Chapter 14. Natural Language Processing with RNNs and Attention

When Alan Turing imagined his famous Turing test[1] in 1950, he proposed a way to evaluate a machine's ability to match human intelligence. He could have tested for many things, such as the ability to recognize cats in pictures, play chess, compose music, or escape a maze, but, interestingly, he chose a linguistic task. More specifically, he devised a *chatbot* capable of fooling its interlocutor into thinking it was human.[2] This test does have its weaknesses: a set of hard-coded rules can fool unsuspecting or naive humans (e.g., the machine could give vague predefined answers in response to some keywords, it could pretend that it is joking or drunk to get a pass on its weirdest answers, or it could escape difficult questions by answering them with its own questions), and many aspects of human intelligence are utterly ignored (e.g., the ability to interpret nonverbal communication such as facial expressions, or to learn a manual task). But the test does highlight the fact that mastering language is arguably *Homo sapiens*'s greatest cognitive ability.

Until recently, state-of-the-art natural language processing (NLP) models were pretty much all based on recurrent neural networks (introduced in Chapter 13). However, in recent years, RNNs have been replaced with transformers, which we will explore in Chapter 15. That said, it's still important to learn how RNNs can be used for NLP tasks, if only because it helps better understand transformers. Moreover, most of the techniques we will discuss in this chapter are also useful with Transformer architectures (e.g., tokenization, beam search, attention mechanisms, and more). Plus, RNNs have recently made a surprise comeback in the form of state space models (SSMs) (see Chapter 17).

This chapter is organized in three sections. In the first section, we will start by building a *character RNN*, or *char-RNN*, trained to predict the next character in a sentence. On the way, we will learn about trainable embeddings. Our char-RNN will be our first tiny *language model*, capable of generating original text.

In the second section, we will turn to text classification, and more specifically sentiment analysis, which aims to predict how positive or negative some text is. Our model will read movie reviews and estimate the rater's feeling about the movie. This time, instead of splitting the text into individual characters, we will split it into *tokens*: a token is a small piece of text from a fixed-sized vocabulary, such as the top 10,000 most common words in the English language, or the most common subwords (e.g., "smartest" = "smart" + "est"), or even individual characters or bytes. To split the text into tokens, we will use a *tokenizer*. This section will also introduce popular Hugging Face libraries: the *Datasets* library to download datasets, the *Tokenizers library* for tokenizers, and the *Transformers* library for popular models, downloaded automatically from the *Hugging Face Hub*. Hugging Face is a hugely influential company and open source community, and it plays a central role in the open source AI space, especially in NLP.

The final boss of this chapter will be neural machine translation (NMT), the topic of the third and last section: we will build an encoder–decoder model capable of translating English to Spanish. This will lead us to *attention mechanisms*, which we will apply to our encoder-decoder model to improve its capacity to handle long input texts. As their name suggests, attention mechanisms are neural network components that learn to select the part of the inputs that the model should focus on at each time step. They directly led to the transformers revolution, as we will see in the next chapter.

Let's start with a simple and fun char-RNN model that can write like Shakespeare (sort of).

Generating Shakespearean Text Using a Character RNN

In a famous 2015 blog post titled "The Unreasonable Effectiveness of Recurrent Neural Networks", Andrej Karpathy showed how to train an RNN to predict the next character in a sentence. This *char-RNN* can then be used to generate novel text, one character at a time. Here is a small sample of the text generated by a char-RNN model after it was trained on all of Shakespeare's works:

> *PANDARUS: Alas, I think he shall be come approached and the day When little srain would be attain'd into being never fed, And who is but a chain and subjects of his death, I should not sleep.*

Not exactly a masterpiece, but it is still impressive that the model was able to learn words, grammar, proper punctuation, and more, just by learning to predict the next character in a sentence. This is our first example of a *language model*. In the remainder of this section we'll build a char-RNN step by step, starting with the creation of the dataset.

Creating the Training Dataset

First, let's download a subset of Shakespeare's works (about 25%). The data is loaded from Andrej Karpathy's char-rnn project:

```
from pathlib import Path
import urllib.request

def download_shakespeare_text():
    path = Path("datasets/shakespeare/shakespeare.txt")
    if not path.is_file():
        path.parent.mkdir(parents=True, exist_ok=True)
        url = "https://homl.info/shakespeare"
        urllib.request.urlretrieve(url, path)
    return path.read_text()

shakespeare_text = download_shakespeare_text()
```

Let's print the first few lines:

```
>>> print(shakespeare_text[:80])
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.
```

Looks like Shakespeare, all right!

Neural networks work with numbers, not text, so we need a way to encode text into numbers. In general, this is done by splitting the text into *tokens*, such as words or characters, and assigning an integer ID to each possible token. For example, let's split our text into characters, and assign an ID to each possible character. We first need to find the list of characters used in the text. This will constitute our token *vocabulary*:

```
>>> vocab = sorted(set(shakespeare_text.lower()))
>>> "".join(vocab)
"\n !$&',-.3:;?abcdefghijklmnopqrstuvwxyz"
```

Note that we call `lower()` to ignore case and thereby reduce the vocabulary size. We must now assign a token ID to each character. For this, we can just use its index in the vocabulary. To decode the output of our model, we will also need a way to go from a token ID to a character:

```
>>> char_to_id = {char: index for index, char in enumerate(vocab)}
>>> id_to_char = {index: char for index, char in enumerate(vocab)}
>>> char_to_id["a"]
13
>>> id_to_char[13]
'a'
```

Next, let's create two helper functions to encode text to tensors of token IDs, and to decode them back to text:

```
import torch

def encode_text(text):
    return torch.tensor([char_to_id[char] for char in text.lower()])

def decode_text(char_ids):
    return "".join([id_to_char[char_id.item()] for char_id in char_ids])
```

Let's try them out:

```
>>> encoded = encode_text("Hello, world!")
>>> encoded
tensor([20, 17, 24, 24, 27,  6,  1, 35, 27, 30, 24, 16,  2])
>>> decode_text(encoded)
'hello, world!'
```

Next, let's prepare the dataset. Right now, we have a single, extremely long sequence of characters containing Shakespeare's works. Just like we did in Chapter 13, we can turn this long sequence into a dataset of windows that we can then use to train a sequence-to-sequence RNN. The targets will be similar to the inputs, but shifted by one time step into the "future". For example, one sample in the dataset may be a sequence of character IDs representing the text "to be or not to b" (without the final "e"), and the corresponding target—a sequence of character IDs representing the text "o be or not to be" (with the final "e", but without the leading "t"). Let's create our dataset class:

```
from torch.utils.data import Dataset, DataLoader

class CharDataset(Dataset):
    def __init__(self, text, window_length):
        self.encoded_text = encode_text(text)
        self.window_length = window_length

    def __len__(self):
        return len(self.encoded_text) - self.window_length

    def __getitem__(self, idx):
        if idx >= len(self):
            raise IndexError("dataset index out of range")
        end = idx + self.window_length
        window = self.encoded_text[idx : end]
        target = self.encoded_text[idx + 1 : end + 1]
        return window, target
```

And now let's create the data loaders, as usual. Since the text is quite large, we can afford to use roughly 90% for training (i.e., one million characters), and just 5% for validation, and 5% for testing (60,000 characters each):

```
window_length = 50
batch_size = 512  # reduce if your GPU cannot handle such a large batch size
train_set = CharDataset(shakespeare_text[:1_000_000], window_length)
valid_set = CharDataset(shakespeare_text[1_000_000:1_060_000], window_length)
test_set = CharDataset(shakespeare_text[1_060_000:], window_length)
train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
valid_loader = DataLoader(valid_set, batch_size=batch_size)
test_loader = DataLoader(test_set, batch_size=batch_size)
```

Each batch will be composed of 512 50-character windows, where each character is represented by its token ID, and where each window comes with its 50-character target window (offset by one character). Note that the training batches are shuffled at each epoch (see Figure 14-1).

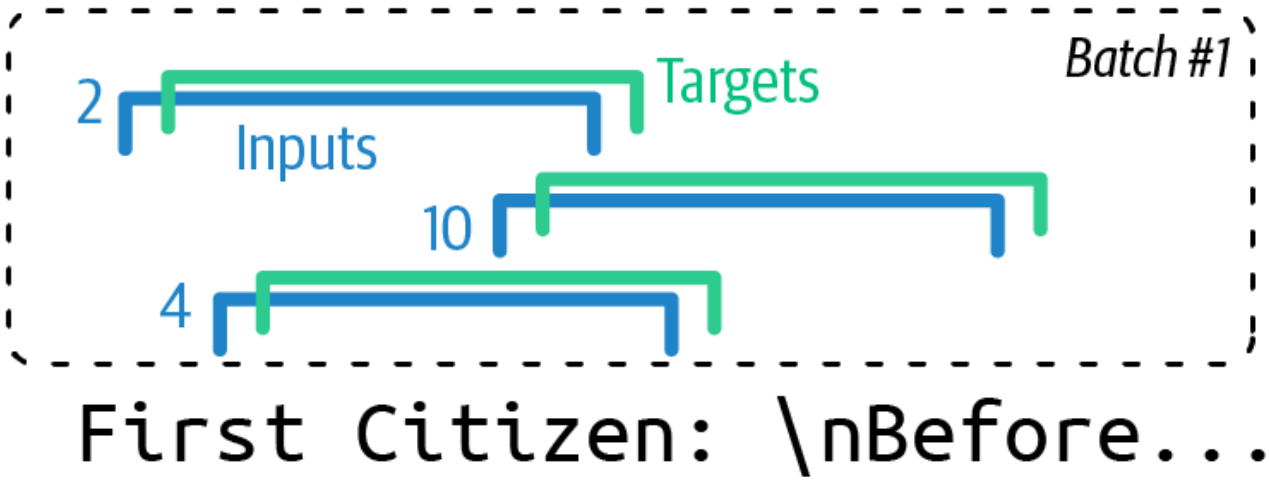


Figure 14-1. Each training batch is composed of shuffled windows, along with their shifted targets. In this figure, the window length is 10 instead of 50.

TIP

We set the window length to 50, but you can try tuning it. It's easier and faster to train RNNs on shorter input sequences, but the RNN will not be able to learn any pattern longer than the window length, so don't make it too small.

While we could technically feed the token IDs directly to a neural network without any further preprocessing, it wouldn't work very well. Indeed, as we saw in Chapter 2, most ML models—including neural networks—assume that similar inputs represent similar things; unfortunately, similar IDs may represent totally unrelated tokens, and conversely, distant IDs may represent similar tokens. The neural net would be biased in a weird way, and it would have great difficulty overcoming this bias during training.

One solution is to use one-hot encoding, since all one-hot vectors are equally distant from one another. However, when the vocabulary is large, one-hot vectors are equally large. In our case, the vocabulary contains just 39 characters, so each character would be represented by a 39-dimensional one-hot vector. That's still manageable, but if we were dealing with words instead of characters, the vocabulary size could be in the tens of thousands, so one-hot encoding would be out of the question. Luckily, since we are dealing with neural networks, we have a better option: embeddings.

Embeddings

An embedding is a dense representation of some higher-dimensional data, typically a categorical feature. If there are 50,000 possible categories, then one-hot encoding produces a 50,000-dimensional sparse vector (i.e., containing mostly zeros). In contrast, an embedding is a com-

paratively small dense vector; for example, with just 300 dimensions.

TIP

The embedding size is a hyperparameter you can tune. As a rule of thumb, a good embedding size is often close to the square root of the number of categories.

In deep learning, embeddings are usually initialized randomly, and they are then trained by gradient descent, along with the other model parameters. For example, if we wanted to train a neural network on the California housing dataset (see Chapter 2), we could represent the `ocean_proximity` categorical feature using embeddings. The `"NEAR BAY"` category could be represented initially by a random vector such as `[0.831, 0.696]`, while the `"NEAR OCEAN"` category might be represented by another random vector such as `[0.127, 0.868]` (in this example we are using 2D embeddings).

Since these embeddings are trainable, they will gradually improve during training; and as they represent fairly similar categories in this example, gradient descent will certainly end up pushing them closer together, while it will tend to move them away from the `"INLAND"` category's embedding (see Figure 14-2). Indeed, the better the representation, the easier it will be for the neural network to make accurate predictions, so training tends to make embeddings useful representations of the categories. This is called *representation learning* (you will see other types of representation learning in Chapter 18).

