multimodal QA*, which involves QA over multiple modalities like text, tables, and images. As described in the MultiModalQA benchmark,[17] such systems could enable users to answer complex questions that integrate information across different modalities, like "When was the famous painting with two touching fingers completed?" Another area with practical business applications is QA over a *knowledge graph*, where the nodes of the graph correspond to real-world entities and their relations are defined by the edges. By encoding factoids as (*subject*, *predicate*, *object*) triples, one can use the graph to answer questions about a missing element. For an example that combines transformers with knowledge graphs, see the Haystack tutorials. One more promising direction is *automatic question generation* as a way to do some form of unsupervised/weakly supervised training using unlabeled data or data augmentation. Two recent examples include the papers on the Probably Answered Questions (PAQ) benchmark and synthetic data augmentation for cross-lingual settings.[18]

In this chapter we've seen that in order to successfully use QA models for real-world use cases we need to apply a few tricks, such as implementing a fast retrieval pipeline to make predictions in near real time. Still, applying a QA model to a handful of preselected documents can take a couple of seconds on production hardware. Although this may not sound like much, imagine how different your experience would be if you had to wait a few seconds to get the results of a Google search—a few seconds of wait time can decide the fate of your transformer-powered application. In the next chapter we'll have a look at a few methods to accelerate model predictions further.

17 A. Talmor et al., "MultiModalQA: Complex Question Answering over Text, Tables and Images", (2021).

18 P. Lewis et al., "PAQ: 65 Million Probably-Asked Questions and What You Can Do with Them", (2021); A. Riabi et al., "Synthetic Data Augmentation for Zero-Shot Cross-Lingual Question Answering", (2020).

CHAPTER 8

Making Transformers Efficient in Production

In the previous chapters, you've seen how transformers can be fine-tuned to produce great results on a wide range of tasks. However, in many situations accuracy (or whatever metric you're optimizing for) is not enough; your state-of-the-art model is not very useful if it's too slow or large to meet the business requirements of your application. An obvious alternative is to train a faster and more compact model, but the reduction in model capacity is often accompanied by a degradation in performance. So what can you do when you need a fast, compact, yet highly accurate model?

In this chapter we will explore four complementary techniques that can be used to speed up the predictions and reduce the memory footprint of your transformer models: *knowledge distillation*, *quantization*, *pruning*, and *graph optimization* with the Open Neural Network Exchange (ONNX) format and ONNX Runtime (ORT). We'll also see how some of these techniques can be combined to produce significant performance gains. For example, this was the approach taken by the Roblox engineering team in their article "How We Scaled Bert to Serve 1+ Billion Daily Requests on CPUs", who as shown in Figure 8-1 found that combining knowledge distillation and quantization enabled them to improve the latency and throughput of their BERT classifier by over a factor of 30!

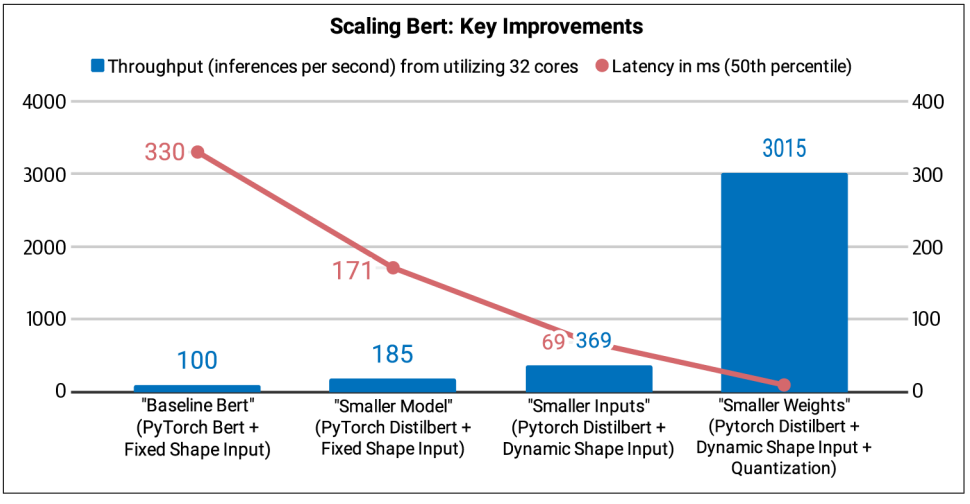


Figure 8-1. How Roblox scaled BERT with knowledge distillation, dynamic padding, and weight quantization (photo courtesy of Roblox employees Quoc N. Le and Kip Kaehler)

To illustrate the benefits and trade-offs associated with each technique, we'll use intent detection as a case study; this is an important component of text-based assistants, where low latencies are critical for maintaining a conversation in real time. Along the way you'll learn how to create custom trainers, perform efficient hyperparameter search, and gain a sense of what it takes to implement cutting-edge research with 🤗 Transformers. Let's dive in!

Intent Detection as a Case Study

Let's suppose that we're trying to build a text-based assistant for our company's call center so that customers can request their account balance or make bookings without needing to speak with a human agent. In order to understand the goals of a customer, our assistant will need to be able to classify a wide variety of natural language text into a set of predefined actions or *intents*. For example, a customer might send a message like the following about an upcoming trip:

> Hey, I'd like to rent a vehicle from Nov 1st to Nov 15th in Paris and I need a 15 passenger van

and our intent classifier could automatically categorize this as a *Car Rental* intent, which then triggers an action and response. To be robust in a production environment, our classifier will also need to be able to handle *out-of-scope* queries, where a customer makes a query that doesn't belong to any of the predefined intents and the system should yield a fallback response. For example, in the second case shown in Figure 8-2, a customer asks a question about sports (which is out of scope), and the text assistant mistakenly classifies it as one of the known in-scope intents and returns

the payday response. In the third case, the text assistant has been trained to detect out-of-scope queries (usually labeled as a separate class) and informs the customer about which topics it can answer questions about.

Figure 8-2. Three exchanges between a human (right) and a text-based assistant (left) for personal finance (courtesy of Stefan Larson et al.)

As a baseline, we've fine-tuned a BERT-base model that achieves around 94% accuracy on the CLINC150 dataset.[1] This dataset includes 22,500 in-scope queries across 150 intents and 10 domains like banking and travel, and also includes 1,200 out-of-scope queries that belong to an `oos` intent class. In practice we would also gather our own in-house dataset, but using public data is a great way to iterate quickly and generate preliminary results.

To get started, let's download our fine-tuned model from the Hugging Face Hub and wrap it in a pipeline for text classification:

```
from transformers import pipeline

bert_ckpt = "transformersbook/bert-base-uncased-finetuned-clinc"
pipe = pipeline("text-classification", model=bert_ckpt)
```

Now that we have a pipeline, we can pass a query to get the predicted intent and confidence score from the model:

1 S. Larson et al., "An Evaluation Dataset for Intent Classification and Out-of-Scope Prediction", (2019).

```
query = """Hey, I'd like to rent a vehicle from Nov 1st to Nov 15th in
Paris and I need a 15 passenger van"""
pipe(query)
```

```
[{'label': 'car_rental', 'score': 0.549003541469574}]
```

Great, the `car_rental` intent makes sense. Let's now look at creating a benchmark that we can use to evaluate the performance of our baseline model.

Creating a Performance Benchmark

Like other machine learning models, deploying transformers in production environments involves a trade-off among several constraints, the most common being:[2]

Model performance
: How well does our model perform on a well-crafted test set that reflects production data? This is especially important when the cost of making errors is large (and best mitigated with a human in the loop), or when we need to run inference on millions of examples and small improvements to the model metrics can translate into large gains in aggregate.

Latency
: How fast can our model deliver predictions? We usually care about latency in real-time environments that deal with a lot of traffic, like how Stack Overflow needed a classifier to quickly detect unwelcome comments on the website.

Memory
: How can we deploy billion-parameter models like GPT-2 or T5 that require gigabytes of disk storage and RAM? Memory plays an especially important role in mobile or edge devices, where a model has to generate predictions without access to a powerful cloud server.

Failing to address these constraints can have a negative impact on the user experience of your application. More commonly, it can lead to ballooning costs from running expensive cloud servers that may only need to handle a few requests. To explore how each of these constraints can be optimized with various compression techniques, let's begin by creating a simple benchmark that measures each quantity for a given pipeline and test set. A skeleton of what we'll need is given by the following class:

```
class PerformanceBenchmark:
    def __init__(self, pipeline, dataset, optim_type="BERT baseline"):
        self.pipeline = pipeline
```

2 As described by Emmanuel Ameisen in *Building Machine Learning Powered Applications* (O'Reilly), business or product metrics are the *most* important ones to consider. After all, it doesn't matter how accurate your model is if it doesn't solve a problem your business cares about. In this chapter we'll assume that you have already defined the metrics that matter for your application and focus on optimizing the model metrics.

```
        self.dataset = dataset
        self.optim_type = optim_type

    def compute_accuracy(self):
        # We'll define this later
        pass

    def compute_size(self):
        # We'll define this later
        pass

    def time_pipeline(self):
        # We'll define this later
        pass

    def run_benchmark(self):
        metrics = {}
        metrics[self.optim_type] = self.compute_size()
        metrics[self.optim_type].update(self.time_pipeline())
        metrics[self.optim_type].update(self.compute_accuracy())
        return metrics
```

We've defined an `optim_type` parameter to keep track of the different optimization techniques that we'll cover in this chapter. We'll use the `run_benchmark()` method to collect all the metrics in a dictionary, with keys given by `optim_type`.

Let's now put some flesh on the bones of this class by computing the model accuracy on the test set. First we need some data to test on, so let's download the CLINC150 dataset that was used to fine-tune our baseline model. We can get the dataset from the Hub with 🤗 Datasets as follows:

```
from datasets import load_dataset

clinc = load_dataset("clinc_oos", "plus")
```

Here, the `plus` configuration refers to the subset that contains the out-of-scope training examples. Each example in the CLINC150 dataset consists of a query in the `text` column and its corresponding intent. We'll use the test set to benchmark our models, so let's take a look at one of the dataset's examples:

```
sample = clinc["test"][42]
sample
```

```
{'intent': 133, 'text': 'transfer $100 from my checking to saving account'}
```

The intents are provided as IDs, but we can easily get the mapping to strings (and vice versa) by accessing the `features` attribute of the dataset:

```
intents = clinc["test"].features["intent"]
intents.int2str(sample["intent"])
```

```
'transfer'
```

Now that we have a basic understanding of the contents in the CLINC150 dataset, let's implement the `compute_accuracy()` method of `PerformanceBenchmark`. Since the dataset is balanced across the intent classes, we'll use accuracy as our metric. We can load this metric with 🤗 Datasets as follows:

```
from datasets import load_metric

accuracy_score = load_metric("accuracy")
```

