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.

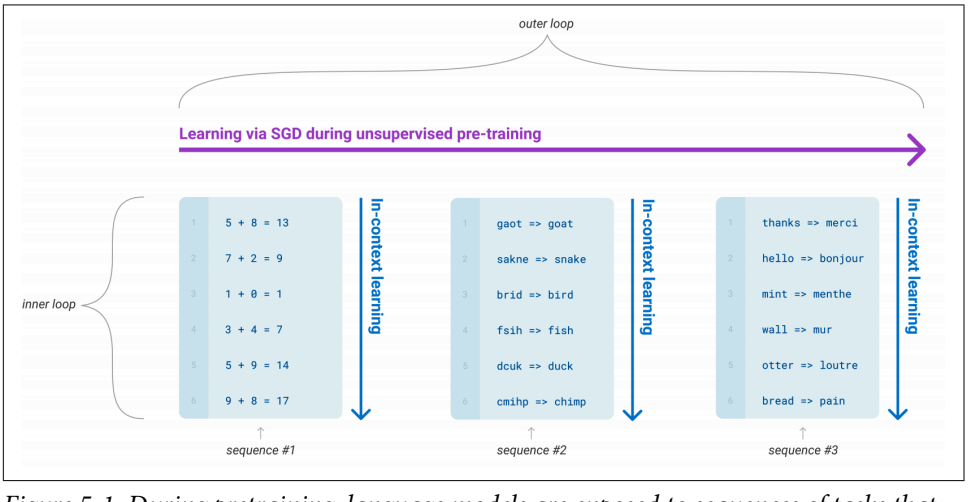


Figure 5-1. During pretraining, language models are exposed to sequences of tasks that can be adapted during inference (courtesy of Tom B. Brown)

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.

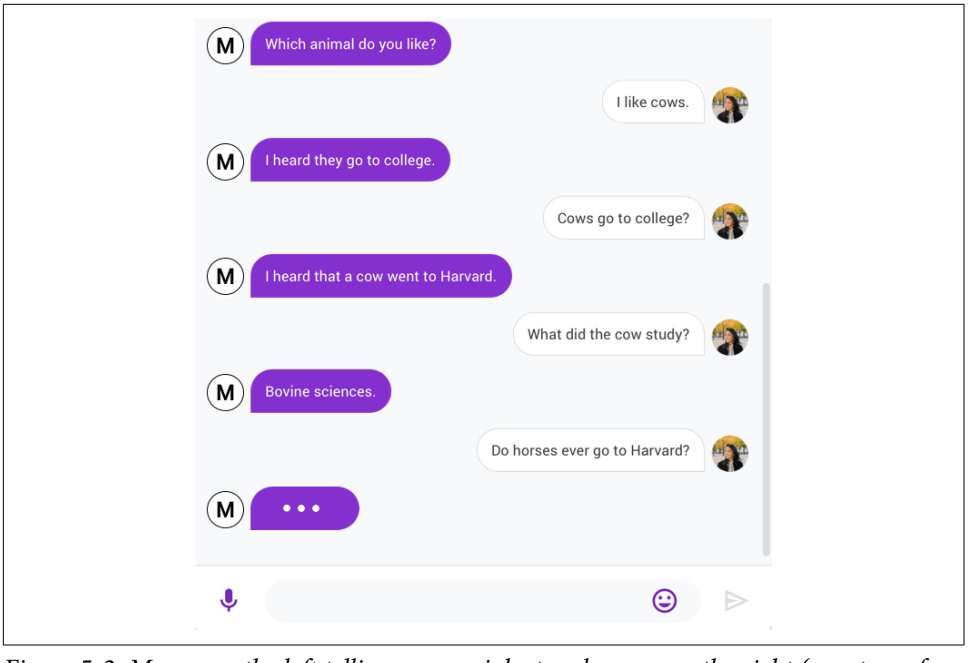


Figure 5-2. Meena on the left telling a corny joke to a human on the right (courtesy of Daniel Adiwardana and Thang Luong)

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.