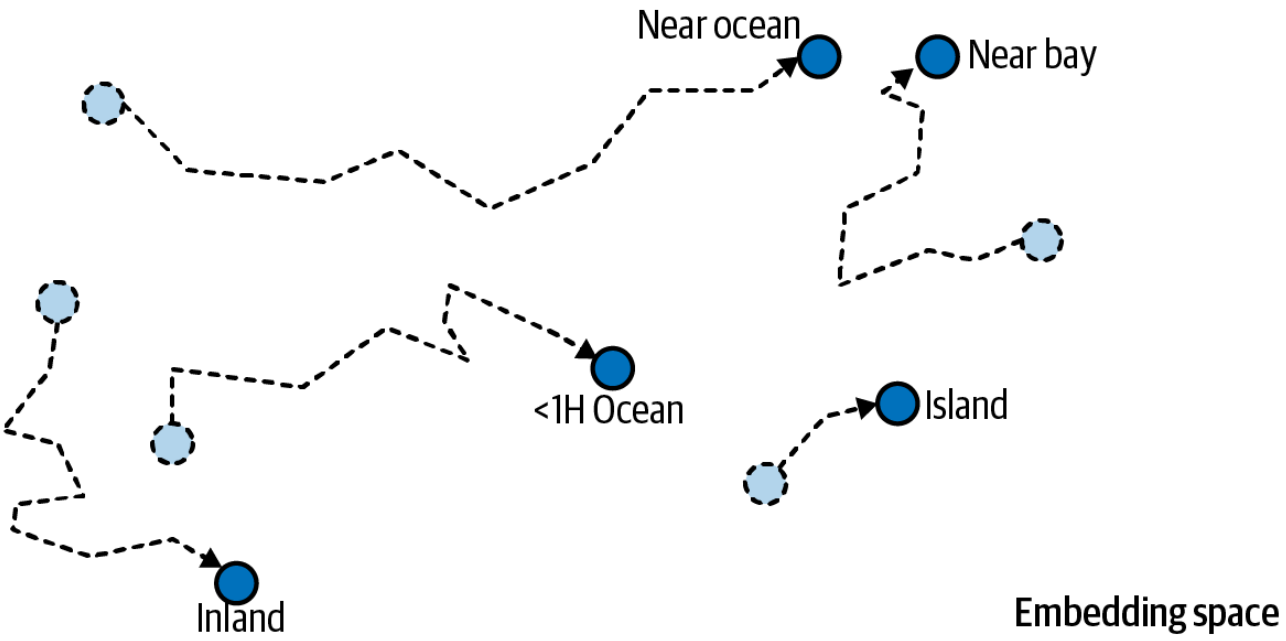


Figure 14-2. Embeddings will gradually improve during training

Not only will embeddings generally be useful representations for the task at hand, but quite often these same embeddings can be reused successfully for other tasks. The most common example of this is *word embeddings* (i.e., embeddings of individual words): when you are working on a natural language processing task, you are often better off reusing pretrained word embeddings than training your own, as we will see later in this chapter.

The idea of using vectors to represent words dates back to the 1960s, and many sophisticated techniques have been used to generate useful vectors, including using neural networks. But things really took off in 2013, when Tomáš Mikolov and other Google researchers published a paper[3] describing an efficient technique to learn word embeddings using neural networks, significantly outperforming previous attempts. This allowed them to learn embeddings on a very large corpus of text: they trained a neural network to predict the words near any given word and obtained astounding word embeddings. For example, synonyms had very close embeddings, and semantically related words such as *France*, *Spain*, and *Italy* were clustered together.

It's not just about proximity, though: word embeddings are also organized along meaningful axes in the embedding space. Here is a famous example: if you compute *King – Man + Woman* (adding and subtracting the embedding vectors of these words), then the result will be very close to the embedding of the word *Queen* (see Figure 14-3). In other words, the word embeddings encode the concept of gender! Similarly, you can compute *Madrid – Spain + France*, and the result is close to *Paris*, which seems to show that the notion of capital city is also encoded in the embeddings.

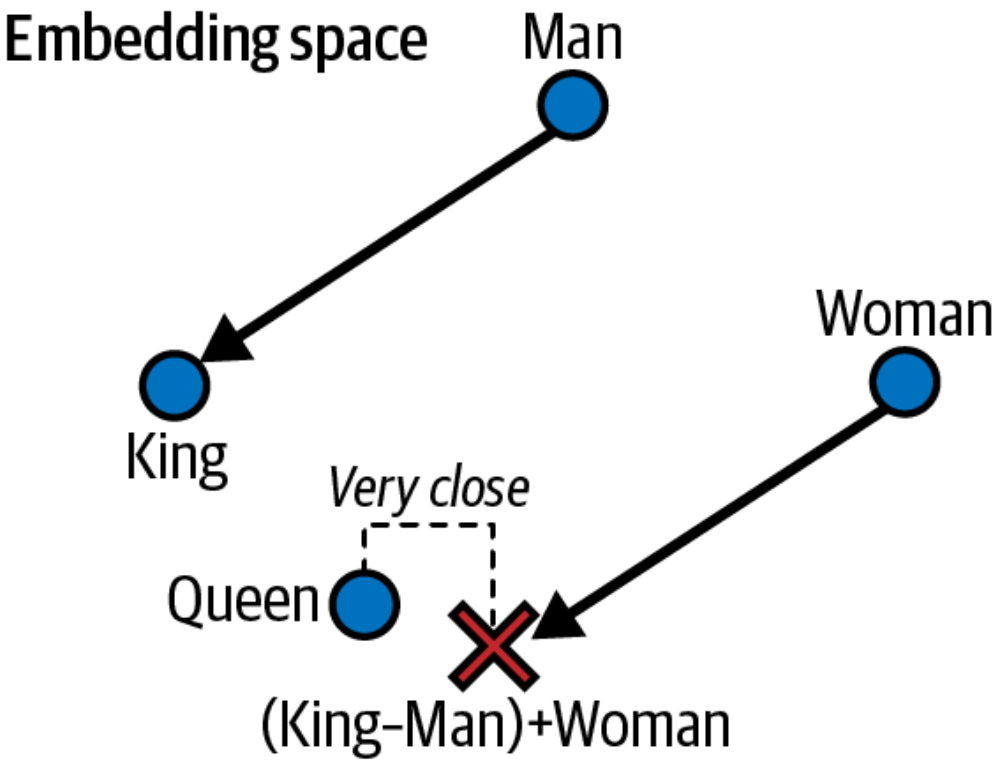


Figure 14-3. Word embeddings of similar words tend to be close, and some axes seem to encode meaningful concepts

---

WARNING

Word embeddings can have some meaningful structure, as the "King – Man + Woman" shows. However, they are also noisy and often hard to interpret. I've added some code at the end of the notebook so you can judge for yourself.

Unfortunately, word embeddings sometimes capture our worst biases. For example, although they correctly learn that *Man is to King as Woman is to Queen*, they also seem to learn that *Man is to Doctor as Woman is to Nurse*: quite a sexist bias! To be fair, this particular example is probably exaggerated, as was pointed out in a 2019 paper[4] by Malvina Nissim et al.

Nevertheless, ensuring fairness in deep learning algorithms is an important and active research topic.

PyTorch provides an `nn.Embedding` module, which wraps an *embedding matrix*: this matrix has one row per possible category (e.g., one row for each token in the vocabulary) and one column per embedding dimension. The embedding dimensionality is a hyperparameter you can tune. By default, the embedding matrix is initialized randomly.

To convert a category ID to an embedding, the `nn.Embedding` layer just looks up and returns the corresponding row. That's all there is to it! For example, let's initialize an `nn.Embedding` layer with five categories and 3D embeddings, and use it to encode some categories:

```
>>> import torch.nn as nn
>>> torch.manual_seed(42)
>>> embed = nn.Embedding(5, 3)  # 5 categories × 3D embeddings
>>> embed(torch.tensor([[3, 2], [0, 2]]))
tensor([[[ 0.2674,  0.5349,  0.8094],
         [ 2.2082, -0.6380,  0.4617]],

        [[ 0.3367,  0.1288,  0.2345],
         [ 2.2082, -0.6380,  0.4617]]], grad_fn=<EmbeddingBackward0>)
```

As you can see, category 3 gets encoded as the 3D vector `[0.2674, 0.5349, 0.8094]`, category 2 gets encoded (twice) as the 3D vector `[2.2082, -0.6380, 0.4617]`, and category 0 gets encoded as the 3D vector `[0.3367, 0.1288, 0.2345]` (categories 1 and 4 were not used in this example). Since the layer is not trained yet, these encodings are just random.

Note that an embedding layer is mathematically equivalent to one-hot encoding followed by a linear layer (with no bias parameter). For example, if you create a linear layer with `nn.Linear(5, 3, bias=False)` and pass it the one-hot vector `torch.tensor([[0., 0., 0., 1., 0.]])`, you get a vector equal to row #3 of the linear layer's transposed weight matrix (which acts as an embedding matrix). That's because all rows in the transposed weight matrix get multiplied by zero, except for row #3 which gets multiplied by 1, so the result is just row #3. However, it's much more efficient to use `nn.Embedding(5, 3)` and pass it `torch.tensor([3])`: this looks up row #3 in the embedding matrix without the need for one-hot encoding, and without all the pointless multiplications by zero.

OK, now that you know about embeddings, you are ready to build the Shakespeare model.

Building and Training the Char-RNN Model

Since our dataset is reasonably large, and modeling language is quite a difficult task, we need more than a simple RNN with a few recurrent neurons. Let's build and train a model with a two-layer `nn.GRU` module (introduced in Chapter 13), with 128 units per layer, and a bit of dropout. You can try tweaking the number of layers and units later, if needed:

```
class ShakespeareModel(nn.Module):
    def __init__(self, vocab_size, n_layers=2, embed_dim=10, hidden_dim=128,
                 dropout=0.1):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim)
        self.gru = nn.GRU(embed_dim, hidden_dim, num_layers=n_layers,
                          batch_first=True, dropout=dropout)
        self.output = nn.Linear(hidden_dim, vocab_size)

    def forward(self, X):
        embeddings = self.embed(X)
        outputs, _states = self.gru(embeddings)
        return self.output(outputs).permute(0, 2, 1)

torch.manual_seed(42)
model = ShakespeareModel(len(vocab)).to(device)
```

Let's go over this code:

- We use an `nn.Embedding` layer as the first layer, to encode the character IDs. As we just saw, the `nn.Embedding` layer's number of input dimensions is the number of categories, so in our case it's the number of distinct character IDs. The embedding size is a hyperparameter you can tune—we'll set it to 10 for now. Whereas the inputs of the `nn.Embedding` layer will be integer tensors of shape [*batch size*, *window length*], the outputs of the `nn.Embedding` layer will be float tensors of shape [*batch size*, *window length*, *embedding size*].
- The `nn.GRU` layer has 10 inputs (i.e., the embedding size), 128 outputs (i.e., the hidden size), two layers, and as usual we must specify `batch_first=True` because otherwise the layer assumes that the batch dimension comes after the time dimension.
- We use a `nn.Linear` layer for the output layer: it must have 39 units because there are 39 distinct characters in the text, and we want to output a logit for each possible character (at each time step).
- In the `forward()` method, we just call these layers one by one. Note that the `nn.GRU` layer's output shape is [*batch size*, *window length*, *hidden size*], and the `nn.Linear` layer's output shape is [*batch size*, *window length*, *vocabulary size*], but as we saw in Chapter 13, the `nn.CrossEntropyLoss` and `Accuracy` modules that we will use for training both expect the class dimension (i.e., `vocab_size`) to be the second dimension, not the last one. This is why we must permute the last two dimensions of the `nn.Linear` layer's output. Note that the `nn.GRU` layer also returns the final hidden states, but we ignore them.[5]

Now you can now train and evaluate the model as usual, using the `nn.CrossEntropyLoss` and the `Accuracy` metric.

WARNING

If you are running this code on Colab with a GPU activated, then training will take a few hours. You can reduce the number of epochs if you don't want to wait that long, but of course the model's accuracy will probably be lower. If the Colab session times out, make sure to reconnect quickly, or else the Colab run-

time will be destroyed.

And now let's use our model to predict the next character in a sentence:

```
model.eval()  # don't forget to switch the model to evaluation mode!
text = "To be or not to b"
encoded_text = encode_text(text).unsqueeze(dim=0).to(device)
with torch.no_grad():
    Y_logits = model(encoded_text)
    predicted_char_id = Y_logits[0, :, -1].argmax().item()
    predicted_char = id_to_char[predicted_char_id]  # correctly predicts "e"
```

We first encode the text, add a batch dimension of size 1, and move the tensor to the GPU. Then we call our model and get logits for each time step. We're only interested in logits for the final time step (hence the –1), and we want to know which token ID has the highest logit, so we use `argmax()`. We then use `item()` to extract the token ID from the tensor. Lastly, we convert the token ID to a character, and that's our prediction.

The model correctly predicts "e", great! Now let's use this model to pretend we're Shakespeare!