The accuracy metric expects the predictions and references (i.e., the ground truth labels) to be integers. We can use the pipeline to extract the predictions from the `text` field and then use the `str2int()` method of our `intents` object to map each prediction to its corresponding ID. The following code collects all the predictions and labels in lists before returning the accuracy on the dataset. Let's also add it to our `PerformanceBenchmark` class:

```
def compute_accuracy(self):
    """This overrides the PerformanceBenchmark.compute_accuracy() method"""
    preds, labels = [], []
    for example in self.dataset:
        pred = self.pipeline(example["text"])[0]["label"]
        label = example["intent"]
        preds.append(intents.str2int(pred))
        labels.append(label)
    accuracy = accuracy_score.compute(predictions=preds, references=labels)
    print(f"Accuracy on test set - {accuracy['accuracy']:.3f}")
    return accuracy

PerformanceBenchmark.compute_accuracy = compute_accuracy
```

Next, let's compute the size of our model by using the `torch.save()` function from PyTorch to serialize the model to disk. Under the hood, `torch.save()` uses Python's `pickle` module and can be used to save anything from models to tensors to ordinary Python objects. In PyTorch, the recommended way to save a model is by using its `state_dict`, which is a Python dictionary that maps each layer in a model to its learnable parameters (i.e., weights and biases). Let's see what is stored in the `state_dict` of our baseline model:

```
list(pipe.model.state_dict().items())[42]
```

```
('bert.encoder.layer.2.attention.self.value.weight',
 tensor([[-1.0526e-02, -3.2215e-02,  2.2097e-02,  ..., -6.0953e-03,
           4.6521e-03,  2.9844e-02],
         [-1.4964e-02, -1.0915e-02,  5.2396e-04,  ...,  3.2047e-05,
          -2.6890e-02, -2.1943e-02],
         [-2.9640e-02, -3.7842e-03, -1.2582e-02,  ..., -1.0917e-02,
           3.1152e-02, -9.7786e-03],
         ...,
         [-1.5116e-02, -3.3226e-02,  4.2063e-02,  ..., -5.2652e-03,
           1.1093e-02,  2.9703e-03],
```

```
          [-3.6809e-02,  5.6848e-02, -2.6544e-02,  ..., -4.0114e-02,
            6.7487e-03,  1.0511e-03],
          [-2.4961e-02,  1.4747e-03, -5.4271e-02,  ...,  2.0004e-02,
            2.3981e-02, -4.2880e-02]]))
```

We can clearly see that each key/value pair corresponds to a specific layer and tensor in BERT. So if we save our model with:

```
torch.save(pipe.model.state_dict(), "model.pt")
```

we can then use the `Path.stat()` function from Python's `pathlib` module to get information about the underlying files. In particular, `Path("model.pt").stat().st_size` will give us the model size in bytes. Let's put this all together in the `compute_size()` function and add it to `PerformanceBenchmark`:

```
import torch
from pathlib import Path

def compute_size(self):
    """This overrides the PerformanceBenchmark.compute_size() method"""
    state_dict = self.pipeline.model.state_dict()
    tmp_path = Path("model.pt")
    torch.save(state_dict, tmp_path)
    # Calculate size in megabytes
    size_mb = Path(tmp_path).stat().st_size / (1024 * 1024)
    # Delete temporary file
    tmp_path.unlink()
    print(f"Model size (MB) - {size_mb:.2f}")
    return {"size_mb": size_mb}

PerformanceBenchmark.compute_size = compute_size
```

Finally let's implement the `time_pipeline()` function so that we can time the average latency per query. For this application, latency refers to the time it takes to feed a text query to the pipeline and return the predicted intent from the model. Under the hood the pipeline also tokenizes the text, but this is around one thousand times faster than generating the predictions and thus adds a negligible contribution to the overall latency. A simple way to measure the execution time of a code snippet is to use the `perf_counter()` function from Python's `time` module. This function has a better time resolution than the `time.time()` function and is well suited for getting precise results.

We can use `perf_counter()` to time our pipeline by passing our test query and calculating the time difference in milliseconds between the start and end:

```
from time import perf_counter

for _ in range(3):
    start_time = perf_counter()
    _ = pipe(query)
```

```
        latency = perf_counter() - start_time
        print(f"Latency (ms) - {1000 * latency:.3f}")

    Latency (ms) - 85.367
    Latency (ms) - 85.241
    Latency (ms) - 87.275
```

These results exhibit quite some spread in the latencies and suggest that timing a single pass through the pipeline can give wildly different results each time we run the code. So instead, we'll collect the latencies over many runs and then use the resulting distribution to calculate the mean and standard deviation, which will give us an idea about the spread in values. The following code does what we need and includes a phase to warm up the CPU before performing the actual timed run:

```
    import numpy as np

    def time_pipeline(self, query="What is the pin number for my account?"):
        """This overrides the PerformanceBenchmark.time_pipeline() method"""
        latencies = []
        # Warmup
        for _ in range(10):
            _ = self.pipeline(query)
        # Timed run
        for _ in range(100):
            start_time = perf_counter()
            _ = self.pipeline(query)
            latency = perf_counter() - start_time
            latencies.append(latency)
        # Compute run statistics
        time_avg_ms = 1000 * np.mean(latencies)
        time_std_ms = 1000 * np.std(latencies)
        print(f"Average latency (ms) - {time_avg_ms:.2f} +\- {time_std_ms:.2f}")
        return {"time_avg_ms": time_avg_ms, "time_std_ms": time_std_ms}

    PerformanceBenchmark.time_pipeline = time_pipeline
```

To keeps things simple, we'll use the same `query` value to benchmark all our models. In general, the latency will depend on the query length, and a good practice is to benchmark your models with queries that they're likely to encounter in production environments.

Now that our `PerformanceBenchmark` class is complete, let's give it a spin! Let's start by benchmarking our BERT baseline. For the baseline model, we just need to pass the pipeline and the dataset we wish to perform the benchmark on. We'll collect the results in the `perf_metrics` dictionary to keep track of each model's performance:

```
    pb = PerformanceBenchmark(pipe, clinc["test"])
    perf_metrics = pb.run_benchmark()

    Model size (MB) - 418.16
    Average latency (ms) - 54.20 +\- 1.91
    Accuracy on test set - 0.867
```

Now that we have a reference point, let's look at our first compression technique: knowledge distillation.

The average latency values will differ depending on what type of hardware you are running on. For example, you can usually get better performance by running inference on a GPU since it enables batch processing. For the purposes of this chapter, what's important is the relative difference in latencies between models. Once we have determined the best-performing model, we can then explore different backends to reduce the absolute latency if needed.

Making Models Smaller via Knowledge Distillation

Knowledge distillation is a general-purpose method for training a smaller *student* model to mimic the behavior of a slower, larger, but better-performing *teacher*. Originally introduced in 2006 in the context of ensemble models,[3] it was later popularized in a famous 2015 paper that generalized the method to deep neural networks and applied it to image classification and automatic speech recognition.[4]

Given the trend toward pretraining language models with ever-increasing parameter counts (the largest at the time of writing having over one trillion parameters),[5] knowledge distillation has also become a popular strategy to compress these huge models and make them more suitable for building practical applications.

Knowledge Distillation for Fine-Tuning

So how is knowledge actually "distilled" or transferred from the teacher to the student during training? For supervised tasks like fine-tuning, the main idea is to augment the ground truth labels with a distribution of "soft probabilities" from the teacher which provide complementary information for the student to learn from. For example, if our BERT-base classifier assigns high probabilities to multiple intents, then this could be a sign that these intents lie close to each other in the feature space. By training the student to mimic these probabilities, the goal is to distill some of this "dark knowledge"[6] that the teacher has learned—that is, knowledge that is not available from the labels alone.

3 C. Buciluǎ et al., "Model Compression," *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (August 2006): 535–541, *https://doi.org/10.1145/1150402.1150464*.

4 G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network", (2015).

5 W. Fedus, B. Zoph, and N. Shazeer, "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity", (2021).

6 Geoff Hinton coined this term in a talk to refer to the observation that softened probabilities reveal the hidden knowledge of the teacher.

Mathematically, the way this works is as follows. Suppose we feed an input sequence x to the teacher to generate a vector of logits $\mathbf{z}(x) = [z_1(x), ..., z_N(x)]$. We can convert these logits into probabilities by applying a softmax function:

$$\frac{\exp\left(z_i(x)\right)}{\sum_j \exp\left(z_i(x)\right)}$$

This isn't quite what we want, though, because in many cases the teacher will assign a high probability to one class, with all other class probabilities close to zero. When that happens, the teacher doesn't provide much additional information beyond the ground truth labels, so instead we "soften" the probabilities by scaling the logits with a temperature hyperparameter T before applying the softmax:[7]

$$p_i(x) = \frac{\exp\left(z_i(x)/T\right)}{\sum_j \exp\left(z_i(x)/T\right)}$$

As shown in Figure 8-3, higher values of T produce a softer probability distribution over the classes and reveal much more information about the decision boundary that the teacher has learned for each training example. When $T = 1$ we recover the original softmax distribution.

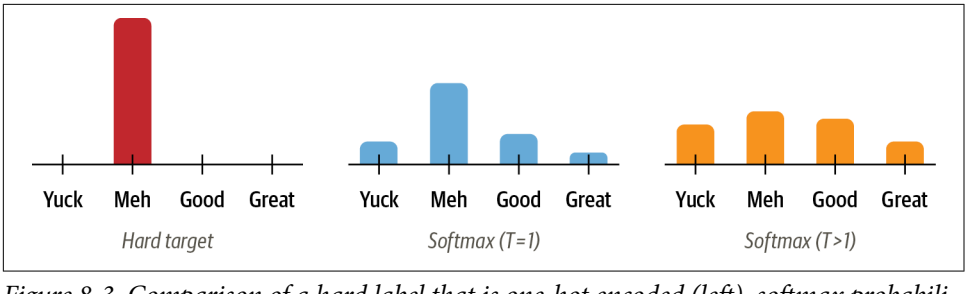


Figure 8-3. Comparison of a hard label that is one-hot encoded (left), softmax probabilities (middle), and softened class probabilities (right)

Since the student also produces softened probabilities $q_i(x)$ of its own, we can use the Kullback–Leibler (KL) divergence to measure the difference between the two probability distributions:

$$D_{KL}(p, q) = \sum_i p_i(x) \log \frac{p_i(x)}{q_i(x)}$$

7 We also encountered temperature in the context of text generation in Chapter 5.

With the KL divergence we can calculate how much is lost when we approximate the probability distribution of the teacher with the student. This allows us to define a knowledge distillation loss:

$$L_{KD} = T^2 D_{KL}$$

where T^2 is a normalization factor to account for the fact that the magnitude of the gradients produced by soft labels scales as $1/T^2$. For classification tasks, the student loss is then a weighted average of the distillation loss with the usual cross-entropy loss L_{CE} of the ground truth labels:

$$L_{\text{student}} = \alpha L_{CE} + (1 - \alpha) L_{KD}$$

where α is a hyperparameter that controls the relative strength of each loss. A diagram of the whole process is shown in Figure 8-4; the temperature is set to 1 at inference time to recover the standard softmax probabilities.