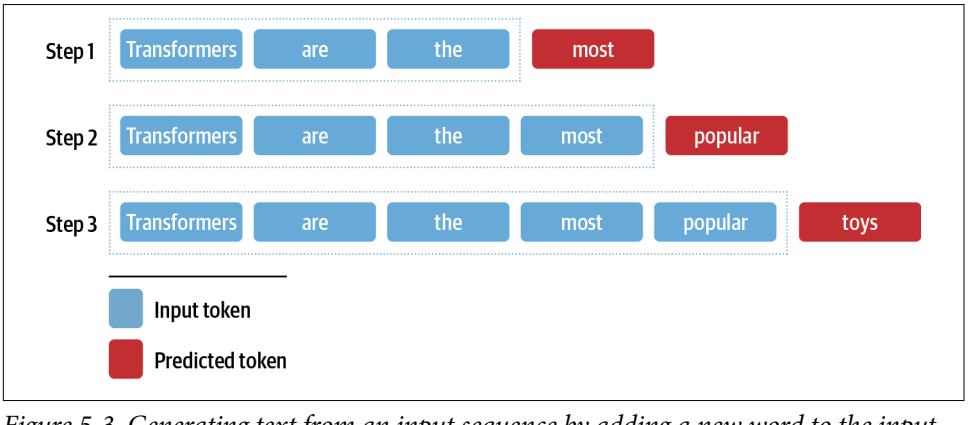


Figure 5-3. Generating text from an input sequence by adding a new word to the input at each step

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.

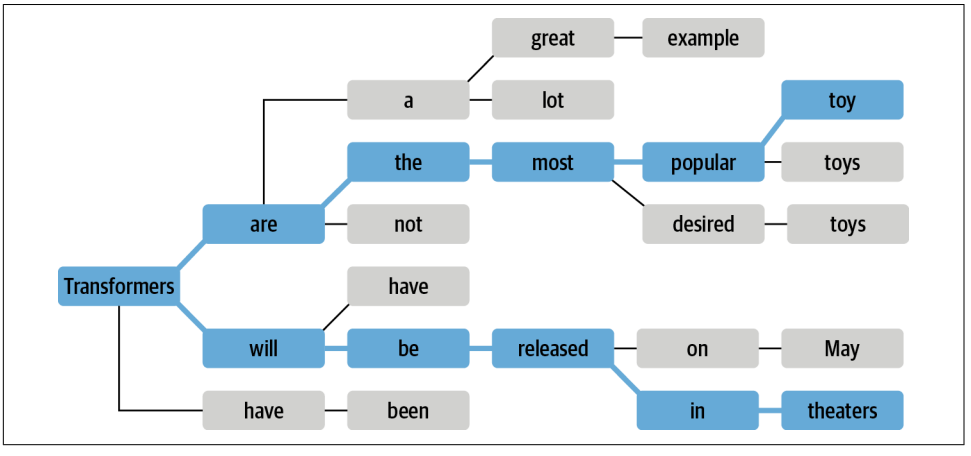


Figure 5-4. Beam search with two beams

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.

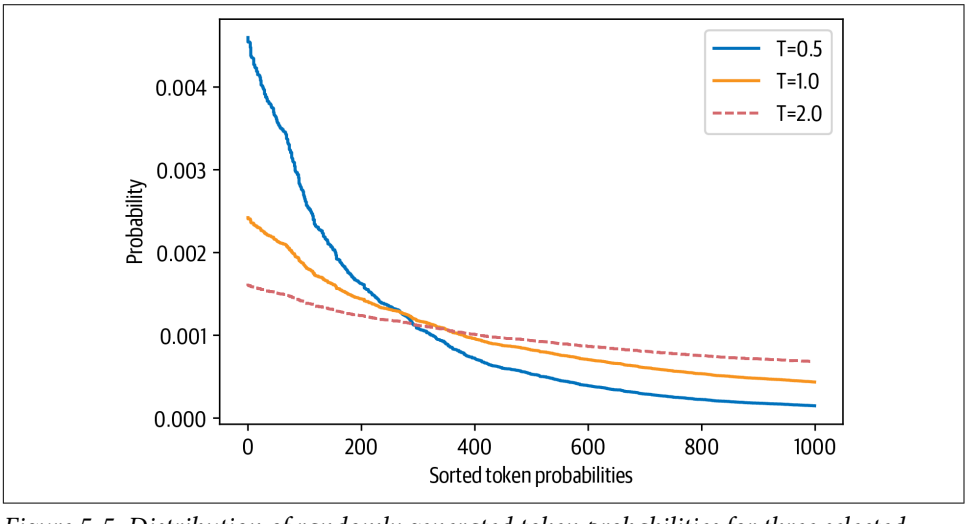


Figure 5-5. Distribution of randomly generated token probabilities for three selected temperatures

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.