Generating Fake Shakespearean Text

To generate new text using the char-RNN model, we could feed it some text, make the model predict the most likely next letter, add it to the end of the text, then give the extended text to the model to guess the next letter, and so on. This is called *greedy decoding*. But in practice this often leads to the same words being repeated over and over again. Instead, we can sample the next character randomly, using the model's estimated probability distribution: if the model estimates a probability *p* for a given token, then this token will be sampled with probability *p*. This process will generate more diverse and interesting text since the most likely token won't always be sampled. To sample the next token, we can use the `torch.multinomial()` function, which samples random class indices, given a list of class probabilities. For example:

```
>>> torch.manual_seed(42)
>>> probs = torch.tensor([[0.5, 0.4, 0.1]])  # probas = 50%, 40%, and 10%
>>> samples = torch.multinomial(probs, replacement=True, num_samples=8)
>>> samples
tensor([[0, 0, 0, 0, 1, 0, 2, 2]])
```

To have more control over the diversity of the generated text, we can divide the logits by a number called the *temperature*, which we can tweak as we wish. A temperature close to zero favors high-probability characters, while a high temperature gives all characters an equal probability. Lower temperatures are typically preferred when generating fairly rigid and precise text, such as mathematical equations, while higher temperatures are preferred when generating more diverse and creative text. Let's write a `next_char()` helper function that will

creating more diverse and creative text. Let's write a `next_char()` helper function that will use this approach to pick the next character to add to the input text:

```
import torch.nn.functional as F

def next_char(model, text, temperature=1):
    encoded_text = encode_text(text).unsqueeze(dim=0).to(device)
    with torch.no_grad():
        Y_logits = model(encoded_text)
        Y_probas = F.softmax(Y_logits[0, :, -1] / temperature, dim=-1)
        predicted_char_id = torch.multinomial(Y_probas, num_samples=1).item()
    return id_to_char[predicted_char_id]
```

Next, we can write another small helper function that will repeatedly call `next_char()` to get the next character and append it to the given text:

```
def extend_text(model, text, n_chars=80, temperature=1):
    for _ in range(n_chars):
        text += next_char(model, text, temperature)
    return text
```

We are now ready to generate some text! Let's try low, medium, and high temperatures:

```
>>> print(extend_text(model, "To be or not to b", temperature=0.01))
To be or not to be the state
and the contrary of the state and the sea,
the common people of the
>>> print(extend_text(model, "To be or not to b", temperature=0.4))
To be or not to be the better from the cause
that thou think you may be so be gone.

romeo:
that
>>> print(extend_text(model, "To be or not to b", temperature=100))
To be or not to b-c3;m-rkn&:x:uyve:b&hi n;n-h;wt3k
&cixxh:a!kq$c$ 3 ncq$ ;;wq cp:!xq;yh
!3
d!nhi.
```

Notice the repetitions when the temperature is low: "the state and the" appears twice. The intermediate temperature led to more convincing results, although Romeo wasn't very talkative today. But in the last example the temperature was way too high, we fried Shakespeare. To generate more convincing text, a common technique is to sample only from the top *k* characters, or only from the smallest set of top characters whose total probability exceeds some threshold: this is called *top-p sampling*, or *nucleus sampling*. Alternatively, you could try using *beam search*, which we will discuss later in this chapter, or using more `nn.GRU` layers and more neurons per layer, training for longer, and adding more regularization if needed.

NOTE

The model is currently incapable of learning patterns longer than `window_length`, which is just 50 characters. You could try making this window larger, but it would also make training harder, and even LSTM and GRU cells cannot handle very long sequences.[6]

Interestingly, although a char-RNN model is just trained to predict the next character, this seemingly simple task actually requires it to learn some higher-level tasks as well. For example, to find the next character after "Great movie, I really ", it's helpful to understand that the sentence is positive, so what follows is more likely to be the letter "l" (for "loved") rather than "h" (for "hated"). In fact, a 2017 paper[7] by Alec Radford and other OpenAI researchers describes how the authors trained a big char-RNN-like model on a large dataset, and found that one of the neurons acted as an excellent sentiment analysis classifier. Although the model was trained without any labels, the *sentiment neuron*—as they called it—reached state-of-the-art performance on sentiment analysis benchmarks (this foreshadowed and motivated unsupervised pretraining in NLP).

Speaking of which, let's say farewell to Shakespeare and turn to the second part of this chapter: sentiment analysis.

Sentiment Analysis Using Hugging Face Libraries

One of the most common applications of NLP is text classification—especially sentiment analysis. If image classification on the MNIST dataset is the "Hello, world!" of computer vision, then sentiment analysis on the IMDb reviews dataset is the "Hello, world!" of natural language processing. The IMDb dataset consists of 50,000 movie reviews in English (25,000 for training, 25,000 for testing) extracted from the famous Internet Movie Database, along with a simple binary target for each review indicating whether it is negative (0) or positive (1). Just like MNIST, the IMDb reviews dataset is popular for good reasons: it is simple enough to be tackled on a laptop in a reasonable amount of time, but challenging enough to be fun and rewarding.

To download the IMDb dataset, we will use the Hugging Face *Datasets* library, which gives easy access to hundreds of thousands of datasets hosted on the Hugging Face Hub. It is preinstalled on Colab; otherwise it can be installed using `pip install datasets`. We'll use 80% of the original training set for training, and the remaining 20% for validation, using the `train_test_split()` method to split the set:

```
from datasets import load_dataset

imdb_dataset = load_dataset("imdb")
split = imdb_dataset["train"].train_test_split(train_size=0.8, seed=42)
imdb_train_set, imdb_valid_set = split["train"], split["test"]
imdb_test_set = imdb_dataset["test"]
```

Let's inspect a couple of reviews:

```
>>> imdb_train_set[0]["text"]
"'The Rookie' was a wonderful movie about the second chances life holds [...]"
>>> imdb_train_set[0]["label"]
1
>>> imdb_train_set[16]["text"]
"Lillian Hellman's play, adapted by Dashiell Hammett with help from Hellman,
becomes a curious project to come out of gritty Warner Bros. [...] It seems to
take forever for this drama to find its focus, [...], it seems a little
patronizing [...] Lukas has several speeches in the third-act which undoubtedly
won him the Academy Award [...] this tasteful, tactful movie [...] It should be
a heady mix, but instead it's rather dry-eyed and inert. ** from ****"
>>> imdb_train_set[13]["label"]
0
```

The first review immediately says that it's a wonderful movie, no need to read any further: it's clearly positive (label = 1). The second review is much harder to classify: it contains a detailed description of the movie, sprinkled with both positive and negative comments. Luckily, the conclusion is quite clearly negative, making the task much easier (label = 0). Still, it's not a trivial task.

A simple char-RNN model would struggle; we need a more powerful tokenization technique. So let's focus on tokenization before we return to sentiment analysis.

Tokenization Using the Hugging Face Tokenizers Library

In a 2016 paper,[8] Rico Sennrich et al. from the University of Edinburgh explored several methods to tokenize and detokenize text at the subword level. This way, even if your model encounters a rare word it has never seen before, it can still reasonably guess what it means. For example, even if the model never saw the word "smartest" during training, if it learned the word "smart" and it also learned that the suffix "est" means "the most", it can infer the meaning of "smartest". One of the techniques the authors evaluated is *byte pair encoding* (BPE), introduced by Philip Gage in 1994 (initially for data compression). BPE works by splitting the whole training set into individual characters, then at each iteration it finds the most frequent pair of adjacent tokens and adds it to the vocabulary. It repeats this process until the vocabulary reaches the desired size.

The Hugging Face Tokenizers library includes highly efficient implementations of several popular tokenization algorithms, including BPE. It is preinstalled on Colab (or you can install it with `pip install tokenizers`). Here's how to train a BPE model on the IMDb dataset:

```
import tokenizers

bpe_model = tokenizers.models.BPE(unk_token="<unk>")
bpe_tokenizer = tokenizers.Tokenizer(bpe_model)
```

```
bpe_tokenizer.pre_tokenizer = tokenizers.pre_tokenizers.Whitespace()
special_tokens = ["<pad>", "<unk>"]
bpe_trainer = tokenizers.trainers.BpeTrainer(vocab_size=1000,
                                             special_tokens=special_tokens)
train_reviews = [review["text"].lower() for review in imdb_train_set]
bpe_tokenizer.train_from_iterator(train_reviews, bpe_trainer)
```

Let's walk through this code:

- We import the Tokenizers library, and we create a BPE model, specifying an unknown token `"<unk>"` which will be used later if we try to tokenize some text containing tokens that the model never saw during training: the unknown tokens will be replaced with the `"<unk>"` token.
- We then create a `Tokenizer` based on the BPE model.
- The Tokenizers library lets you specify optional preprocessing and post-processing steps, and it also provides common preprocessors and postprocessors. In this example, we use the `Whitespace` preprocessor which splits the text at spaces (and drops the spaces), and also separates groups of letters and groups of nonletters. For example "Hello, world!!!" will be split into ["Hello", ",", "world", "!!!"]. The BPE algorithm will then run on these individual chunks, which dramatically speeds up training and improves token quality (at least when the text is in English) by providing reasonable word boundaries.
- We then define a list of special tokens: a padding token `"<pad>"` that will come in handy when we create batches of texts of different lengths, and the unknown token we have already discussed.
- We create a `BpeTrainer`, specifying the maximum vocabulary size and the list of special tokens. The trainer will add the special tokens at the beginning of the vocabulary, so `"<pad>"` will be token 0, and `"<unk>"` will be token 1.
- Next we create a list of all the text in the IMBd training set.
- Lastly, we train the tokenizer on this list, using the `BpeTrainer`. A few seconds later, the BPE tokenizer is ready to be used!

Now let's use our BPE tokenizer to tokenize some text:

```
>>> some_review = "what an awesome movie! 😊"
>>> bpe_encoding = bpe_tokenizer.encode(some_review)
>>> bpe_encoding
Encoding(num_tokens=8, attributes=[ids, type_ids, tokens, offsets,
         attention_mask, special_tokens_mask, overflowing])
```

The `encode()` method returns an `Encoding` object that contains eight tokens. Let's look at these tokens and their IDs:

```
>>> bpe_encoding.tokens
['what', 'an', 'aw', 'es', 'ome', 'movie', '!', '<unk>']
>>> bpe_token_ids = bpe_encoding.ids
>>> bpe_token_ids
```

```
>>> bpe_token_ids
[303, 139, 373, 149, 240, 211, 4, 1]
```

Notice that frequent words like “what” and “movie” have been identified by the BPE model and are represented by a single token, while less frequent words like “awesome” are split into multiple tokens. Also note that the smiley was not part of the training data, so it gets replaced with the unknown token `"<unk>"`.

The tokenizer provides a `get_vocab()` method which returns a dictionary mapping each token to its ID. You can also use the `token_to_id()` method to map a single token, or conversely use the `id_to_token()` method to go from ID to token. However, you will more often use the `decode()` method to convert a list of token IDs into a string:

```
>>> bpe_tokenizer.decode(bpe_token_ids)
'what an aw es ome movie !'
```

The tokenizer keeps track of each token’s start and end offset in the original string, which can come in handy, especially for debugging:

```
>>> bpe_encoding.offsets
[(0, 4), (5, 7), (8, 10), (10, 12), (12, 15), (16, 21), (21, 22), (23, 24)]
```

It’s also possible to encode a whole batch of strings at once. For example, let’s encode the first three reviews of the training set:

```
>>> bpe_tokenizer.encode_batch(train_reviews[:3])
[Encoding(num_tokens=281, attributes=[ids, type_ids, tokens, [...]]),
 Encoding(num_tokens=114, attributes=[ids, type_ids, tokens, [...]]),
 Encoding(num_tokens=285, attributes=[ids, type_ids, tokens, [...]]),
```

If we want to create a single integer tensor containing the token IDs of all three reviews, we must first ensure that they all have the same number of tokens, which is not the case right now. For this, we can ask the tokenizer to pad the shorter reviews with the padding token ID until they are as long as the longest review in the batch. We can also ask the tokenizer to truncate any sequence longer than some maximum length, since RNNs don’t handle very long sequences very well anyway:

```
bpe_tokenizer.enable_padding(pad_id=0, pad_token="<pad>")
bpe_tokenizer.enable_truncation(max_length=500)
```

Now let’s encode the batch again. This time all sequences will have the same number of tokens, so we can create a tensor containing all the token IDs:

```
>>> bpe_encodings = bpe_tokenizer.encode_batch(train_reviews[:3])
>>> bpe_batch_ids = torch.tensor([encoding.ids for encoding in bpe_encodings])
>>> bpe_batch_ids
tensor([[159, 402, 176, 246,  61, [...], 215, 156, 586,   0,   0,   0,   0],
        [ 10, 138, 198, 289, 175, [...],   0,   0,   0,   0,   0,   0,   0],
        [289,  15, 209, 398, 177, [...],  50,  29,  22,  17,  24,  18,  24]])
```

Notice how the first and second review were padded with 0s, which is our padding token ID. Each `Encoding` object also includes an `attention_mask` attribute containing a 1 for each nonpadding token, and a 0 for each padding token. This can be used in your models to easily ignore the padded time steps: just multiply a tensor with the attention mask. In some cases you will prefer to have the list of sequence lengths (ignoring padding). Here's how to get both the attention mask tensor and the sequence lengths:

```
>>> attention_mask = torch.tensor([encoding.attention_mask
...                                for encoding in bpe_encodings])
...
>>> attention_mask
tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, [...], 1, 1, 1, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, [...], 0, 0, 0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, [...], 1, 1, 1, 1, 1, 1, 1]])
>>> lengths = attention_mask.sum(dim=-1)
>>> lengths
tensor([281, 114, 285])
```

You may have noted that spaces were not handled very well by our tokenizer. In particular, the word "awesome" came back as "aw es ome", and "movie!" came back as "movie !". This is because the `Whitespace` pre-tokenizer dropped all spaces, therefore the BPE tokenizer doesn't know where spaces should go and it just adds spaces between all tokens. To fix this, we can replace the `Whitespace` pre-tokenizer with the `ByteLevel` pre-tokenizer: it replaces all spaces with a special character Ġ so the BPE model doesn't lose track of them. For example, if you use this pre-tokenizer and you encode and decode the text "what an awesome movie! 😊", you will get: "Ġwhat Ġan Ġaw es ome Ġmovie !". After removing the spaces, then replacing every Ġ with a space, you get " what an awesome movie!". This is almost perfect, except for the extra space at the start—which is easily removed—and the lost emoji, which was replaced with an unknown token because it's not in the vocabulary, and dropped by the `decode()` method.