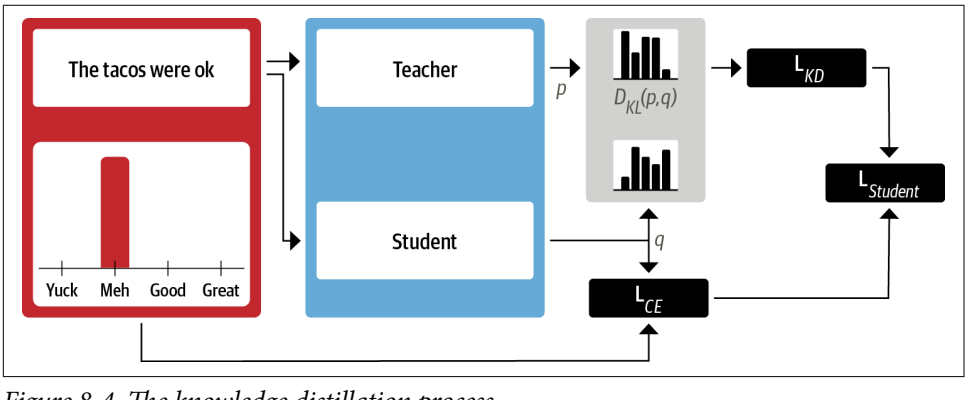


Figure 8-4. The knowledge distillation process

Knowledge Distillation for Pretraining

Knowledge distillation can also be used during pretraining to create a general-purpose student that can be subsequently fine-tuned on downstream tasks. In this case, the teacher is a pretrained language model like BERT, which transfers its knowledge about masked language modeling to the student. For example, in the DistilBERT paper,[8] the masked language modeling loss L_{mlm} is augmented with a term from knowledge distillation and a cosine embedding loss $L_{cos} = 1 - \cos\left(h_s, h_t\right)$ to align the directions of the hidden state vectors between the teacher and student:

$$L_{\text{DistilBERT}} = \alpha L_{mlm} + \beta L_{KD} + \gamma L_{cos}$$

Since we already have a fine-tuned BERT-base model, let's see how we can use knowledge distillation to fine-tune a smaller and faster model. To do that we'll need a way to augment the cross-entropy loss with an L_{KD} term. Fortunately we can do this by creating our own trainer!

Creating a Knowledge Distillation Trainer

To implement knowledge distillation we need to add a few things to the `Trainer` base class:

- The new hyperparameters α and T, which control the relative weight of the distillation loss and how much the probability distribution of the labels should be smoothed
- The fine-tuned teacher model, which in our case is BERT-base
- A new loss function that combines the cross-entropy loss with the knowledge distillation loss

Adding the new hyperparameters is quite simple, since we just need to subclass `TrainingArguments` and include them as new attributes:

```
from transformers import TrainingArguments

class DistillationTrainingArguments(TrainingArguments):
    def __init__(self, *args, alpha=0.5, temperature=2.0, **kwargs):
        super().__init__(*args, **kwargs)
        self.alpha = alpha
        self.temperature = temperature
```

8 V. Sanh et al., "DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter", (2019).

For the trainer itself, we need a new loss function. The way to implement this is by subclassing `Trainer` and overriding the `compute_loss()` method to include the knowledge distillation loss term L_{KD}:

```
import torch.nn as nn
import torch.nn.functional as F
from transformers import Trainer

class DistillationTrainer(Trainer):
    def __init__(self, *args, teacher_model=None, **kwargs):
        super().__init__(*args, **kwargs)
        self.teacher_model = teacher_model

    def compute_loss(self, model, inputs, return_outputs=False):
        outputs_stu = model(**inputs)
        # Extract cross-entropy loss and logits from student
        loss_ce = outputs_stu.loss
        logits_stu = outputs_stu.logits
        # Extract logits from teacher
        with torch.no_grad():
            outputs_tea = self.teacher_model(**inputs)
            logits_tea = outputs_tea.logits
        # Soften probabilities and compute distillation loss
        loss_fct = nn.KLDivLoss(reduction="batchmean")
        loss_kd = self.args.temperature ** 2 * loss_fct(
            F.log_softmax(logits_stu / self.args.temperature, dim=-1),
            F.softmax(logits_tea / self.args.temperature, dim=-1))
        # Return weighted student loss
        loss = self.args.alpha * loss_ce + (1. - self.args.alpha) * loss_kd
        return (loss, outputs_stu) if return_outputs else loss
```

Let's unpack this code a bit. When we instantiate `DistillationTrainer` we pass a `teacher_model` argument with a teacher that has already been fine-tuned on our task. Next, in the `compute_loss()` method we extract the logits from the student and teacher, scale them by the temperature, and then normalize them with a softmax before passing them to PyTorch's `nn.KLDivLoss()` function for computing the KL divergence. One quirk with `nn.KLDivLoss()` is that it expects the inputs in the form of log probabilities and the labels as normal probabilities. That's why we've used the `F.log_softmax()` function to normalize the student's logits, while the teacher's logits are converted to probabilities with a standard softmax. The `reduction=batchmean` argument in `nn.KLDivLoss()` specifies that we average the losses over the batch dimension.

You can also perform knowledge distillation with the Keras API of the 🤗 Transformers library. To do this, you'll need to implement a custom `Distiller` class that overrides the `train_step()`, `test_step()`, and `compile()` methods of `tf.keras.Model()`. See the Keras documentation for an example of how to do this.

Choosing a Good Student Initialization

Now that we have our custom trainer, the first question you might have is which pretrained language model should we pick for the student? In general we should pick a smaller model for the student to reduce the latency and memory footprint. A good rule of thumb from the literature is that knowledge distillation works best when the teacher and student are of the same *model type*.[9] One possible reason for this is that different model types, say BERT and RoBERTa, can have different output embedding spaces, which hinders the ability of the student to mimic the teacher. In our case study the teacher is BERT, so DistilBERT is a natural candidate to initialize the student with since it has 40% fewer parameters and has been shown to achieve strong results on downstream tasks.

First we'll need to tokenize and encode our queries, so let's instantiate the tokenizer from DistilBERT and create a simple `tokenize_text()` function to take care of the preprocessing:

```
from transformers import AutoTokenizer

student_ckpt = "distilbert-base-uncased"
student_tokenizer = AutoTokenizer.from_pretrained(student_ckpt)

def tokenize_text(batch):
    return student_tokenizer(batch["text"], truncation=True)

clinc_enc = clinc.map(tokenize_text, batched=True, remove_columns=["text"])
clinc_enc = clinc_enc.rename_column("intent", "labels")
```

Here we've removed the `text` column since we no longer need it, and we've also renamed the `intent` column to `labels` so it can be automatically detected by the trainer.[10]

Now that we've processed our texts, the next thing we need to do is define the hyperparameters and `compute_metrics()` function for our `DistillationTrainer`. We'll also push all of our models to the Hugging Face Hub, so let's start by logging in to our account:

```
from huggingface_hub import notebook_login

notebook_login()
```

9 Y. Kim and H. Awadalla, "FastFormers: Highly Efficient Transformer Models for Natural Language Understanding", (2020).

10 By default, the `Trainer` looks for a column called `labels` when fine-tuning on classification tasks. You can also override this behavior by specifying the `label_names` argument of `TrainingArguments`.

Next, we'll define the metrics to track during training. As we did in the performance benchmark, we'll use accuracy as the main metric. This means we can reuse our `accuracy_score()` function in the `compute_metrics()` function that we'll include in `DistillationTrainer`:

```
def compute_metrics(pred):
    predictions, labels = pred
    predictions = np.argmax(predictions, axis=1)
    return accuracy_score.compute(predictions=predictions, references=labels)
```

In this function, the predictions from the sequence modeling head come in the form of logits, so we use the `np.argmax()` function to find the most confident class prediction and compare that against the ground truth label.

Next we need to define the training arguments. To warm up, we'll set $\alpha = 1$ to see how well DistilBERT performs without any signal from the teacher.[11] Then we will push our fine-tuned model to a new repository called `distilbert-base-uncased-finetuned-clinc`, so we just need to specify that in the `output_dir` argument of `DistillationTrainingArguments`:

```
batch_size = 48

finetuned_ckpt = "distilbert-base-uncased-finetuned-clinc"
student_training_args = DistillationTrainingArguments(
    output_dir=finetuned_ckpt, evaluation_strategy = "epoch",
    num_train_epochs=5, learning_rate=2e-5,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size, alpha=1, weight_decay=0.01,
    push_to_hub=True)
```

We've also tweaked a few of the default hyperparameter values, like the number of epochs, the weight decay, and the learning rate. The next thing to do is initialize a student model. Since we will be doing multiple runs with the trainer, we'll create a `student_init()` function to initialize the model with each new run. When we pass this function to the `DistillationTrainer`, this will ensure we initialize a new model each time we call the `train()` method.

One other thing we need to do is provide the student model with the mappings between each intent and label ID. These mappings can be obtained from our BERT-base model that we downloaded in the pipeline:

```
id2label = pipe.model.config.id2label
label2id = pipe.model.config.label2id
```

11 This approach of fine-tuning a general-purpose, distilled language model is sometimes referred to as "task-agnostic" distillation.

With these mappings, we can now create a custom model configuration with the `AutoConfig` class hat we encountered in Chapters 3 and 4. Let's use this to create a configuration for our student with the information about the label mappings:

```
from transformers import AutoConfig

num_labels = intents.num_classes
student_config = (AutoConfig
                  .from_pretrained(student_ckpt, num_labels=num_labels,
                                   id2label=id2label, label2id=label2id))
```

Here we've also specified the number of classes our model should expect. We can then provide this configuration to the `from_pretrained()` function of the `AutoModelForSequenceClassification` class as follows:

```
import torch
from transformers import AutoModelForSequenceClassification

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def student_init():
    return (AutoModelForSequenceClassification
            .from_pretrained(student_ckpt, config=student_config).to(device))
```

We now have all the ingredients needed for our distillation trainer, so let's load the teacher and fine-tune:

```
teacher_ckpt = "transformersbook/bert-base-uncased-finetuned-clinc"
teacher_model = (AutoModelForSequenceClassification
                 .from_pretrained(teacher_ckpt, num_labels=num_labels)
                 .to(device))

distilbert_trainer = DistillationTrainer(model_init=student_init,
    teacher_model=teacher_model, args=student_training_args,
    train_dataset=clinc_enc['train'], eval_dataset=clinc_enc['validation'],
    compute_metrics=compute_metrics, tokenizer=student_tokenizer)

distilbert_trainer.train()
```

Epoch	Training Loss	Validation Loss	Accuracy
1	4.2923	3.289337	0.742258
2	2.6307	1.883680	0.828065
3	1.5483	1.158315	0.896774
4	1.0153	0.861815	0.909355
5	0.7958	0.777289	0.917419

The 92% accuracy on the validation set looks quite good compared to the 94% that the BERT-base teacher achieves. Now that we've fine-tuned DistilBERT, let's push the model to the Hub so we can reuse it later:

```
distilbert_trainer.push_to_hub("Training completed!")
```

With our model now safely stored on the Hub, we can immediately use it in a pipeline for our performance benchmark:

```
finetuned_ckpt = "transformersbook/distilbert-base-uncased-finetuned-clinc"
pipe = pipeline("text-classification", model=finetuned_ckpt)
```