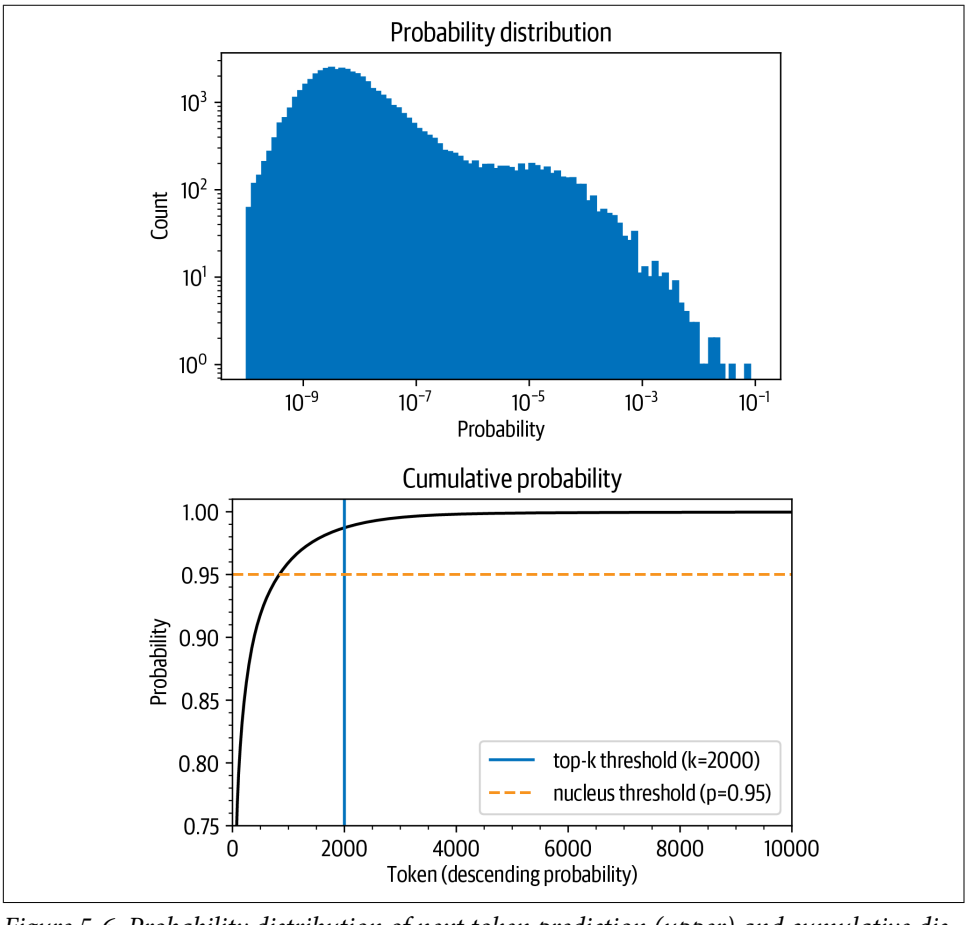


Figure 5-6. Probability distribution of next token prediction (upper) and cumulative distribution of descending token probabilities (lower)

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).

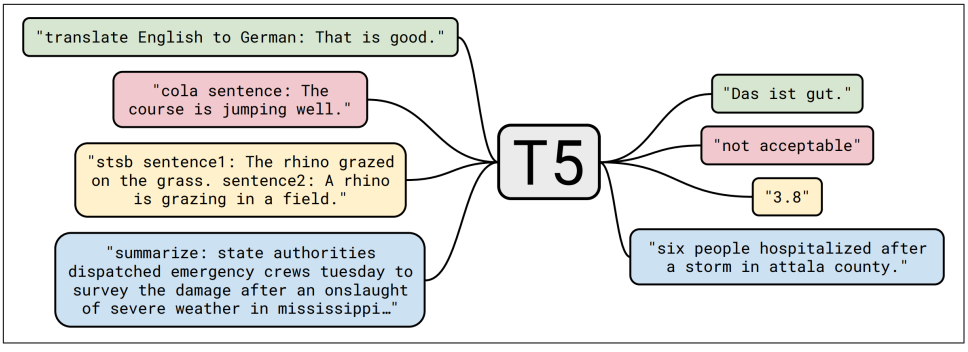


Figure 6-1. Diagram of T5's text-to-text framework (courtesy of Colin Raffel); besides translation and summarization, the CoLA (linguistic acceptability) and STSB (semantic similarity) tasks are shown

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).