As its name suggests, the `ByteLevel` pre-tokenizer allows the BPE model to work at the byte level, rather than the character level: unsurprisingly, this is called Byte-level BPE (BBPE). For example, the 😊 emoji will be converted to four bytes, using Unicode's UTF-8 encoding. This means that BBPE will never output an unknown token if its vocabulary contains all 256 possible bytes, since any text can be broken down into its individual bytes whenever longer tokens are not found in the vocabulary. This makes BBPE well suited when the corpus contains rare characters such as emojis.

Another important variant of BPE is WordPiece,[9], proposed by Google in 2016. This tokeniza-

tion algorithm is very similar to BPE, but instead of adding the most frequent adjacent pair of tokens to the vocabulary at each iteration, it adds the pair with the highest score. This score is computed using Equation 14-1: the frequency(AB) term is just like in BPE—it boosts pairs that are frequent in the corpus. However, the denominator reduces the score of a pair when the individual tokens are themselves frequent. This normalization tends to favor more useful and meaningful tokens than BPE, and the algorithm often produces shorter encoded sequences than BPE or BBPE.

Equation 14-1. WordPiece score for a pair AB composed of tokens A and B

To train a WordPiece tokenizer using the Tokenizers library, you can use the same code as for BPE, but replace the `BPE` model with `WordPiece`, and the `BpeTrainer` with `WordPieceTrainer`. If you encode and decode the same review as earlier, you will get "what an aw esome movie !". Notice that WordPiece adds a prefix to tokens that are inside a word, which makes it easy to reconstruct the original string: just remove " ##" (as well as spaces before punctuations). Note that the smiley emoji once again disappeared because it was not in the vocabulary.

One last popular tokenization algorithm we will discuss is Unigram LM (Language Model), introduced in a 2018 paper[10] by Taku Kudo at Google. This technique is a bit different than the previous ones: it starts out with a very large vocabulary containing every frequent word, subword, and character in the training corpus, then it gradually removes the least useful tokens until it reaches the desired vocabulary size. To determine how useful a token is, this method makes one big simplifying assumption: it assumes that the corpus was sampled randomly from the vocabulary, one token at a time (hence the name Unigram LM), and that every token was sampled independently from the others. Therefore, this tokenizer model assumes that the probability of sampling the pair AB is equal to the probability of sampling A times the probability of sampling B. Given this assumption, it can estimate the probability of sampling the whole training corpus. At each iteration, the training algorithm attempts to remove tokens without reducing this overall probability too much.

For example, suppose that the vocabulary contains the tokens "them", "the", and "m", respectively, with 1%, 5%, and 2% probability. This means that the word "them" has a 1% chance of being sampled as the single token "them", or a 5% × 2% = 0.1% chance of being sampled as the pair "the" + "m". Overall, the word "them" has a 1% + 0.1% = 1.1% chance of being sampled. If we remove the token "them" from the vocabulary, then the probability of sampling the word "them" drops down to 0.1%. If instead we remove either "m" or "the", then the probability only drops down to 1% since we can still sample the single token "them". So if the training corpus only contained the word "them", then the algorithm would prefer to drop either "the" or "m". Of course, in reality the corpus contains many other words that contain these two tokens, so the algorithm will likely find other less useful tokens to drop.

TIP

Unigram LM is great for languages that don't use spaces to separate words, like English does. For exam-

Unigram LM is great for languages that don't use spaces to separate words, like English does. For example, Chinese text does not use spaces between words, Vietnamese uses spaces even within words, and German often attaches multiple words together, without spaces.

The same paper also proposed a novel regularization technique called *subword regularization*, which improves generalization and robustness by introducing some randomness in tokenization while training the NLP model (not the tokenizer model). For example, assuming the vocabulary contains the tokens "them", "the", and "m", and you choose to use subword regularization, then the word "them" will sometimes be tokenized as "the" + "m", and sometimes as "them". This technique works best with *morphologically rich languages*, meaning languages where words carry a lot of grammatical information through affixes, inflections, and internal modifications (such as Arabic, Finnish, German, Hungarian, Polish, or Turkish), as opposed to languages that rely on word order or additional helper words (such as English, Chinese, Thai, or Vietnamese).

Unfortunately, the Tokenizers library does not natively support subword regularization, so you either have to implement it yourself, or you can use Google's *SentencePiece* library (`pip install sentencepiece`) which provides an open source implementation. This project is described in a 2018 paper[11] by Taku Kudo and John Richardson.

Table 14-1 summarizes the three main tokenizers used today.

Table 14-1. Overview of the three main tokenizers

Feature	BBPE	WordPiece	Unigram LM
How	Merge most frequent pairs	Merge pairs that maximize data likelihood	Remove least likely tokens
Pros	Fast, simple, great for multilingual	Good balance of efficiency and token quality	Most meaningful, shortest sequences
Cons	Can produce awkward splits	Less robust than BBPE for multilingual	Slower to train and tokenize
Used By	GPT, Llama, RoBERTa, BLOOM	BERT, DistilBERT, ELECTRA	T5, ALBERT, mBART

Training your own tokenizer is useful in many situations; for example, if you are dealing with domain-specific text, such as medical, legal, or engineering documents full of jargon, or if the text is written in a low-resource language or dialect, or if it's code written in a new programming language, and so on. However, in most cases you will want to simply reuse a pretrained tokenizer.

Reusing Pretrained Tokenizers

To download a pretrained tokenizer, we will use the Hugging Face *Transformers library*. This library provides many popular models for NLP, computer vision, audio processing, and more.

Pretrained weights are available for almost all of these models, and the library can automatically download them from the Hugging Face Hub. The models were originally all based on the *Transformer architecture* (which we will discuss in detail in Chapter 15), hence the name of the library, but other kinds of models are now available as well, such as CNNs. Lastly, each model comes with all the tools it needs, including tokenizers for NLP models: in a single line of code, you can have a fully functional, high-performance model for a given task, as we will see later in this chapter.

For now, let's just grab the pretrained tokenizer from some NLP model. For example, the following code downloads the pretrained BBPE tokenizer used by the GPT-2 model (a text generation model), and it uses this tokenizer to encode the first 3 IMDb reviews, truncating the encoded sequences if they exceed 500 tokens:

```
import transformers

gpt2_tokenizer = transformers.AutoTokenizer.from_pretrained("gpt2")
gpt2_encoding = gpt2_tokenizer(train_reviews[:3], truncation=True,
                               max_length=500)
```

Notice that we use the tokenizer object like a function. The result is a dictionary-like object of type `BatchEncoding`. You can get the token IDs using the `"input_ids"` key. It returns a Python list of lists of token IDs. For example, let's look at the first 10 token IDs of the first encoded review, and use the tokenizer to decode them, using its `decode()` method:

```
>>> gpt2_token_ids = gpt2_encoding["input_ids"][0][:10]
>>> gpt2_token_ids
[14247, 35030, 1690, 423, 257, 1688, 8046, 13, 484, 1690]
>>> gpt2_tokenizer.decode(gpt2_token_ids)
'stage adaptations often have a major fault. they often'
```

If you would prefer to use a pretrained WordPiece tokenizer, you can reuse the tokenizer of any LLM that was pretrained using WordPiece, such as BERT (another popular NLP model, which stands for Bidirectional Encoder Representations from Transformers). This tokenizer has a padding token (unlike the previous tokenizer, since GPT-2 didn't need it), so we can specify `padding=True` when encoding a batch of reviews: as usual, the shortest texts will be padded to the length of the longest one using the padding token. This allows us to also specify `return_tensors="pt"` to get a PyTorch tensor instead of a Python list of lists of token IDs: very convenient! So let's encode the first three IMDb reviews:

```
bert_tokenizer = transformers.AutoTokenizer.from_pretrained("bert-base-uncased")
bert_encoding = bert_tokenizer(train_reviews[:3], padding=True,
                               truncation=True, max_length=500,
                               return_tensors="pt")
```

TIP

The name `"bert-base-uncased"` refers to a *model checkpoint*: this particular checkpoint is a case-insensitive BERT model, pretrained on English text. Other checkpoints are available, such as `"bert-large-cased"` if you want a larger and case-sensitive BERT model, or `"bert-base-multilingual-uncased"` if you want an uncased model pretrained on over 100 languages. For now we are just using the model's tokenizer.

The resulting token IDs and attention masks are nicely padded tensors:

```
>>> bert_encoding["input_ids"]
tensor([[ 101, 2754,17241, 2411, 2031, [...],  102, 0, 0, 0, [...], 0, 0, 0],
        [ 101, 1005, 1996, 8305, 1005, [...],  102, 0, 0, 0, [...], 0, 0, 0],
        [ 101, 7929, 1010, 2021, 2515, [...], 1012,  1019,  1013,  1019,  102]])
>>> bert_encoding["attention_mask"]
tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, [...], 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, [...], 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, [...], 1, 1, 1, 1, 1, 1, 1, 1, 1]])
```

Notice that each token ID sequence starts with token 101 ([CLS]), and ends with token 102 ([SEP]) (ignoring padding tokens). These tokens are needed by the BERT model (as we will see in Chapter 15), but unless your model needs them too, you can drop them by setting `add_special_tokens=False` when calling the tokenizer.

What about a pretrained Unigram LM tokenizer? Well, many models were trained using Unigram LM, such as ALBERT, T5, or XML-R models, just to name a few. For example:

```
albert_tokenizer = transformers.AutoTokenizer.from_pretrained("albert-base-v2")
albert_encoding = albert_tokenizer(train_reviews[:3], padding=True, [...])
```

The Transformers library also provides an object that can wrap your own tokenizer (from the Tokenizers library) and give it the same API as the pretrained tokenizers (from the Transformers library). For example, let's wrap the BPE tokenizer we trained earlier:

```
hf_tokenizer = transformers.PreTrainedTokenizerFast(
    tokenizer_object=bpe_tokenizer)
hf_encodings = hf_tokenizer(train_reviews[:3], padding=True, [...])
```

With that, we have all the tokenization tools we need, so let's go back to sentiment analysis.

Building and Training a Sentiment Analysis Model

Our sentiment analysis model must be trained using batches of tokenized reviews. However, the datasets we created did not take care of tokenization. One option would be to update them (e.g., using the `map()` method), but it's just as simple to handle tokenization in the data load

(e.g., using the `map()` method), but it's just as simple to handle tokenization in the data loaders. To do this, we can pass a function to the `DataLoader` constructor using its `collate_fn` argument: the data loader will call this function for every batch, passing it a list of dataset samples. Our function will take this batch, tokenize the reviews, truncate and pad them if needed, and return a `BatchEncoding` object containing PyTorch tensors for the token IDs and attention masks, along with another tensor containing the labels. For tokenization, we will simply use the pretrained WordPiece tokenizer we just loaded:

```
def collate_fn(batch, tokenizer=bert_tokenizer):
    reviews = [review["text"] for review in batch]
    labels = [[review["label"]] for review in batch]
    encodings = tokenizer(reviews, padding=True, truncation=True,
                          max_length=200, return_tensors="pt")
    labels = torch.tensor(labels, dtype=torch.float32)
    return encodings, labels

batch_size = 256
imdb_train_loader = DataLoader(imdb_train_set, batch_size=batch_size,
                               collate_fn=collate_fn, shuffle=True)
imdb_valid_loader = DataLoader(imdb_valid_set, batch_size=batch_size,
                               collate_fn=collate_fn)
imdb_test_loader = DataLoader(imdb_test_set, batch_size=batch_size,
                              collate_fn=collate_fn)
```

Now we're ready to create our sentiment analysis model:

```
class SentimentAnalysisModel(nn.Module):
    def __init__(self, vocab_size, n_layers=2, embed_dim=128, hidden_dim=64,
                 pad_id=0, dropout=0.2):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim,
                                  padding_idx=pad_id)
        self.gru = nn.GRU(embed_dim, hidden_dim, num_layers=n_layers,
                          batch_first=True, dropout=dropout)
        self.output = nn.Linear(hidden_dim, 1)

    def forward(self, encodings):
        embeddings = self.embed(encodings["input_ids"])
        _outputs, hidden_states = self.gru(embeddings)
        return self.output(hidden_states[-1])
```

As you can see, this model is very similar to our Shakespeare model, but with a few important differences:

- When creating the `nn.Embedding` layer, we set its `padding_idx` argument to our padding ID. This ensures that the padding ID gets embedded as a nontrainable zero vector to reduce the impact of padding tokens on the loss.
- Since this is a sequence-to-vector model, not a sequence-to-sequence model, we only need

the last output of the top GRU layer to make our final prediction (through the output `nn.Linear` layer). We could have used `outputs[:, -1]` instead of `hidden_states[-1]`, as they are equal.
- The output `nn.Linear` layer has a single output dimension because it's a binary classification model. The final output will be a 2D tensor with a single column containing one logit per review, positive for positive reviews, and negative for negative reviews.
- The `forward()` method takes a `BatchEncoding` object as input, containing the token IDs (possibly padded and truncated).