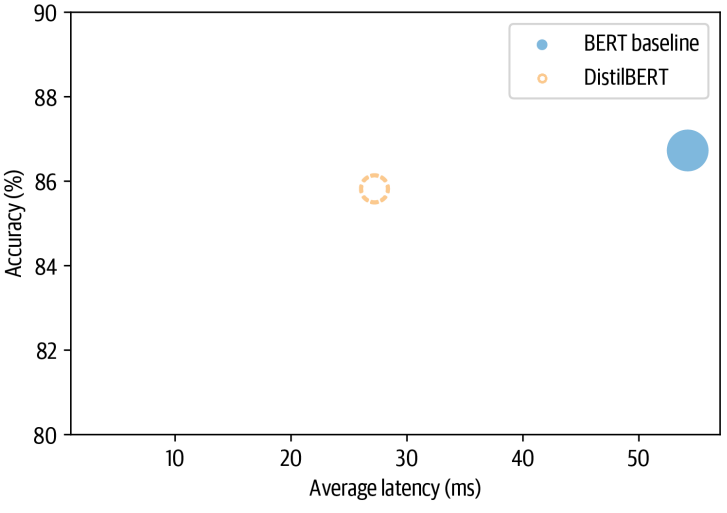
We can then pass this pipeline to our `PerformanceBenchmark` class to compute the metrics associated with this model:

```
optim_type = "DistilBERT"
pb = PerformanceBenchmark(pipe, clinc["test"], optim_type=optim_type)
perf_metrics.update(pb.run_benchmark())
```

```
Model size (MB) - 255.89
Average latency (ms) - 27.53 +\- 0.60
Accuracy on test set - 0.858
```

To compare these results against our baseline, let's create a scatter plot of the accuracy against the latency, with the radius of each point corresponding to the size of the model on disk. The following function does what we need and marks the current optimization type as a dashed circle to aid the comparison to previous results:

```
import pandas as pd

def plot_metrics(perf_metrics, current_optim_type):
    df = pd.DataFrame.from_dict(perf_metrics, orient='index')

    for idx in df.index:
        df_opt = df.loc[idx]
        # Add a dashed circle around the current optimization type
        if idx == current_optim_type:
            plt.scatter(df_opt["time_avg_ms"], df_opt["accuracy"] * 100,
                        alpha=0.5, s=df_opt["size_mb"], label=idx,
                        marker='$\u25CC$')
        else:
            plt.scatter(df_opt["time_avg_ms"], df_opt["accuracy"] * 100,
                        s=df_opt["size_mb"], label=idx, alpha=0.5)

    legend = plt.legend(bbox_to_anchor=(1,1))
    for handle in legend.legendHandles:
        handle.set_sizes([20])

    plt.ylim(80,90)
    # Use the slowest model to define the x-axis range
    xlim = int(perf_metrics["BERT baseline"]["time_avg_ms"] + 3)
    plt.xlim(1, xlim)
    plt.ylabel("Accuracy (%)")
    plt.xlabel("Average latency (ms)")
    plt.show()

plot_metrics(perf_metrics, optim_type)
```



From the plot we can see that by using a smaller model we've managed to significantly decrease the average latency. And all this at the price of just over a 1% reduction in accuracy! Let's see if we can close that last gap by including the distillation loss of the teacher and finding good values for α and T.

Finding Good Hyperparameters with Optuna

To find good values for α and T, we could do a grid search over the 2D parameter space. But a much better alternative is to use *Optuna*,[12] which is an optimization framework designed for just this type of task. Optuna formulates the search problem in terms of an objective function that is optimized through multiple *trials*. For example, suppose we wished to minimize Rosenbrock's "banana function":

$$f(x, y) = (1 - x)^2 + 100\left(y - x^2\right)^2$$

which is a famous test case for optimization frameworks. As shown in Figure 8-5, the function gets its name from the curved contours and has a global minimum at $(x, y) = (1, 1)$. Finding the valley is an easy optimization problem, but converging to the global minimum is not.

12 T. Akiba et al., "Optuna: A Next-Generation Hyperparameter Optimization Framework", (2019).

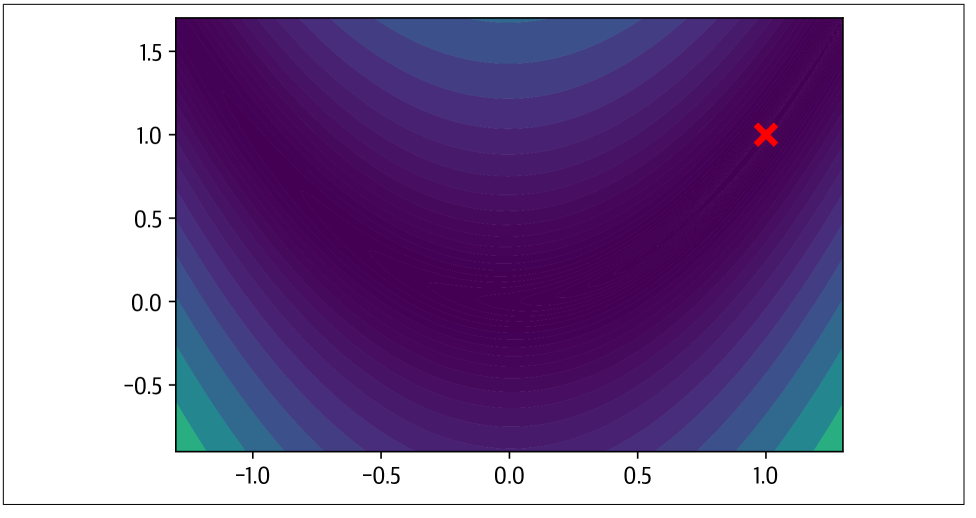


Figure 8-5. Plot of the Rosenbrock function of two variables

In Optuna, we can find the minimum of $f(x, y)$ by defining an `objective()` function that returns the value of $f(x, y)$:

```
def objective(trial):
    x = trial.suggest_float("x", -2, 2)
    y = trial.suggest_float("y", -2, 2)
    return (1 - x) ** 2 + 100 * (y - x ** 2) ** 2
```

The `trial.suggest_float` object specifies the parameter ranges to sample uniformly from; Optuna also provides `suggest_int` and `suggest_categorical` for integer and categorical parameters, respectively. Optuna collects multiple trials as a *study*, so to create one we just pass the `objective()` function to `study.optimize()` as follows:

```
import optuna

study = optuna.create_study()
study.optimize(objective, n_trials=1000)
```

Once the study is completed, we can then find the best parameters as follows:

```
study.best_params
```

```
{'x': 1.003024865971437, 'y': 1.00315167589307}
```

We see that with one thousand trials, Optuna has managed to find values for x and y that are reasonably close to the global minimum. To use Optuna in 🤗 Transformers, we use similar logic by first defining the hyperparameter space that we wish to optimize over. In addition to α and T, we'll include the number of training epochs as follows:

```
def hp_space(trial):
    return {"num_train_epochs": trial.suggest_int("num_train_epochs", 5, 10),
        "alpha": trial.suggest_float("alpha", 0, 1),
        "temperature": trial.suggest_int("temperature", 2, 20)}
```

Running the hyperparameter search with the `Trainer` is then quite simple; we just need to specify the number of trials to run and a direction to optimize for. Because we want the best possible accuracy, we specify `direction="maximize"` in the `hyperparameter_search()` method of the trainer and pass the hyperparameter search space as follows:

```
best_run = distilbert_trainer.hyperparameter_search(
    n_trials=20, direction="maximize", hp_space=hp_space)
```

The `hyperparameter_search()` method returns a `BestRun` object, which contains the value of the objective that was maximized (by default, the sum of all metrics) and the hyperparameters it used for that run:

```
print(best_run)
```

```
BestRun(run_id='1', objective=0.927741935483871,
hyperparameters={'num_train_epochs': 10, 'alpha': 0.12468168730193585,
'temperature': 7})
```

This value of α tells us that most of the training signal is coming from the knowledge distillation term. Let's update our training arguments with these values and run the final training run:

```
for k,v in best_run.hyperparameters.items():
    setattr(student_training_args, k, v)

# Define a new repository to store our distilled model
distilled_ckpt = "distilbert-base-uncased-distilled-clinc"
student_training_args.output_dir = distilled_ckpt

# Create a new Trainer with optimal parameters
distil_trainer = DistillationTrainer(model_init=student_init,
    teacher_model=teacher_model, args=student_training_args,
    train_dataset=clinc_enc['train'], eval_dataset=clinc_enc['validation'],
    compute_metrics=compute_metrics, tokenizer=student_tokenizer)

distil_trainer.train();
```

Epoch	Training Loss	Validation Loss	Accuracy
1	0.9031	0.574540	0.736452
2	0.4481	0.285621	0.874839
3	0.2528	0.179766	0.918710
4	0.1760	0.139828	0.929355
5	0.1416	0.121053	0.934839
6	0.1243	0.111640	0.934839
7	0.1133	0.106174	0.937742
8	0.1075	0.103526	0.938710
9	0.1039	0.101432	0.938065
10	0.1018	0.100493	0.939355

Remarkably, we've been able to train the student to match the accuracy of the teacher, despite it having almost half the number of parameters! Let's push the model to the Hub for future use:

```
distil_trainer.push_to_hub("Training complete")
```

Benchmarking Our Distilled Model

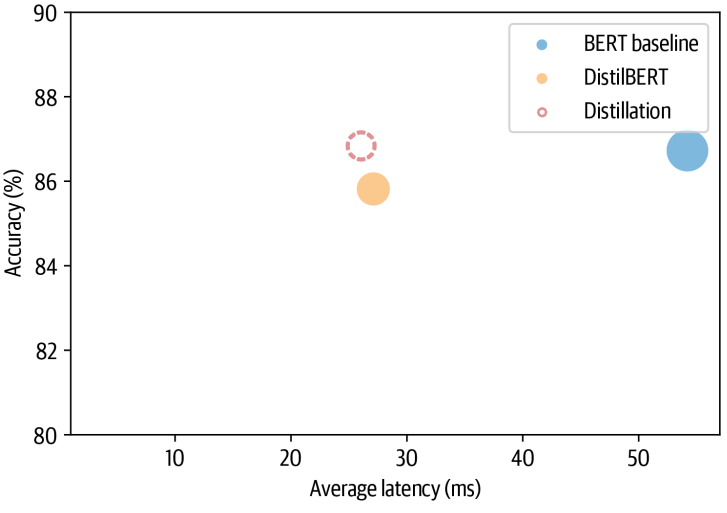
Now that we have an accurate student, let's create a pipeline and redo our benchmark to see how we perform on the test set:

```
distilled_ckpt = "transformersbook/distilbert-base-uncased-distilled-clinc"
pipe = pipeline("text-classification", model=distilled_ckpt)
optim_type = "Distillation"
pb = PerformanceBenchmark(pipe, clinc["test"], optim_type=optim_type)
perf_metrics.update(pb.run_benchmark())
```

```
Model size (MB) - 255.89
Average latency (ms) - 25.96 +\- 1.63
Accuracy on test set - 0.868
```

To put these results in context, let's also visualize them with our `plot_metrics()` function:

```
plot_metrics(perf_metrics, optim_type)
```



As expected, the model size and latency remain essentially unchanged compared to the DistilBERT benchmark, but the accuracy has improved and even surpassed the performance of the teacher! One way to interpret this surprising result is that the teacher has likely not been fine-tuned as systematically as the student. This is great, but we can actually compress our distilled model even further using a technique known as quantization. That's the topic of the next section.