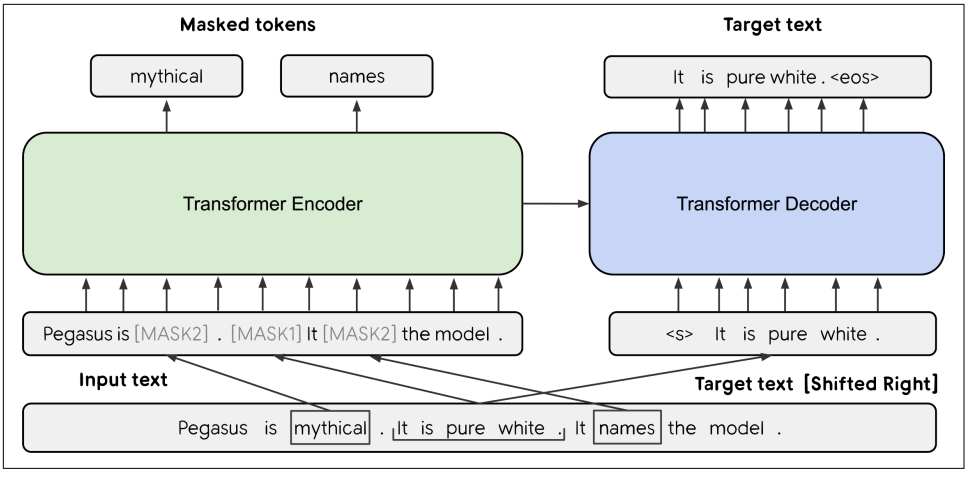


Figure 6-2. Diagram of PEGASUS architecture (courtesy of Jingqing Zhang et al.)

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.

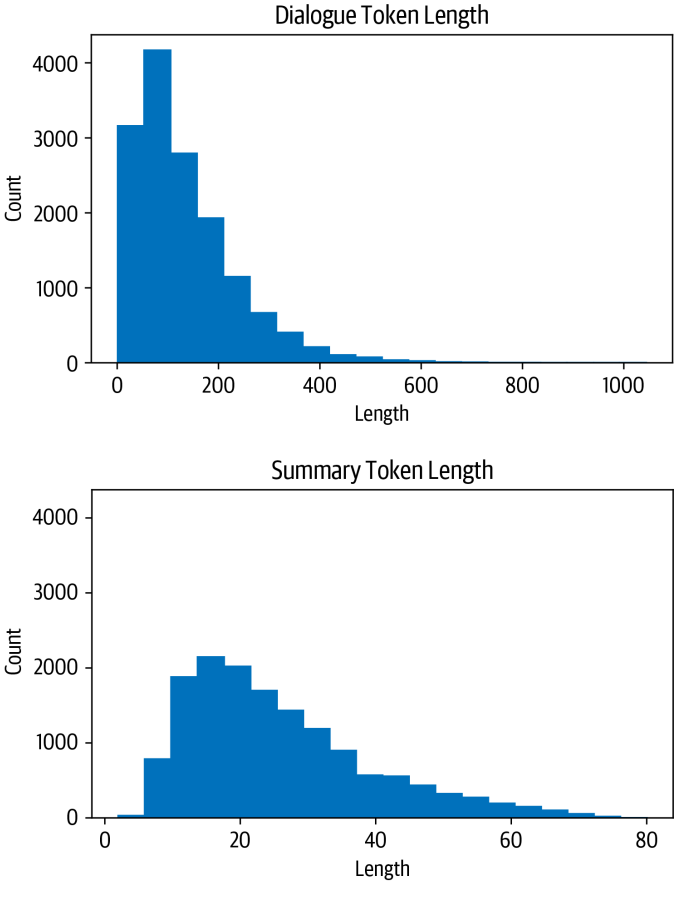
Fine-Tuning PEGASUS

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).