We can then train this model using the `nn.BCEWithLogitsLoss` since this is a binary classification task. It reaches 85% accuracy on the validation set, which is reasonably good, although the best models reach human level, slightly above 90% accuracy. It's probably not possible to go much higher than that, because many reviews are ambiguous, classifying them feels like flipping a coin.

One problem with our model is the fact that we are not fully ignoring the padding tokens. Indeed, if a review ends with many padding tokens, the `nn.GRU` module will have to process them, and by the time it gets through all of them, it might have forgotten what the review was all about. To avoid this, we can use a *packed sequence* instead of a regular tensor. A packed sequence is a special data structure designed to efficiently represent a batch of sequences of variable lengths.[12] You can use the `pack_padded_sequence()` function to convert a tensor containing padded sequences to a packed sequence object, and conversely you can use the `pad_packed_sequence()` function whenever you want to convert a packed sequence object to a padded tensor:

```
>>> from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence
>>> sequences = torch.tensor([[1, 2, 0, 0], [5, 6, 7, 8]])
>>> packed = pack_padded_sequence(sequences, lengths=(2, 4),
...                               enforce_sorted=False, batch_first=True)
...
>>> packed
PackedSequence(data=tensor([5, 1, 6, 2, 7, 8]), [...])
>>> padded, lengths = pad_packed_sequence(packed, batch_first=True)
>>> padded, lengths
(tensor([[1, 2, 0, 0],
         [5, 6, 7, 8]]),
 tensor([2, 4]))
```

By default, the `pack_padded_sequence()` function assumes that the sequences in the batch are ordered from the longest to the shortest. If this is not the case, you must set `enforce_sorted=False`. Moreover, the function also assumes that the time dimension comes before the batch dimension. If the batch dimension is first, you must set `batch_first=True`.

PyTorch's recurrent layers support packed sequences: they efficiently process the sequences, stopping at the end of each sequence. So let's update out sentiment analysis model to use packed sequences: in the `forward()` method, just replace the `self.gru(embeddings)` line

packed sequences. In the `forward()` method, just replace the `self.gru(embeddings)` line with the following code:

```
lengths = encodings["attention_mask"].sum(dim=1)
packed = pack_padded_sequence(embeddings, lengths=lengths.cpu(),
                              batch_first=True, enforce_sorted=False)
_outputs, hidden_states = self.gru(packed)
```

This code starts by computing the length of each sequence in the batch, just like we did earlier, then it packs the embeddings tensor and passes the packed sequence to the `nn.GRU` module. With that, the model will properly handle sequences without being bothered by any padding tokens. You don't actually need to set `padding_idx` anymore when creating the `nn.Embedding` layer, but it doesn't hurt, and it makes debugging a bit easier, so I prefer to keep it.

NOTE

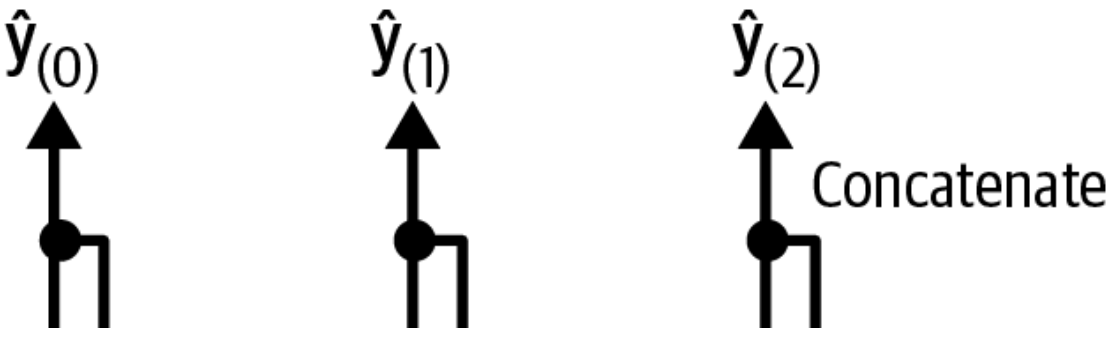
If you pass a packed sequence to a `nn.GRU` module, its outputs will also be a packed sequence, and you will need to convert it back to a padded tensor before you can pass it to the next layers. Luckily, we don't need these outputs for our sentiment analysis model, only the hidden states.

Another way to improve our model is to let it look at the review in both directions: left to right, and right to left. Let's see how this works.

Bidirectional RNNs

At each time step, a regular recurrent layer only looks at past and present inputs before generating its output. In other words, it is *causal*, meaning it cannot look into the future. This type of RNN makes sense when forecasting time series, or in the decoder of a sequence-to-sequence (seq2seq) model. But for tasks like text classification, or in the encoder of a seq2seq model, it is often preferable to look ahead at the next words before encoding a given word.

For example, consider the phrases "the right arm", "the right person", and "the right to speak": to properly encode the word "right", you need to look ahead. One solution is to run two recurrent layers on the same inputs, one reading the words from left to right and the other reading them from right to left, then combine their outputs at each time step, typically by concatenating them. This is what a *bidirectional recurrent layer* does (see Figure 14-4).



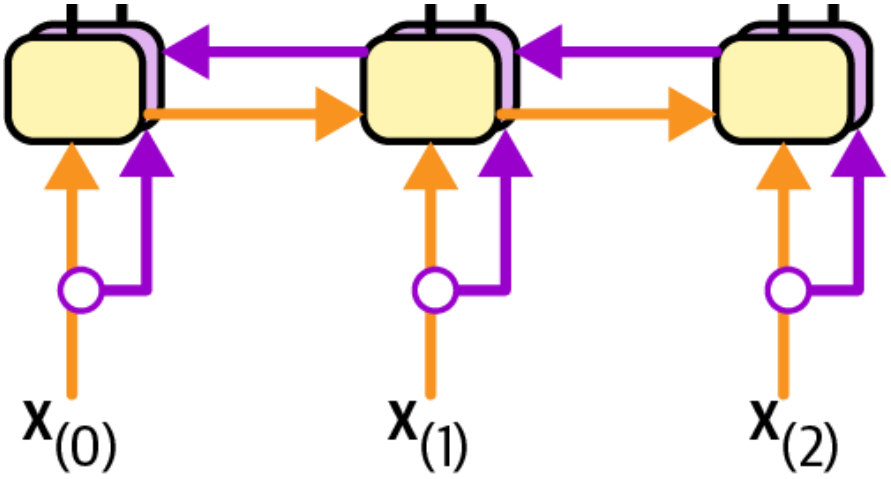


Figure 14-4. A bidirectional recurrent layer

To make our sentiment analysis model bidirectional, we can just set `bidirectional=True` when creating the `nn.GRU` layer (this also works with the `nn.RNN` and `nn.LSTM` modules).

However, once we do that, we must adjust our model a bit. In particular, we must double the input dimension of the output `nn.Linear` layer, since the hidden states will double in size:

```
self.output = nn.Linear(2 * hidden_dim, 1)
```

We must also concatenate the forward and backward hidden states of the GRU's top layer before passing the result to the output layer. For this, we can replace the last line of the `forward()` method (i.e., `return self.output(hidden_states[-1])`) with the following code:

```
n_dims = self.output.in_features
top_states = hidden_states[-2:].permute(1, 0, 2).reshape(-1, n_dims)
return self.output(top_states)
```

Let's see how the middle line works:

- Until now, the shape of the hidden states returned by the `nn.GRU` module was [*number of layers*, *batch size*, *hidden size*], so [2, 256, 64] in our case. But when we set `bidirectional=True`, we doubled the first dimension size, so we now have a shape of [4, 256, 64]: the tensor contains the hidden states for layer 1 forward, layer 1 backward, layer 2 forward, and layer 2 backward. Since we only want the top layer's hidden states, both forward and backward, we must get `hidden_states[-2:]`.
- We also need to concatenate the forward and backward states. One way to do this is to permute the first two dimensions of the top hidden states using `permute(1, 0, 2)` to get the shape [256, 2, 64], then reshape the result using `reshape(-1, n_dims)` (where `n_dims` equals 128) to get the desired shape: [256, 2 * 64].

NOTE

In this model we only use the last hidden states, ignoring the outputs at each time step. If you ever want

to use the outputs of a bidirectional module, be aware that its last dimension's size will be doubled.

You can try training this model, but you will not see any improvement in this case, because the first model actually overfit the training set, and this new version makes it even worse: it reaches 99.6% accuracy on the training set, but just 83.8% on the validation set. To fix this, you could try to regularize the model a bit more, reduce the size of the model, or increase the size of the training set.

But let's instead try something different: using pretrained embeddings.

Reusing Pretrained Embeddings and Language Models

Our model was able to learn useful embeddings for thousands of tokens, based on just 25,000 movie reviews: that's quite impressive! Imagine how good the embeddings would be if we had billions of reviews to train on. The good news is that we can reuse word embeddings even when they were trained on some other (very) large text corpus, even if it was not composed of movie reviews, and even if they were not trained for sentiment analysis. After all, the word "amazing" generally has the same meaning whether you use it to talk about movies or anything else.

Since we used pretrained tokens for the BERT model, we might as well try using its embedding layer. First, we need to download the pretrained model using the `AutoModel.from_pretrained()` function from the Transformers library, then we can directly access its embeddings layer:

```
>>> bert_model = transformers.AutoModel.from_pretrained("bert-base-uncased")
>>> bert_model.embeddings.word_embeddings
Embedding(30522, 768, padding_idx=0)
```

As you can see, this BERT model is implemented using PyTorch, and it contains a regular `nn.Embedding` layer. We could just replace our model's `nn.Embedding` layer with this one (and retrain our model), but we can keep models cleanly separated by initializing our own `nn.Embedding` layer with a copy of the pretrained embedding matrix. This can be done using the `Embedding.from_pretrained()` function:

```
class SentimentAnalysisModelPreEmbeds(nn.Module):
    def __init__(self, pretrained_embeddings, n_layers=2, hidden_dim=64,
                 dropout=0.2):
        super().__init__()
        weights = pretrained_embeddings.weight.data
        self.embed = nn.Embedding.from_pretrained(weights, freeze=True)
        embed_dim = weights.shape[-1]
        [...]  # the rest of the model is exactly like earlier

imdb_model_bert_embeds = SentimentAnalysisModelPreEmbeds(
```

```
    bert_model.embeddings.word_embeddings).to(device)
```

Note that we set `freeze=True` when creating the `nn.Embedding` layer: this makes it non-trainable and ensures that the pretrained embeddings won't be damaged by large gradients at the beginning of training. You can train the model for a few epochs like this, then make the embedding layer trainable and continue training, letting the model fine-tune the embeddings for our task.

Pretrained word embeddings have been popular for quite a while, starting with Google's Word2vec embeddings (2013), Stanford's GloVe embeddings (2014), Facebook's FastText embeddings (2016), and more. However, this approach has its limits. In particular, a word has a single representation, no matter the context. For example, the word "right" is encoded the same way in "left and right" and "right and wrong", even though it means two very different things. To address this limitation, a 2018 paper[13] by Matthew Peters introduced *Embeddings from Language Models* (ELMo): these are contextualized word embeddings learned from the internal states of a deep bidirectional RNN language model. In other words, instead of just using pretrained word embeddings in your model, you can reuse several layers of a pretrained language model.

At roughly the same time, the Universal Language Model Fine-Tuning (ULMFiT) paper[14] by Jeremy Howard and Sebastian Ruder demonstrated the effectiveness of unsupervised pretraining for NLP tasks. The authors trained an LSTM language model on a huge text corpus using self-supervised learning (i.e., generating the labels automatically from the data), then they fine-tuned it on various tasks. Their model outperformed the state of the art on six text classification tasks by a large margin (reducing the error rate by 18–24% in most cases). Moreover, the authors showed that a pretrained model fine-tuned on just 100 labeled examples could achieve the same performance as one trained from scratch on 10,000 examples. Before the ULMFiT paper, using pretrained models was only the norm in computer vision; in the context of NLP, pretraining was limited to word embeddings. This paper marked the beginning of a new era in NLP: today, reusing pretrained language models is the norm.