Making Models Faster with Quantization

We've now seen that with knowledge distillation we can reduce the computational and memory cost of running inference by transferring the information from a teacher into a smaller student. Quantization takes a different approach; instead of reducing the number of computations, it makes them much more efficient by representing the weights and activations with low-precision data types like 8-bit integer (INT8) instead of the usual 32-bit floating point (FP32). Reducing the number of bits means the resulting model requires less memory storage, and operations like matrix multiplication can be performed much faster with integer arithmetic. Remarkably, these performance gains can be realized with little to no loss in accuracy!

A Primer on Floating-Point and Fixed-Point Numbers

Most transformers today are pretrained and fine-tuned with floating-point numbers (usually FP32 or a mix of FP16 and FP32), since they provide the precision needed to accommodate the very different ranges of weights, activations, and gradients. A floating-point number like FP32 represents a sequence of 32 bits that are grouped in terms of a *sign*, *exponent*, and *significand*. The sign determines whether the number is positive or negative, while the significand corresponds to the number of significant digits, which are scaled using the exponent in some fixed base (usually 2 for binary or 10 for decimal).

For example, the number 137.035 can be expressed as a decimal floating-point number through the following arithmetic:

$$137.035 = (-1)^0 \times 1.37035 \times 10^2$$

where the 1.37035 is the significand and 2 is the exponent of the base 10. Through the exponent we can represent a wide range of real numbers, and the decimal or binary point can be placed anywhere relative to the significant digits (hence the name "floating-point").

However, once a model is trained, we only need the forward pass to run inference, so we can reduce the precision of the data types without impacting the accuracy too much. For neural networks it is common to use a *fixed-point format* for the low-precision data types, where real numbers are represented as B-bit integers that are scaled by a common factor for all variables of the same type. For example, 137.035 can be represented as the integer 137,035 that is scaled by 1/1,000. We can control the range and precision of a fixed-point number by adjusting the scaling factor.

The basic idea behind quantization is that we can "discretize" the floating-point values f in each tensor by mapping their range $[f_{\text{max}}, f_{\text{min}}]$ into a smaller one $[q_{\text{max}}, q_{\text{min}}]$ of fixed-point numbers q, and linearly distributing all values in between. Mathematically, this mapping is described by the following equation:

$$f = \left(\frac{f_{\text{max}} - f_{\text{min}}}{q_{\text{max}} - q_{\text{min}}}\right)(q - Z) = S(q - Z)$$

where the scale factor S is a positive floating-point number and the constant Z has the same type as q and is called the *zero point* because it corresponds to the quantized value of the floating-point value $f = 0$. Note that the map needs to be *affine* so that we

get back floating-point numbers when we dequantize the fixed-point ones.[13] An illustration of the conversion is shown in Figure 8-6.

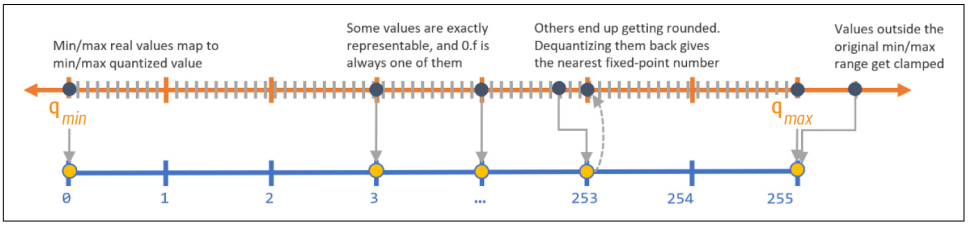
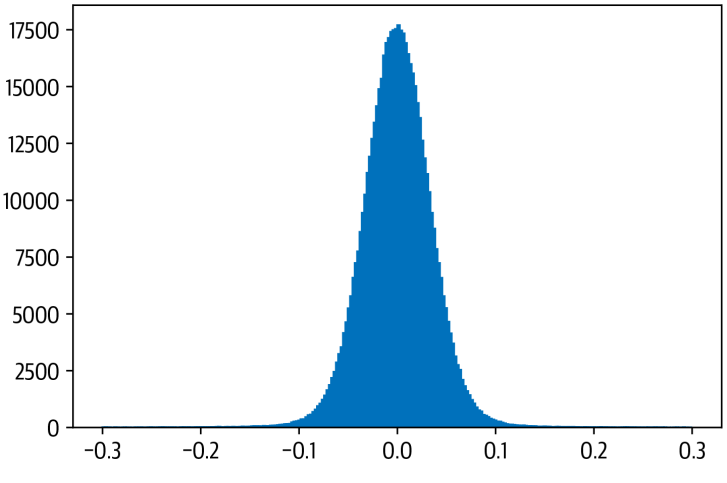


Figure 8-6. Quantizing floating-point numbers as unsigned 8-bit integers (courtesy of Manas Sahni)

Now, one of the main reasons why transformers (and deep neural networks more generally) are prime candidates for quantization is that the weights and activations tend to take values in relatively small ranges. This means we don't have to squeeze the whole range of possible FP32 numbers into, say, the $2^8 = 256$ numbers represented by INT8. To see this, let's pick out one of the attention weight matrices from our distilled model and plot the frequency distribution of the values:

```
import matplotlib.pyplot as plt

state_dict = pipe.model.state_dict()
weights = state_dict["distilbert.transformer.layer.0.attention.out_lin.weight"]
plt.hist(weights.flatten().numpy(), bins=250, range=(-0.3,0.3), edgecolor="C0")
plt.show()
```



13 An affine map is just a fancy name for the $y = Ax + b$ map that you're familiar with in the linear layers of a neural network.

As we can see, the values of the weights are distributed in the small range $[-0.1, 0.1]$ around zero. Now, suppose we want to quantize this tensor as a signed 8-bit integer. In that case, the range of possible values for our integers is $[q_{\text{max}}, q_{\text{min}}] = [-128, 127]$. The zero point coincides with the zero of FP32 and the scale factor is calculated according to the previous equation:

```
zero_point = 0
scale = (weights.max() - weights.min()) / (127 - (-128))
```

To obtain the quantized tensor, we just need to invert the mapping $q = f/S + Z$, clamp the values, round them to the nearest integer, and represent the result in the `torch.int8` data type using the `Tensor.char()` function:

```
(weights / scale + zero_point).clamp(-128, 127).round().char()
```

```
tensor([[ -5,  -8,   0,  ...,  -6,  -4,   8],
        [  8,   3,   1,  ...,  -4,   7,   0],
        [ -9,  -6,   5,  ...,   1,   5,  -3],
        ...,
        [  6,   0,  12,  ...,   0,   6,  -1],
        [  0,  -2, -12,  ...,  12,  -7, -13],
        [-13,  -1, -10,  ...,   8,   2,  -2]], dtype=torch.int8)
```

Great, we've just quantized our first tensor! In PyTorch we can simplify the conversion by using the `quantize_per_tensor()` function together with a quantized data type, `torch.qint`, that is optimized for integer arithmetic operations:

```
from torch import quantize_per_tensor

dtype = torch.qint8
quantized_weights = quantize_per_tensor(weights, scale, zero_point, dtype)
quantized_weights.int_repr()
```

```
tensor([[ -5,  -8,   0,  ...,  -6,  -4,   8],
        [  8,   3,   1,  ...,  -4,   7,   0],
        [ -9,  -6,   5,  ...,   1,   5,  -3],
        ...,
        [  6,   0,  12,  ...,   0,   6,  -1],
        [  0,  -2, -12,  ...,  12,  -7, -13],
        [-13,  -1, -10,  ...,   8,   2,  -2]], dtype=torch.int8)
```

The plot in Figure 8-7 shows very clearly the discretization that's induced by only mapping some of the weight values precisely and rounding the rest.

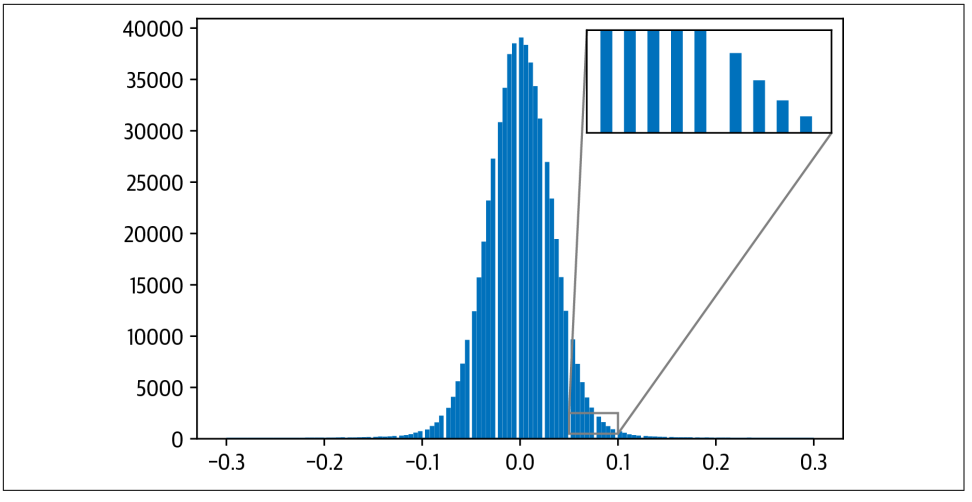


Figure 8-7. Effect of quantization on a transformer's weights

To round out our little analysis, let's compare how long it takes to compute the multiplication of two weight tensors with FP32 and INT8 values. For the FP32 tensors, we can multiply them using PyTorch's nifty `@` operator:

```
%%timeit
weights @ weights
```

```
393 µs ± 3.84 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

For the quantized tensors we need the `QFunctional` wrapper class so that we can perform operations with the special `torch.qint8` data type:

```
from torch.nn.quantized import QFunctional

q_fn = QFunctional()
```

This class supports various elementary operations, like addition, and in our case we can time the multiplication of our quantized tensors as follows:

```
%%timeit
q_fn.mul(quantized_weights, quantized_weights)
```

```
23.3 µs ± 298 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Compared to our FP32 computation, using the INT8 tensors is almost 100 times faster! Even larger gains can be obtained by using dedicated backends for running quantized operators efficiently. As of this book's writing, PyTorch supports:

- x86 CPUs with AVX2 support or higher
- ARM CPUs (typically found in mobile/embedded devices)

Since INT8 numbers have four times fewer bits than FP32 numbers, quantization also reduces the memory storage requirements by up to a factor of four. In our simple example we can verify this by comparing the underlying storage size of our weight tensor and its quantized cousin by using the `Tensor.storage()` function and the `getsizeof()` function from Python's `sys` module:

```
import sys

sys.getsizeof(weights.storage()) / sys.getsizeof(quantized_weights.storage())
```

```
3.999633833760527
```

For a full-scale transformer, the actual compression rate depends on which layers are quantized (as we'll see in the next section it is only the linear layers that typically get quantized).

So what's the catch with quantization? Changing the precision for all computations in our model introduces small disturbances at each point in the model's computational graph, which can compound and affect the model's performance. There are several ways to quantize a model, which all have pros and cons. For deep neural networks, there are typically three main approaches to quantization:

Dynamic quantization
: When using dynamic quantization nothing is changed during training and the adaptations are only performed during inference. Like with all the quantization methods we will discuss, the weights of the model are converted to INT8 ahead of inference time. In addition to the weights, the model's activations are also quantized. This approach is dynamic because the quantization happens on the fly. This means that all the matrix multiplications can be calculated with highly optimized INT8 functions. Of all the quantization methods discussed here, dynamic quantization is the simplest one. However, with dynamic quantization the activations are written and read to memory in floating-point format. This conversion between integer and floating point can be a performance bottleneck.

Static quantization
: Instead of computing the quantization of the activations on the fly, we can avoid the conversion to floating point by precomputing the quantization scheme. Static quantization achieves this by observing the activation patterns on a representative sample of the data ahead of inference time. The ideal quantization scheme is calculated and then saved. This enables us to skip the conversion between INT8 and FP32 values and speeds up the computations. However, it requires access to a good data sample and introduces an additional step in the pipeline, since we now need to train and determine the quantization scheme before we can perform inference. There is also one aspect that static quantization does not address: the discrepancy between the precision during training and inference, which leads to a performance drop in the model's metrics (e.g., accuracy).

Quantization-aware training
: The effect of quantization can be effectively simulated during training by "fake" quantization of the FP32 values. Instead of using INT8 values during training, the FP32 values are rounded to mimic the effect of quantization. This is done during both the forward and the backward pass and improves performance in terms of model metrics over static and dynamic quantization.

The main bottleneck for running inference with transformers is the compute and memory bandwidth associated with the enormous numbers of weights in these models. For this reason, dynamic quantization is currently the best approach for transformer-based models in NLP. In smaller computer vision models the limiting factor is the memory bandwidth of the activations, which is why static quantization is generally used (or quantization-aware training in cases where the performance drops are too significant).

Implementing dynamic quantization in PyTorch is quite simple and can be done with a single line of code:

```
from torch.quantization import quantize_dynamic

model_ckpt = "transformersbook/distilbert-base-uncased-distilled-clinc"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
model = (AutoModelForSequenceClassification
         .from_pretrained(model_ckpt).to("cpu"))

model_quantized = quantize_dynamic(model, {nn.Linear}, dtype=torch.qint8)
```

Here we pass to `quantize_dynamic()` the full-precision model and specify the set of PyTorch layer classes in that model that we want to quantize. The `dtype` argument specifies the target precision and can be `fp16` or `qint8`. A good practice is to pick the lowest precision that you can tolerate with respect to your evaluation metrics. In this chapter we'll use INT8, which as we'll soon see has little impact on our model's accuracy.

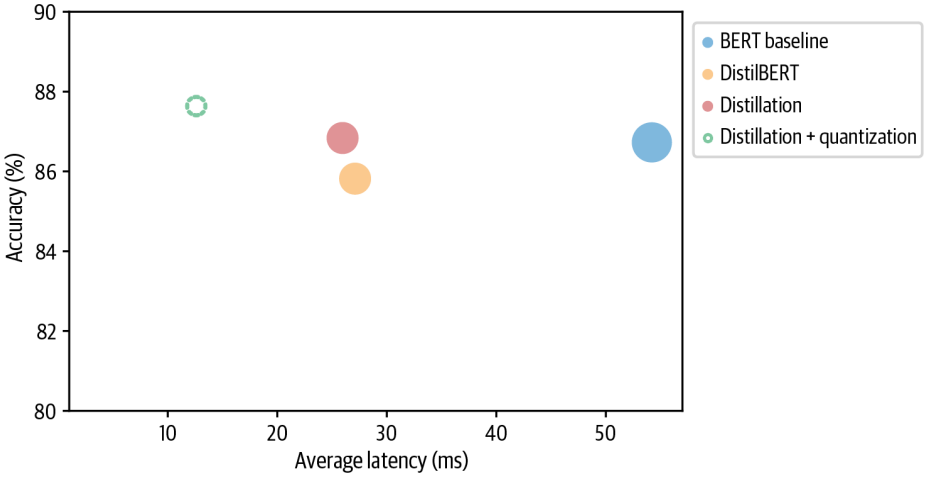
Benchmarking Our Quantized Model

With our model now quantized, let's pass it through the benchmark and visualize the results:

```
pipe = pipeline("text-classification", model=model_quantized,
                tokenizer=tokenizer)
optim_type = "Distillation + quantization"
pb = PerformanceBenchmark(pipe, clinc["test"], optim_type=optim_type)
perf_metrics.update(pb.run_benchmark())
```

```
Model size (MB) - 132.40
Average latency (ms) - 12.54 +\- 0.73
Accuracy on test set - 0.876
```

```
plot_metrics(perf_metrics, optim_type)
```



Nice, the quantized model is almost half the size of our distilled one and has even gained a slight accuracy boost! Let's see if we can push our optimization to the limit with a powerful framework called the ONNX Runtime.

Optimizing Inference with ONNX and the ONNX Runtime

ONNX is an open standard that defines a common set of operators and a common file format to represent deep learning models in a wide variety of frameworks, including PyTorch and TensorFlow.[14] When a model is exported to the ONNX format, these operators are used to construct a computational graph (often called an *intermediate representation*) that represents the flow of data through the neural network. An example of such a graph for BERT-base is shown in Figure 8-8, where each node receives some input, applies an operation like `Add` or `Squeeze`, and then feeds the output to the next set of nodes.

14 There is a separate standard called ONNX-ML that is designed for traditional machine learning models like random forests and frameworks like Scikit-learn.

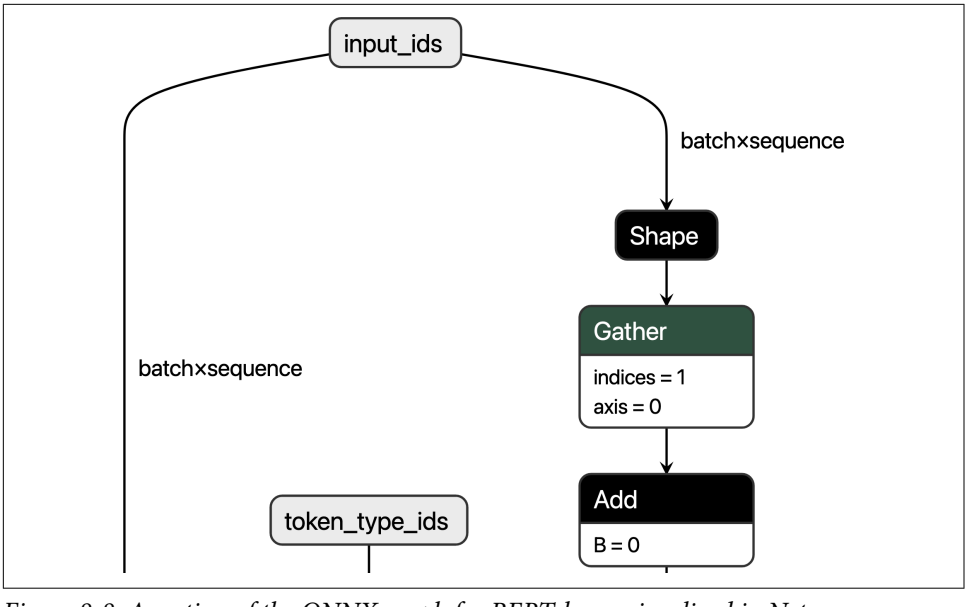


Figure 8-8. A section of the ONNX graph for BERT-base, visualized in Netron

By exposing a graph with standardized operators and data types, ONNX makes it easy to switch between frameworks. For example, a model trained in PyTorch can be exported to ONNX format and then imported in TensorFlow (and vice versa).

Where ONNX really shines is when it is coupled with a dedicated accelerator like ONNX Runtime, or ORT for short.[15] ORT provides tools to optimize the ONNX graph through techniques like operator fusion and constant folding,[16] and defines an interface to *execution providers* that allow you to run the model on different types of hardware. This is a powerful abstraction. Figure 8-9 shows the high-level architecture of the ONNX and ORT ecosystem.

15 Other popular accelerators include NVIDIA's TensorRT and Apache TVM.

16 A fused operation involves merging one operator (usually an activation function) into another so that they can be executed together. For example, suppose we want to apply an activation f to a matrix product $A \times B$. Normally the result of the product needs to be written back to the GPU memory before the activation is computed. Operator fusion allows as to compute $f(A \times B)$ in a single step. Constant folding refers to the process of evaluating constant expressions at compile time instead of runtime.

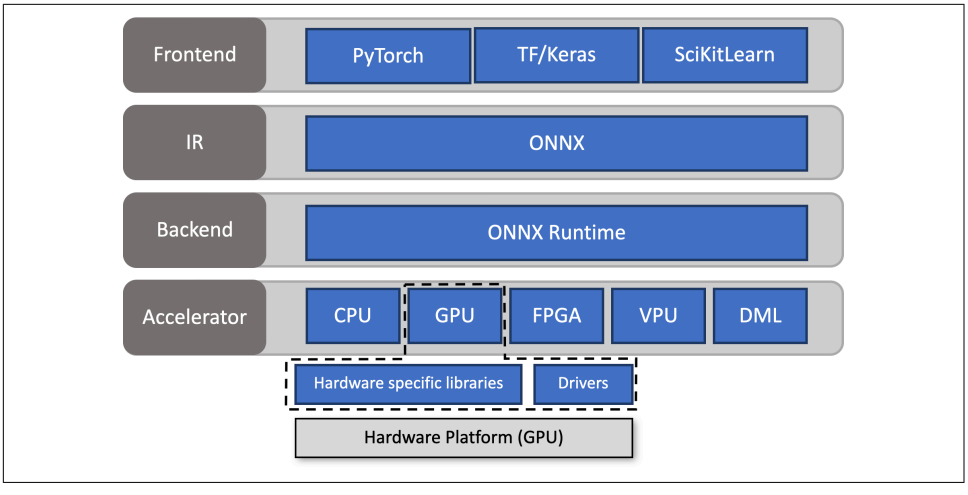


Figure 8-9. Architecture of the ONNX and ONNX Runtime ecosystem (courtesy of the ONNX Runtime team)

To see ORT in action, the first thing we need to do is convert our distilled model into the ONNX format. The 🤗 Transformers library has a built-in function called `convert_graph_to_onnx.convert()` that simplifies the process by taking the following steps:

1. Initialize the model as a `Pipeline`.
2. Run placeholder inputs through the pipeline so that ONNX can record the computational graph.
3. Define dynamic axes to handle dynamic sequence lengths.
4. Save the graph with network parameters.

To use this function, we first need to set some OpenMP environment variables for ONNX:

```
import os
from psutil import cpu_count

os.environ["OMP_NUM_THREADS"] = f"{cpu_count()}"
os.environ["OMP_WAIT_POLICY"] = "ACTIVE"
```

OpenMP is an API designed for developing highly parallelized applications. The `OMP_NUM_THREADS` environment variable sets the number of threads to use for parallel computations in the ONNX Runtime, while `OMP_WAIT_POLICY=ACTIVE` specifies that waiting threads should be active (i.e., using CPU processor cycles).