For example, why not reuse the entire pretrained BERT model for our sentiment analysis model? To use the BERT model, the Transformers library lets us call it like a function, passing it the tokenized reviews:

```
>>> bert_encoding = bert_tokenizer(train_reviews[:3], padding=True,
...                                max_length=200, truncation=True,
...                                return_tensors="pt")
...
>>> bert_output = bert_model(**bert_encoding)
>>> bert_output.last_hidden_state.shape
torch.Size([3, 200, 768])
```

BERT's output includes an attribute named `last_hidden_state`, which contains contextualized embeddings for each token. The word "last" in this case refers to the last layer, not the last

time step (BERT is a transformer, not an RNN). This `last_hidden_state` tensor has a shape of [*batch size*, *max sequence length*, *hidden size*]. Let's use these contextualized embeddings in a sentiment analysis model:

```
class SentimentAnalysisModelBert(nn.Module):
    def __init__(self, n_layers=2, hidden_dim=64, dropout=0.2):
        super().__init__()
        self.bert = transformers.AutoModel.from_pretrained("bert-base-uncased")
        embed_dim = self.bert.config.hidden_size
        self.gru = nn.GRU(embed_dim, hidden_dim, [...])
        self.output = nn.Linear(hidden_dim, 1)

    def forward(self, encodings):
        contextualized_embeddings = self.bert(**encodings).last_hidden_state
        lengths = encodings["attention_mask"].sum(dim=1)
        packed = pack_padded_sequence(contextualized_embeddings, [...])
        _outputs, hidden_states = self.gru(packed)
        return self.output(hidden_states[-1])
```

Note that we don't need to make the `nn.GRU` module bidirectional since the contextualized embeddings already looked ahead.

If you freeze the BERT model (e.g., using `model.bert.requires_grad_(False)`) and train it, you will notice a significant performance boost, reaching almost 90% accuracy. Wonderful!

Another option is to use only the contextualized embedding for the very first token, which is the *class token* [CLS]. Indeed, during pretraining, the BERT model had to perform a text classification task based solely on this token's contextualized embedding (we will discuss BERT pretraining in more detail in Chapter 15). As a result, it learned to summarize the most important features of the text into this embedding. This simplifies our model quite a bit, since we can get rid of the `nn.GRU` module altogether, and the `forward()` method becomes much shorter:

```
def forward(self, encodings):
    bert_output = self.bert(**encodings)
    return self.output(bert_output.last_hidden_state[:, 0])
```

In fact, the BERT model contains an extra hidden layer on top of the class embedding, composed of a `nn.Linear` module and a `nn.Tanh` module. This hidden layer is called the *pooler*. To use it, just replace `bert_output.last_hidden_state[:, 0]` with `bert_output.pooler_output`. You may also want to unfreeze the pooler after a few epochs to fine-tune it for the IMDb task.

So we started by reusing only the pretrained tokenizer, then we reused the pretrained embeddings, then most of the pretrained BERT model, and finally the full model, adding only a `nn.Linear` layer on top of the pooler. We can actually go one step further and just use an off-the-shelf class for sentence classification.

Task-Specific Classes

To tackle our binary classification task using BERT, we can use the `BertForSequenceClassification` class provided by the Transformers library. It's just a BERT model plus a classification head on top. All you need to do to create this model is specify the pretrained BERT checkpoint you want to use and the number of output units for your classification task:

```
from transformers import BertForSequenceClassification

torch.manual_seed(42)
bert_for_binary_clf = BertForSequenceClassification.from_pretrained(
    "bert-base-uncased", num_labels=2)
```

TIP

The Transformers library contains many task-specific classes based on various pretrained models, such as `BertForQuestionAnswering` or `RobertaForSequenceClassification` (see Chapter 15). You can also use `AutoModelForSequenceClassification` to let the library pick the right class for you, based on the requested model checkpoint (e.g., if you ask for `"bert-base-uncased"`, you will get an instance of `BertForSequenceClassification`). Similar `AutoModelFor[...]` classes are available for other tasks.

Until now we have always used a single output for binary classification, so why did we set `num_labels=2`? Well, for simplicity Hugging Face prefers to treat binary classification exactly like multiclass classification, so this model will output two logits instead of one, and it must be trained using the `nn.CrossEntropyLoss` instead of `nn.BCELoss` or `nn.BCEWithLogitsLoss`. If you want to convert the logits to estimated probabilities, you must use `torch.softmax()` rather than `torch.sigmoid()`.

Let's call this model on a very positive review:

```
>>> encoding = bert_tokenizer(["This was a great movie!"])
>>> output = bert_for_binary_clf(
...     input_ids=torch.tensor(encoding["input_ids"]),
...     attention_mask=torch.tensor(encoding["attention_mask"]))
...
>>> output.logits
tensor([[0.2468, 0.3499]], grad_fn=<AddmmBackward0>)
>>> torch.softmax(output.logits, dim=-1)
tensor([[0.4743, 0.5257]], grad_fn=<SoftmaxBackward0>)
```

We first tokenize the review, then we call the model, it returns a `ModelOutput` object containing the logits, and we convert these logits to estimated probabilities using the

`torch.softmax()` function. Ouch! The model classified this review as negative with 52.57% confidence! Indeed, the BERT model inside `BertForSequenceClassification` is pretrained, but not the classification head, so we're going to get terrible performance until we actually train this model on the IMDb dataset.

If you pass labels when calling this model (or any other model from the Transformers library), then it also computes the loss and returns it in the `ModelOutput` object. For example:

```
>>> output = bert_for_binary_clf(
...     input_ids=torch.tensor(encoding["input_ids"]),
...     attention_mask=torch.tensor(encoding["attention_mask"]),
...     labels=torch.tensor([1]))
...
>>> output.loss
tensor(0.6429, grad_fn=<NllLossBackward0>)
```

Since `num_labels` is greater than 1, the model computes the `nn.CrossEntropyLoss` (which is implemented as `nn.LogSoftmax` followed by `nn.NLLLoss`—that's why we see `grad_fn=<NllLossBackward0>`). If we had used `num_labels=1`, then the model would have used the `nn.MSELoss` instead; this can be useful for regression tasks.

We could now train this model using our own training code, as we did so far, but the Transformers library provides a convenient *Trainer API*, so let's check it out.

The Trainer API

The Trainer API lets you fine-tune a model on your own dataset with very little boilerplate code. It can save model checkpoints during training, apply early stopping, distribute the computations across GPUs, log metrics, take care of padding, batching, shuffling, and more. Let's use the Trainer API to train our IMDb model.

The Trainer API works directly with dataset objects, not data loaders, but it expects the datasets to contain tokenized text, not strings, so we must take care of tokenization. We can do this quite simply using the dataset's `map()` method (this method is implemented by the Datasets library; it's not available on pure PyTorch datasets):

```
def tokenize_batch(batch):
    return bert_tokenizer(batch["text"], truncation=True, max_length=200)

tok_imdb_train_set = imdb_train_set.map(tokenize_batch, batched=True)
tok_imdb_valid_set = imdb_valid_set.map(tokenize_batch, batched=True)
tok_imdb_test_set = imdb_test_set.map(tokenize_batch, batched=True)
```

Since we set `batched=True`, the `map()` method passes batches of reviews to the `tokenize_batch()` method: this is optional, but it significantly speeds up this preprocessing step. The `tokenize_batch()` method tokenizes the given batch of reviews, and the resulting

fields are added to each instance by the `map()` method. This includes fields such as `token_ids` and `attention_mask`, which the model expects.

To evaluate our model, we can write a simple function that takes an object with two attributes: `label_ids` and `predictions`:

```
def compute_accuracy(pred):
    return {"accuracy": (pred.label_ids == pred.predictions.argmax(-1)).mean()}
```

TIP

Alternatively, you can use metrics provided by the Hugging Face *Evaluate library*: they are designed to work nicely with the Transformers library. Alternatively, although the Trainer API does not support the streaming metrics from the TorchMetrics library, you can still use them if you wrap them inside a function.

Next, we must specify our training configuration in a `TrainingArguments` object:

```
from transformers import TrainingArguments

train_args = TrainingArguments(
    output_dir="my_imdb_model", num_train_epochs=2,
    per_device_train_batch_size=128, per_device_eval_batch_size=128,
    eval_strategy="epoch", logging_strategy="epoch", save_strategy="epoch",
    load_best_model_at_end=True, metric_for_best_model="accuracy")
```

We specify that the logs and model checkpoints must be saved in the `my_imdb_model` directory; training should run for 2 epochs (you can increase this if you want); the batch size is 128 for both training and evaluation (you can tweak this depending on the amount of VRAM you have); we want evaluation, logging, and saving to take place at the end of each epoch; and the best model should be loaded at the end of training based on the validation accuracy.

Lastly, we create a `Trainer` object and pass it the model, along with the training arguments, the training and validation sets, the evaluation function, plus a data collator which will take care of padding. Finally, we call the trainer's `train()` method, and we're done! The model reaches about 90% accuracy on the validation set after just two epochs:

```
from transformers import DataCollatorWithPadding, Trainer

trainer = Trainer(
    bert_for_binary_clf, train_args, train_dataset=tok_imdb_train_set,
    eval_dataset=tok_imdb_valid_set, compute_metrics=compute_accuracy,
    data_collator=DataCollatorWithPadding(bert_tokenizer))
train_output = trainer.train()
```

Great, you now know how to download a pretrained model like BERT and fine-tune it on your own dataset! But what if you don't have a dataset at all, and you just want to use a pretrained model that was already fine-tuned for sentiment analysis? For this, you can use the *pipelines API*.

Hugging Face Pipelines

The Transformers library provides a very convenient API to download and use pretrained pipelines for various tasks. Each pipeline contains a pretrained model along with its corresponding preprocessing and post-processing modules. For example let's create a sentiment analysis pipeline and run it on the first 10 IMDb reviews in the training set:

```
>>> from transformers import pipeline
>>> model_name = "distilbert-base-uncased-finetuned-sst-2-english"
>>> classifier_imdb = pipeline("sentiment-analysis", model=model_name,
...                            truncation=True, max_length=512)
...
>>> classifier_imdb(train_reviews[:10])
[{'label': 'POSITIVE', 'score': 0.9996108412742615},
 {'label': 'POSITIVE', 'score': 0.9998623132705688},
 [...]
 {'label': 'POSITIVE', 'score': 0.9978922009468079},
 {'label': 'NEGATIVE', 'score': 0.9997020363807678}]
```

Well, it could hardly be any easier, could it? Just create a pipeline by specifying the task and the model to use, and a couple of other parameters, depending on the task, and off you go! In this example, each review gets a `"POSITIVE"` or `"NEGATIVE"` label, along with a score equal to the model's estimated probability for that label. This particular model actually reaches 88.2% accuracy on the validation set, which is reasonably good. Here a few points to note:

- If you don't specify a model, the `pipeline()` function will use the default model for the chosen task. For sentiment analysis, at the time of writing, it's the model we chose: it's a DistilBERT model—a scaled down version of BERT—with an uncased tokenizer, trained on the English Wikipedia and a corpus of English books, and fine-tuned on the Stanford Sentiment Treebank v2 (SST 2) task.
- The pipeline automatically uses the GPU if you have one. If you have several GPUs, you can specify which one to use by setting the pipeline's `device` argument to the GPU index.
- The models from the Transformers library are always in evaluation mode by default (no need to call `eval()`).
- The score is for the chosen label, not for the positive class. In particular, since this is a binary classification task, the score cannot be lower than 0.5 (or else the model would have picked the other label).

BIAS AND FAIRNESS

If you try classifying "I am from the USA" and "I am from Iraq", you will see that the former is

classified as very positive, while the latter is classified as very negative. The model is also positive about Thailand but negative about Vietnam. You can try this model with your own country or city; the result may surprise you. Such a bias generally comes in large part from the training data itself: in this case, there were plenty of references to the wars in Iraq and Vietnam in the model's training data, creating a negative bias. This bias was then amplified during the fine-tuning process since the model was forced to choose between just two classes: positive or negative. If you use a model that was fine-tuned on a dataset with an extra neutral class, then the country bias mostly disappears.

The training data is not the only source of bias: the model's architecture, the type of loss or regularization used for training, the optimizer—all of these can affect what the model ends up learning. Understanding bias in AI and mitigating its negative effects is still an area of active research, but in any case you should probably pause and think before you rush to deploy a model to production. For example, if you train a model to score resumes, you must ensure that it's fair. So make sure you evaluate the model's performance not just on average over the whole test set, but across various subsets as well; for example, you may find that although the model works very well on average, its performance is abysmal for some categories of people. You may also want to run counterfactual tests; for example, check that the model's predictions do not change when you simply switch someone's gender or place of birth. The solution depends on the problem: it may require rebalancing the dataset, fine-tuning on a different dataset, switching to another pretrained model, tweaking the model's architecture or hyperparameters, etc.

Lastly, even if you manage to train a perfectly fair model, it could be *used* in a biased way. For example, the recruiters may use it only for some category of people and not others.

The model we chose is well-suited for general purpose sentiment analysis, such as movie reviews, but other models are better suited for specific use cases, such as social media posts (e.g., trained on a large dataset of tweets, then fine-tuned on a sentiment analysis dataset). To find the best model for your use case, you can search the list of available models on the *Hugging Face Hub*. However, there are over 80,000 models available in the "text-classification" category alone, so you will need to use the filters to narrow down the options. In particular, start by filtering on the task, and sort by trending or most liked models. You can also continue to filter by language and dataset, if necessary. Prefer models from reputable sources (e.g., models from users huggingface, facebook, google, cardiffnlp, and so on), and if the model includes executable code, make absolutely sure you trust the user (and if you do, set `trust_remote_code=True` when calling the `pipeline()` function).

There are many text classification tasks other than sentiment analysis. For example, a model fine-tuned on the Multi-Genre Natural Language Inference (MultiNLI) dataset can classify a pair of texts (each ending with a separation token [SEP]) into three classes: contradiction (if the texts contradict each other), entailment (if the first text entails the second), or neutral otherwise. For example:

```
>>> model_name = "huggingface/distilbert-base-uncased-finetuned-mnli"
>>> classifier_mnli = pipeline("text-classification", model=model_name)
>>> classifier_mnli([
...     "She loves me. [SEP] She loves me not. [SEP]",
...     "Alice just woke up. [SEP] Alice is awake. [SEP]",
...     "I like dogs. [SEP] Everyone likes dogs. [SEP]"])
...
[{'label': 'contradiction', 'score': 0.9717152714729309},
 {'label': 'entailment', 'score': 0.9119168519973755},
 {'label': 'neutral', 'score': 0.9509280323982239}]
```

Many other NLP tasks are also available via the pipeline API, such as question answering, summarization, sentence similarity, text generation, token classification, translation, and more. And it doesn't stop there! There are also many computer vision tasks, such as image classification, image segmentation, object detection, image-to-text, text-to-image, depth estimation, and even audio tasks, such as audio classification, speech-to-text, text-to-speech, and so on. Make sure to check out the full list at *https://huggingface.co/tasks*.

WARNING

Before you download a model, make sure you trust the hosting platform (e.g., the Hugging Face Hub) and the model's author: the model may contain executable code, which could be malicious. It could also produce biased outputs, or it may have been trained with copyrighted or sensitive private data which might be leaked to your users, or it might even have *poisoned weights* which could make it produce harmful content (e.g., propaganda) only for some types of inputs, otherwise behaving normally.

Time to step back. So far we have looked at text generation using a char-RNN, and sentiment analysis using various subword tokenization methods, pretrained embeddings, and even entire pretrained models. Along the way, we discussed embeddings, tokenizers, and the Hugging Face libraries. In the next section, we will explore another important NLP task: *neural machine translation* (NMT). Specifically, we will build an encoder-decoder model capable of translating English to Spanish, and we will see how to boost its performance using beam search and attention mechanisms. ¡Vamos!

An Encoder-Decoder Network for Neural Machine Translation

Let's begin with a relatively simple sequence-to-sequence NMT model[15] that will translate English text to Spanish (see Figure 14-5).

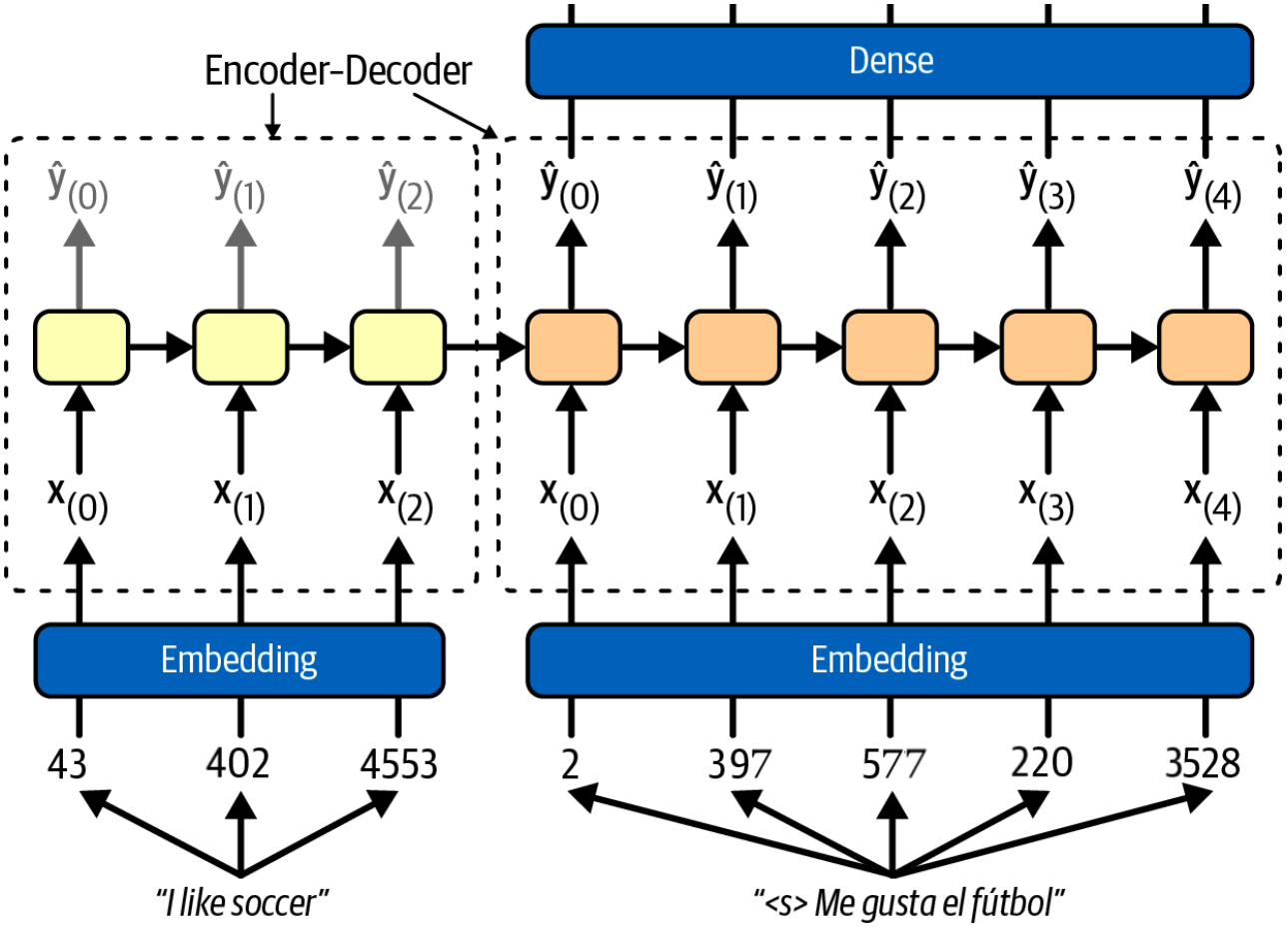


Figure 14-5. A simple machine translation model

In short, the architecture is as follows: English texts are fed as inputs to the encoder, and the decoder outputs the Spanish translations. Note that the Spanish translations are also used as inputs to the decoder during training, but shifted back by one step. In other words, during training the decoder is given as input the token that it *should* have output at the previous step, regardless of what it actually output. This is called *teacher forcing*—a technique that significantly speeds up training and improves the model's performance. For the very first token, the decoder is given the start-of-sequence (SoS, a.k.a. beginning-of-sequence, BoS) token (`"<s>"`), and the decoder is expected to end the text with an end-of-sequence (EoS) token (`"</s>"`).

Each token is initially represented by its ID (e.g., `4553` for the token "soccer"). Next, an `nn.Embedding` layer returns the token embedding. These token embeddings are then fed to the encoder and the decoder.

At each step, the decoder's dense output layer (i.e., a `nn.Linear` layer) outputs a logit score for each token in the output vocabulary (i.e., Spanish). If you pass these logits through the softmax function, you get an estimated probability for each possible token. For example, at the first step the word "Me" may have a probability of 7%, "Yo" may have a probability of 1%, and so on. This is very much like a regular classification task, and indeed we will train the model using the `nn.CrossEntropyLoss`, much like we did in the char-RNN model.

Note that at inference time (after training), you will not have the target text to feed to the decoder. Instead, you need to feed it the word that it has just output at the previous step, as shown in Figure 14-6 (this will require an embedding lookup that is not shown in the diagram).

TIP

In a 2015 paper,[16] Samy Bengio et al. proposed gradually switching from feeding the decoder the previous *target* token to feeding it the previous *output* token during training.

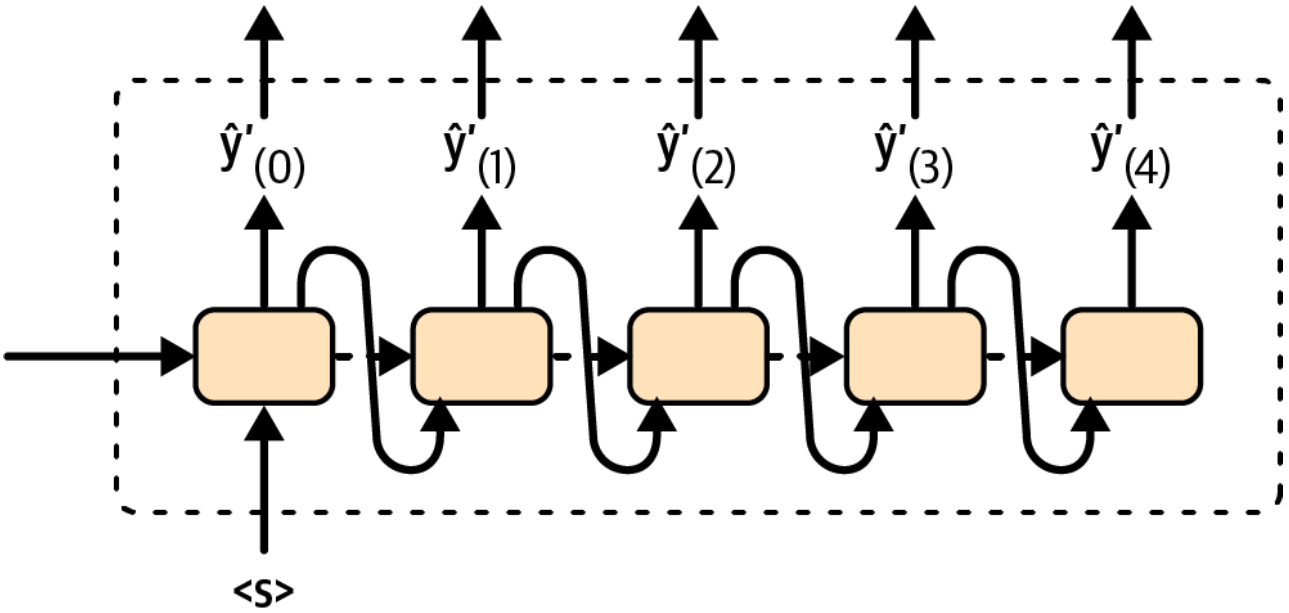


Figure 14-6. At inference time, the decoder is fed as input the word it just output at the previous time step

Let's build and train this model! First, we need to download a dataset of English/Spanish text pairs. For this, we will use the Datasets library to download English/Spanish pairs from the *Tatoeba Machine Translation* dataset, a huge dataset of text pairs from many languages, created by contributors of the Tatoeba project, led by the University of Helsinki. The original training set is so large that we will just use the validation set as our training set, setting aside 20% for validation:

```
nmt_dataset = load_dataset("Helsinki-NLP/tatoeba_mt", language_pair="eng-spa",
                           trust_remote_code=True)  # only if you trust the user
split = nmt_dataset["validation"].train_test_split(train_size=0.8, seed=42)
nmt_train_set, nmt_valid_set = split["train"], split["test"]
nmt_test_set = nmt_dataset["test"]
```

WARNING

The dataset contains a preprocessing script, which is why we have to set `trust_remote_code=True`: we trust that the University of Helsinki did not include malicious code in this script. Only use this option when you trust the source.