Next, let's convert our distilled model to the ONNX format. Here we need to specify the argument `pipeline_name="text-classification"` since `convert()` wraps the

model in a 🤗 Transformers `pipeline()` function during the conversion. In addition to the `model_ckpt`, we also pass the tokenizer to initialize the pipeline:

```
from transformers.convert_graph_to_onnx import convert

model_ckpt = "transformersbook/distilbert-base-uncased-distilled-clinc"
onnx_model_path = Path("onnx/model.onnx")
convert(framework="pt", model=model_ckpt, tokenizer=tokenizer,
        output=onnx_model_path, opset=12, pipeline_name="text-classification")
```

ONNX uses *operator sets* to group together immutable operator specifications, so `opset=12` corresponds to a specific version of the ONNX library.

Now that we have our model saved, we need to create an `InferenceSession` instance to feed inputs to the model:

```
from onnxruntime import (GraphOptimizationLevel, InferenceSession,
                         SessionOptions)

def create_model_for_provider(model_path, provider="CPUExecutionProvider"):
    options = SessionOptions()
    options.intra_op_num_threads = 1
    options.graph_optimization_level = GraphOptimizationLevel.ORT_ENABLE_ALL
    session = InferenceSession(str(model_path), options, providers=[provider])
    session.disable_fallback()
    return session

onnx_model = create_model_for_provider(onnx_model_path)
```

Now when we call `onnx_model.run()`, we can get the class logits from the ONNX model. Let's test this out with an example from the test set. Since the output from `convert()` tells us that ONNX expects just the `input_ids` and `attention_mask` as inputs, we need to drop the `label` column from our sample:

```
inputs = clinc_enc["test"][:1]
del inputs["labels"]
logits_onnx = onnx_model.run(None, inputs)[0]
logits_onnx.shape
```

```
(1, 151)
```

Once we have the logits, we can easily get the predicted label by taking the argmax:

```
np.argmax(logits_onnx)
```

```
61
```

which indeed agrees with the ground truth label:

```
clinc_enc["test"][0]["labels"]
```

```
61
```

The ONNX model is not compatible with the `text-classification` pipeline, so we'll create our own class that mimics the core behavior:

```
from scipy.special import softmax

class OnnxPipeline:
    def __init__(self, model, tokenizer):
        self.model = model
        self.tokenizer = tokenizer

    def __call__(self, query):
        model_inputs = self.tokenizer(query, return_tensors="pt")
        inputs_onnx = {k: v.cpu().detach().numpy()
                       for k, v in model_inputs.items()}
        logits = self.model.run(None, inputs_onnx)[0][0, :]
        probs = softmax(logits)
        pred_idx = np.argmax(probs).item()
        return [{"label": intents.int2str(pred_idx), "score": probs[pred_idx]}]
```

We can then test this on our simple query to see if we recover the `car_rental` intent:

```
pipe = OnnxPipeline(onnx_model, tokenizer)
pipe(query)
```

```
[{'label': 'car_rental', 'score': 0.7848334}]
```

Great, our pipeline works as expected. The next step is to create a performance benchmark for ONNX models. Here we can build on the work we did with the `PerformanceBenchmark` class by simply overriding the `compute_size()` method and leaving the `compute_accuracy()` and `time_pipeline()` methods intact. The reason we need to override the `compute_size()` method is that we cannot rely on the `state_dict` and `torch.save()` to measure a model's size, since `onnx_model` is technically an ONNX `InferenceSession` object that doesn't have access to the attributes of PyTorch's `nn.Module`. In any case, the resulting logic is simple and can be implemented as follows:

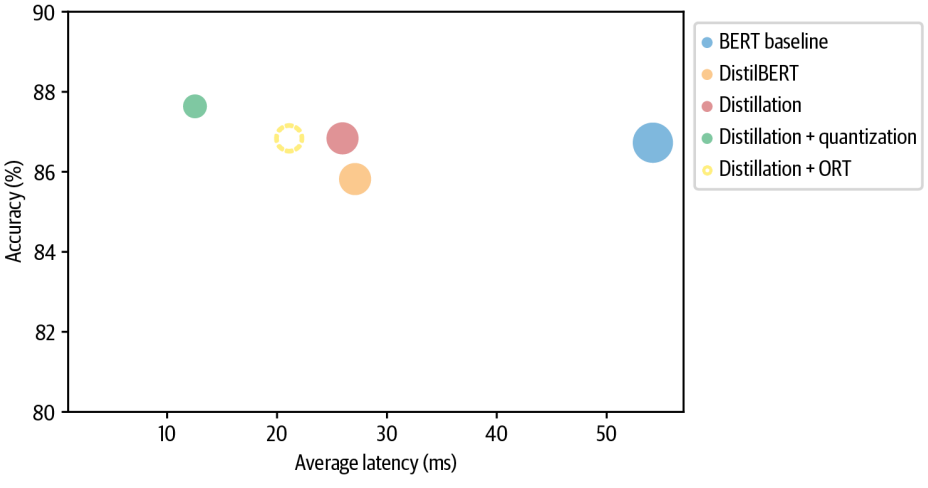
```
class OnnxPerformanceBenchmark(PerformanceBenchmark):
    def __init__(self, *args, model_path, **kwargs):
        super().__init__(*args, **kwargs)
        self.model_path = model_path

    def compute_size(self):
        size_mb = Path(self.model_path).stat().st_size / (1024 * 1024)
        print(f"Model size (MB) - {size_mb:.2f}")
        return {"size_mb": size_mb}
```

With our new benchmark, let's see how our distilled model performs when converted to ONNX format:

```
optim_type = "Distillation + ORT"
pb = OnnxPerformanceBenchmark(pipe, clinc["test"], optim_type,
                              model_path="onnx/model.onnx")
perf_metrics.update(pb.run_benchmark())
```

```
Model size (MB) - 255.88
Average latency (ms) - 21.02 +\- 0.55
Accuracy on test set - 0.868
```

```
plot_metrics(perf_metrics, optim_type)
```



Remarkably, converting to the ONNX format and using the ONNX Runtime has given our distilled model (i.e. the "Distillation" circle in the plot) a boost in latency! Let's see if we can squeeze out a bit more performance by adding quantization to the mix.

Similar to PyTorch, ORT offers three ways to quantize a model: dynamic, static, and quantization-aware training. As we did with PyTorch, we'll apply dynamic quantization to our distilled model. In ORT, the quantization is applied through the `quantize_dynamic()` function, which requires a path to the ONNX model to quantize, a target path to save the quantized model to, and the data type to reduce the weights to:

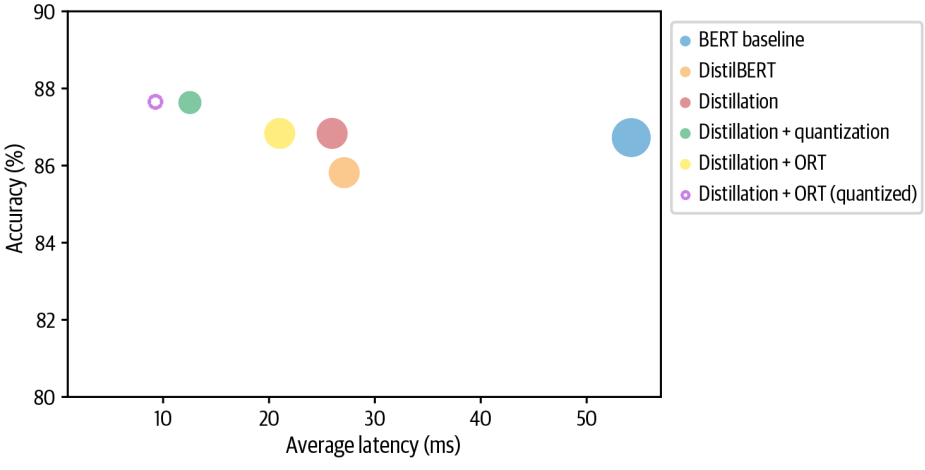
```
from onnxruntime.quantization import quantize_dynamic, QuantType

model_input = "onnx/model.onnx"
model_output = "onnx/model.quant.onnx"
quantize_dynamic(model_input, model_output, weight_type=QuantType.QInt8)
```

Now that the model is quantized, let's run it through our benchmark:

```
onnx_quantized_model = create_model_for_provider(model_output)
pipe = OnnxPipeline(onnx_quantized_model, tokenizer)
optim_type = "Distillation + ORT (quantized)"
pb = OnnxPerformanceBenchmark(pipe, clinc["test"], optim_type,
                              model_path=model_output)
perf_metrics.update(pb.run_benchmark())
```

```
Model size (MB) - 64.20
Average latency (ms) - 9.24 +\- 0.29
Accuracy on test set - 0.877
```

```
plot_metrics(perf_metrics, optim_type)
```



ORT quantization has reduced the model size and latency by around 30% compared to the model obtained from PyTorch quantization (the distillation + quantization blob). One reason for this is that PyTorch only optimizes the `nn.Linear` modules, while ONNX quantizes the embedding layer as well. From the plot we can also see that applying ORT quantization to our distilled model has provided an almost three-fold gain compared to our BERT baseline!

This concludes our analysis of techniques to speed up transformers for inference. We have seen that methods such as quantization reduce the model size by reducing the precision of the representation. Another strategy to reduce the size is to remove some weights altogether. This technique is called *weight pruning*, and it's the focus of the next section.

Making Models Sparser with Weight Pruning

So far we've seen that knowledge distillation and weight quantization are quite effective at producing faster models for inference, but in some cases you might also have strong constraints on the memory footprint of your model. For example, if our product manager suddenly decides that our text assistant needs to be deployed on a mobile device, then we'll need our intent classifier to take up as little storage space as possible. To round out our survey of compression methods, let's take a look at how we can shrink the number of parameters in our model by identifying and removing the least important weights in the network.

Sparsity in Deep Neural Networks

As shown in Figure 8-10, the main idea behind pruning is to gradually remove weight connections (and potentially neurons) during training such that the model becomes progressively sparser. The resulting pruned model has a smaller number of nonzero parameters, which can then be stored in a compact sparse matrix format. Pruning can be also combined with quantization to obtain further compression.

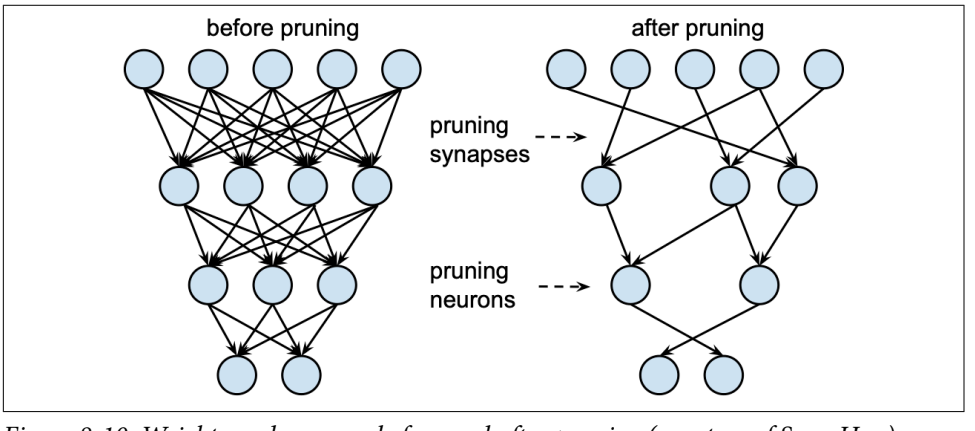


Figure 8-10. Weights and neurons before and after pruning (courtesy of Song Han)

Weight Pruning Methods

Mathematically, the way most weight pruning methods work is to calculate a matrix $\mathbf{S}$ of *importance scores* and then select the top k percent of weights by importance:

$$\text{Top}_k(\mathbf{S})_{ij} = \begin{cases} 1 & \text{if } S_{ij} \text{ in top } k\,\% \\ 0 & \text{otherwise} \end{cases}$$

In effect, k acts as a new hyperparameter to control the amount of sparsity in the model—that is, the proportion of weights that are zero-valued. Lower values of k correspond to sparser matrices. From these scores we can then define a *mask matrix* $\mathbf{M}$ that masks the weights W_{ij} during the forward pass with some input x_i and effectively creates a sparse network of activations a_i:

$$a_i = \sum_k W_{ik} M_{ik} x_k$$

As discussed in the tongue-in-cheek "Optimal Brain Surgeon" paper,[17] at the heart of each pruning method are a set of questions that need to be considered:

- Which weights should be eliminated?
- How should the remaining weights be adjusted for best performance?
- How can such network pruning be done in a computationally efficient way?

The answers to these questions inform how the score matrix $\mathbf{S}$ is computed, so let's begin by looking at one of the earliest and most popular pruning methods: magnitude pruning.

Magnitude pruning

As the name suggests, magnitude pruning calculates the scores according to the magnitude of the weights $\mathbf{S} = \left(\mid W_{ij} \mid \right)_{1 \le j, j \le n}$ and then derives the masks from $\mathbf{M} = \text{Top}_k(\mathbf{S})$. In the literature it is common to apply magnitude pruning in an iterative fashion by first training the model to learn which connections are important and pruning the weights of least importance.[18] The sparse model is then retrained and the process repeated until the desired sparsity is reached.

One drawback with this approach is that it is computationally demanding: at every step of pruning we need to train the model to convergence. For this reason it is generally better to gradually increase the initial sparsity s_i (which is usually zero) to a final value s_f after some number of steps N:[19]

$$s_t = s_f + \left(s_i - s_f\right)\left(1 - \frac{t - t_0}{N\Delta t}\right)^3 \qquad \text{for } t \in \left\{t_0, t_0 + \Delta t, \ldots, t_0 + N\Delta t\right\}$$

Here the idea is to update the binary masks $\mathbf{M}$ every Δt steps to allow masked weights to reactivate during training and recover from any potential loss in accuracy that is induced by the pruning process. As shown in Figure 8-11, the cubic factor implies that the rate of weight pruning is highest in the early phases (when the number of redundant weights is large) and gradually tapers off.

17 B. Hassibi and D. Stork, "Second Order Derivatives for Network Pruning: Optimal Brain Surgeon," *Proceedings of the 5th International Conference on Neural Information Processing Systems* (November 1992): 164–171, *https://papers.nips.cc/paper/1992/hash/303ed4c69846ab36c2904d3ba8573050-Abstract.html*.

18 S. Han et al., "Learning Both Weights and Connections for Efficient Neural Networks", (2015).

19 M. Zhu and S. Gupta, "To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression", (2017).

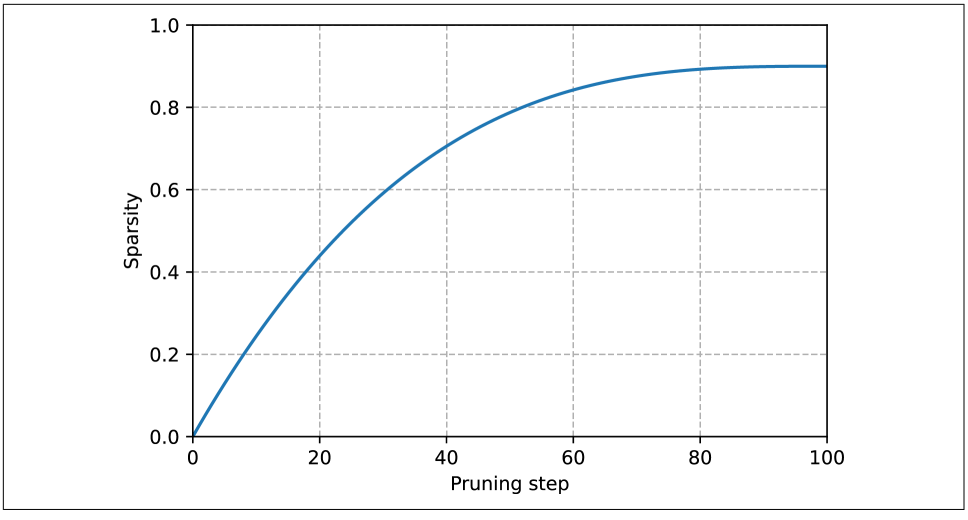


Figure 8-11. The cubic sparsity scheduler used for pruning

One problem with magnitude pruning is that it is really designed for pure supervised learning, where the importance of each weight is directly related to the task at hand. By contrast, in transfer learning the importance of the weights is primarily determined by the pretraining phase, so magnitude pruning can remove connections that are important for the fine-tuning task. Recently, an adaptive approach called movement pruning has been proposed by Hugging Face researchers—let's take a look.[20]

Movement pruning

The basic idea behind movement pruning is to *gradually* remove weights during fine-tuning such that the model becomes progressively *sparser*. The key novelty is that both the weights and the scores are learned during fine-tuning. So, instead of being derived directly from the weights (like with magnitude pruning), the scores in movement pruning are arbitrary and are learned through gradient descent like any other neural network parameter. This implies that in the backward pass, we also track the gradient of the loss L with respect to the scores S_{ij}.

Once the scores are learned, it is then straightforward to generate the binary mask using $\mathbf{M} = \text{Top}_k(\mathbf{S})$.[21]

The intuition behind movement pruning is that the weights that are "moving" the most from zero are the most important ones to keep. In other words, the positive

20 V. Sanh, T. Wolf, and A.M. Rush, "Movement Pruning: Adaptive Sparsity by Fine-Tuning", (2020).

21 There is also a "soft" version of movement pruning where instead of picking the top k% of weights, one uses a global threshold τ to define the binary mask: $\mathbf{M} = (\mathbf{S} > \tau)$.

weights increase during fine-tuning (and vice versa for the negative weights), which is equivalent to saying that the scores increase as the weights move away from zero. As shown in Figure 8-12, this behavior differs from magnitude pruning, which selects as the most important weights those that are *furthest* from zero.

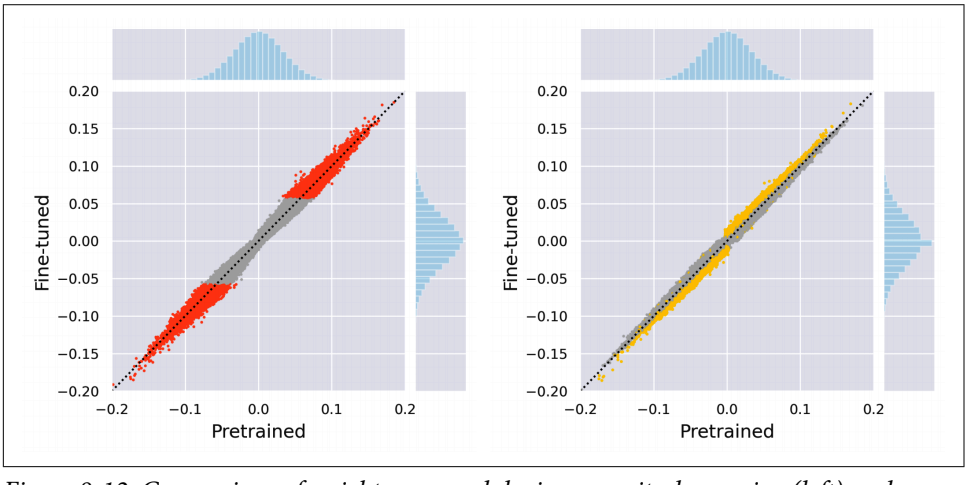


Figure 8-12. Comparison of weights removed during magnitude pruning (left) and movement pruning (right)

These differences between the two pruning methods are also evident in the distribution of the remaining weights. As shown in Figure 8-13, magnitude pruning produces two clusters of weights, while movement pruning produces a smoother distribution.

As of this book's writing, 🤗 Transformers does not support pruning methods out of the box. Fortunately, there is a nifty library called *Neural Networks Block Movement Pruning* that implements many of these ideas, and we recommend checking it out if memory constraints are a concern.

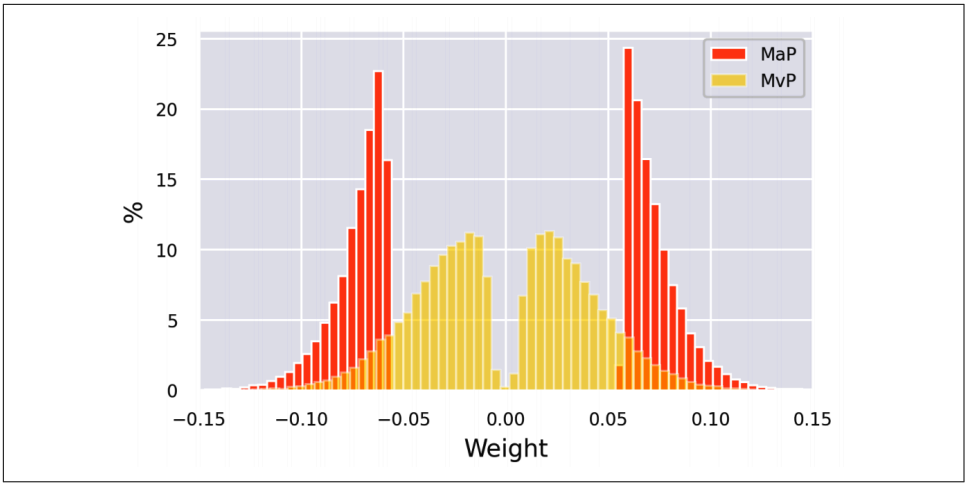


Figure 8-13. Distribution of remaining weights for magnitude pruning (MaP) and movement pruning (MvP)

Conclusion

We've seen that optimizing transformers for deployment in production environments involves compression along two dimensions: latency and memory footprint. Starting from a fine-tuned model, we applied distillation, quantization, and optimizations through ORT to significantly reduce both of these. In particular, we found that quantization and conversion in ORT gave the largest gains with minimal effort.

Although pruning is an effective strategy for reducing the storage size of transformer models, current hardware is not optimized for sparse matrix operations, which limits the usefulness of this technique. However, this is an active area of research, and by the time this book hits the shelves many of these limitations may have been resolved.

So where to from here? All of the techniques in this chapter can be adapted to other tasks, such as question answering, named entity recognition, or language modeling. If you find yourself struggling to meet the latency requirements or your model is eating up all your compute budget, we suggest giving one of them a try.

In the next chapter, we'll switch gears away from performance optimization and explore every data scientist's worst nightmare: dealing with few to no labels.