Each sample in the dataset is a dictionary containing an English text along with its Spanish translation. For example:

```
>>> nmt_train_set[0]
{'sourceLang': 'eng',
 'targetlang': 'spa',
```

```
 'sourceString': "Turn the light off. I can't fall asleep.",
 'targetString': 'Apaga la luz. No me puedo dormir.'}
```

We will need to tokenize this text. We could use a different tokenizer for English and Spanish, but these two languages have many words in common (e.g., animal, color, hotel, hospital, idea, radio, motor), and many similar subwords (e.g., pre, auto, inter, uni), so it makes sense to use a common tokenizer. Let's train a BPE tokenizer on all the training text, both English and Spanish:

```
def train_eng_spa():  # a generator function to iterate over all training text
    for pair in nmt_train_set:
        yield pair["sourceString"]
        yield pair["targetString"]

max_length = 500
vocab_size = 10_000
nmt_tokenizer_model = tokenizers.models.BPE(unk_token="<unk>")
nmt_tokenizer = tokenizers.Tokenizer(nmt_tokenizer_model)
nmt_tokenizer.enable_padding(pad_id=0, pad_token="<pad>")
nmt_tokenizer.enable_truncation(max_length=max_length)
nmt_tokenizer.pre_tokenizer = tokenizers.pre_tokenizers.Whitespace()
nmt_tokenizer_trainer = tokenizers.trainers.BpeTrainer(
    vocab_size=vocab_size, special_tokens=["<pad>", "<unk>", "<s>", "</s>"])
nmt_tokenizer.train_from_iterator(train_eng_spa(), nmt_tokenizer_trainer)
```

Let's test this tokenizer:

```
>>> nmt_tokenizer.encode("I like soccer").ids
[43, 403, 4300]
>>> nmt_tokenizer.encode("<s> Me gusta el fútbol").ids
[2, 397, 583, 221, 3325]
```

Perfect! Now let's create a small utility class that will hold tokenized English texts (i.e., the *source* token ID sequences), along with the corresponding tokenized Spanish targets (i.e., the *target* token ID sequences), plus the corresponding attention masks. For this, we can create a `namedtuple` base class (i.e., a tuple with named fields), and extend it to add a `to()` method, which will make it easy to move all these tensors to the GPU:

```
from collections import namedtuple

fields = ["src_token_ids", "src_mask", "tgt_token_ids", "tgt_mask"]
class NmtPair(namedtuple("NmtPairBase", fields)):
    def to(self, device):
        return NmtPair(self.src_token_ids.to(device), self.src_mask.to(device),
                       self.tgt_token_ids.to(device), self.tgt_mask.to(device))
```

Next, let's create the data loaders:

```
def nmt_collate_fn(batch):
    src_texts = [pair['sourceString'] for pair in batch]
    tgt_texts = [f"<s> {pair['targetString']} </s>" for pair in batch]
    src_encodings = nmt_tokenizer.encode_batch(src_texts)
    tgt_encodings = nmt_tokenizer.encode_batch(tgt_texts)
    src_token_ids = torch.tensor([enc.ids for enc in src_encodings])
    tgt_token_ids = torch.tensor([enc.ids for enc in tgt_encodings])
    src_mask = torch.tensor([enc.attention_mask for enc in src_encodings])
    tgt_mask = torch.tensor([enc.attention_mask for enc in tgt_encodings])
    inputs = NmtPair(src_token_ids, src_mask,
                     tgt_token_ids[:, :-1], tgt_mask[:, :-1])
    labels = tgt_token_ids[:, 1:]
    return inputs, labels

batch_size = 256
nmt_train_loader = DataLoader(nmt_train_set, batch_size=batch_size,
                              collate_fn=nmt_collate_fn, shuffle=True)
nmt_valid_loader = DataLoader(nmt_valid_set, batch_size=batch_size,
                              collate_fn=nmt_collate_fn)
nmt_test_loader = DataLoader(nmt_test_set, batch_size=batch_size,
                             collate_fn=nmt_collate_fn)
```

The `nmt_collate_fn()` function starts by extracting all the English and Spanish texts from the given batch. In the process, it also adds an SoS token at the start of each Spanish text, as well as an EoS token at the end. It then tokenizes both the English and Spanish texts using our BPE tokenizer. Next, the input sequences and the attention masks are converted to tensors and wrapped in an `NmtPair`. Importantly, the function drops the EoS token from the decoder inputs, and drops the SoS token from the decoder targets. For example, the inputs may contain the token IDs for "<s> Me gusta el fútbol", while the targets may contain the token IDs for "Me gusta el fútbol </s>". Lastly, the function returns the inputs (i.e., the `NmtPair`) along with the targets. Then we just create the data loaders as usual.

And now we are ready to build our translation model. It's just like Figure 14-5, except the encoder and decoder share the same `nn.Embedding` layer, and the encoder and decoder `nn.GRU` modules contain two layers each:

```
class NmtModel(nn.Module):
    def __init__(self, vocab_size, embed_dim=512, pad_id=0, hidden_dim=512,
                 n_layers=2):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim, padding_idx=pad_id)
        self.encoder = nn.GRU(embed_dim, hidden_dim, num_layers=n_layers,
                              batch_first=True)
        self.decoder = nn.GRU(embed_dim, hidden_dim, num_layers=n_layers,
                              batch_first=True)
        self.output = nn.Linear(hidden_dim, vocab_size)
```

```
    def forward(self, pair):
        src_embeddings = self.embed(pair.src_token_ids)
        tgt_embeddings = self.embed(pair.tgt_token_ids)
        src_lengths = pair.src_mask.sum(dim=1)
        src_packed = pack_padded_sequence(
            src_embeddings, lengths=src_lengths.cpu(),
            batch_first=True, enforce_sorted=False)
        _, hidden_states = self.encoder(src_packed)
        outputs, _ = self.decoder(tgt_embeddings, hidden_states)
        return self.output(outputs).permute(0, 2, 1)

torch.manual_seed(42)
vocab_size = nmt_tokenizer.get_vocab_size()
nmt_model = NmtModel(vocab_size).to(device)
```

Almost everything in this model should look familiar: it's very similar to our previous models. We create the modules in the constructor, then the `forward()` method embeds the input sequences (both English and Spanish), it packs the English embeddings and passes them through the encoder, then it passes the Spanish embeddings to the decoder, along with the encoder's last hidden states (across all `nn.GRU` layers). Lastly, the decoder's outputs are passed through the output `nn.Linear` layer, and the final outputs are permuted to ensure that the class dimension (containing the token logits) is the second dimension, since this is expected by the `nn.CrossEntropyLoss` and the `Accuracy` metric, as we saw earlier.

NOTE

The most common metric used in NMT is the *bilingual evaluation understudy* (BLEU) score, which compares each translation produced by the model with several good translations produced by humans. It counts the number of *n*-grams (sequences of *n* words) that appear in any of the target translations and adjusts the score to take into account the frequency of the produced *n*-grams in the target translations. It is implemented by TorchMetric's `BLEUScore` class.

We could have packed the Spanish embeddings, but then the decoder's outputs would have been packed sequences, which we would have had to pad before we passed them to the output layer. We avoided this complexity because we can just configure the loss to ignore the output tokens when the targets are padding tokens, like this:

```
xentropy = nn.CrossEntropyLoss(ignore_index=0)  # ignore <pad> tokens
```

Now you can train this model, and it will take quite a while. It's actually not that long when you consider the fact that the model is learning two languages at once!

While it's training, let's write a little helper function to translate some English text to Spanish using our model. It will start by calling the model with the English text for the encoder, and a single SoS token for the decoder. The decoder will just output logits for the first token in the translation. Our function will then pick the most likely token (i.e., with the highest logit) and

translation. Our function will then pick the most likely token (i.e., with the highest logit) and add it to the decoder inputs, then it will call the model again to get the next token. It will repeat this process, adding one token at a time, until the model outputs an EoS token:

```
def translate(model, src_text, max_length=20, pad_id=0, eos_id=3):
    tgt_text = ""
    token_ids = []
    for index in range(max_length):
        batch, _ = nmt_collate_fn([{"sourceString": src_text,
                                    "targetString": tgt_text}])
        with torch.no_grad():
            Y_logits = model(batch.to(device))
            Y_token_ids = Y_logits.argmax(dim=1)  # find the best token IDs
            next_token_id = Y_token_ids[0, index]  # take the last token ID

        next_token = nmt_tokenizer.id_to_token(next_token_id)
        tgt_text += " " + next_token
        if next_token_id == eos_id:
            break
    return tgt_text
```

NOTE

This implementation works but it's not optimized at all. We could run the encoder just once on the English text, and we could also run the decoder just once per time step, instead of running it over the whole growing text at each iteration.

Let's try translating some text!

```
>>> nmt_model.eval()
>>> translate(nmt_model, "I like soccer.")
' Me gusta el fútbol . </s>'
```

Hurray, it works! We just built a model from scratch that can translate English to Spanish.

MODEL OPTIMIZATIONS

When the output vocabulary is large (e.g., 10,000 tokens or more), computing the `nn.CrossEntropyLoss` can be quite slow, depending on the hardware. To speed things up, one technique is to use *sampled softmax*, introduced in 2015 by Sébastien Jean et al.[17] Instead of computing the softmax over all of the logits, it computes an approximation based on the correct class's logit, as well as a random sample of logits for other classes. This technique requires knowing the target, so it is only useful during training. Moreover, it is not included in PyTorch, so you have to implement it yourself.

Another technique is *adaptive softmax*, introduced in 2016 by Edouard Grave et al.,[18] which speeds up softmax computation by splitting the vocabulary into frequency-based clusters

speeds up softmax computation by splitting the vocabulary into frequency based clusters. Frequent tokens are processed normally, while less frequent tokens are placed in progressively larger clusters, reducing computation by only accessing the necessary clusters. This speeds up computations both during training and inference. PyTorch implements this algorithm in the `nn.AdaptiveLogSoftmaxWithLoss` class.

Another thing you can do to speed up training (and also save a lot of memory) is to use the embedding matrix as the weights of the output layer. This is called *tying the weights* of these two layers, and it was first proposed in a 2016 paper by Ofir Press and Lior Wolf.[19] To implement it, just add `self.output.weight = self.embed.weight` to the model's constructor. You can also get rid of the output layer's `bias` parameter, by setting `bias=False` when creating the output layer. Tying the weights significantly reduces the number of model parameters, which speeds up training and may sometimes improve the model's accuracy as well, especially if you don't have a lot of training data. Why does this work? Well, as we saw earlier, the embedding matrix is equivalent to one-hot encoding followed by a linear layer with no bias term and no activation function that maps the one-hot vectors to the embedding space. The output layer does the reverse. So if the model can find an embedding matrix whose transpose is equal to its inverse (such a matrix is called an *orthogonal matrix*), then there's no need to learn a separate set of weights for the output layer.

If you play around with our translation model, you will find that it often works reasonably well on short text, but it really struggles with longer sentences. For example:

```
>>> longer_text = "I like to play soccer with my friends."
>>> translate(nmt_model, longer_text)
' Me gusta jugar con mis amigos . </s>'
```

The translation says "I like to play with my friends". Oops, there's no mention of soccer. So how can we improve this model? One way is to increase the training set size and add more `nn.GRU` layers in both the encoder and the decoder. You could also make the encoder bidirectional (but not the decoder, or else it would no longer be causal and it would see the full translation at each time step, instead of just the previous tokens). Another popular technique that can greatly improve the performance of a translation model at inference time is *beam search*.

Beam Search

To translate an English text to Spanish, we call our model several times, producing one word at a time. Unfortunately, this means that when the model makes one mistake, it is stuck with it for the rest of the translation, which can cause more errors, making the translation worse and worse. For example, suppose we want to translate "I like soccer", and the model correctly starts with "Me", but then predicts "gustan" (plural) instead of "gusta" (singular). This mistake is understandable, since "Me gustan" is the correct way to start translating "I like" in many cases. Unfortunately, once the model has made this mistake, it is stuck with "gustan". It then reasonably adds "los", which is the plural for "the". But since the model never saw "los fútbol"

in the training data (soccer is singular, not plural), the model tries to find something reasonable to add, and given the context it adds “jugadores”, which means “the players”. So “I like soccer” gets translated to “I like the players”: one error caused a chain of errors.

How can we give the model a chance to go back and fix mistakes it made earlier? One of the most common solutions is *beam search*: it keeps track of a short list of the *k* most promising output sequences (say, the top three), and at each decoder step it tries to extend each of them by one word, keeping only the *k* most likely sequences. The parameter *k* is called the *beam width*.

For example, suppose you use the model to translate the sentence “I like soccer” using beam search with a beam width of three (see Figure 14-7). At the first decoder step, the model will output an estimated probability for each possible first word in the translated sentence. Suppose the top three words are “Me” (75% estimated probability), “a” (3%), and “como” (1%). That’s our short list so far. Next, we use the model to find the next word for each sentence. For the first sentence (“Me”), perhaps the model outputs a probability of 36% for the word “gustan”, 32% for the word “gusta”, 16% for the word “encanta”, and so on. Note that these are actually *conditional* probabilities, given that the sentence starts with “Me”. For the second sentence (“a”), the model might output a conditional probability of 50% for the word “mi”, and so on. Assuming the vocabulary has 10,000 tokens, we will end up with 10,000 probabilities per sentence.

Next, we compute the probabilities of each of the 30,000 two-token sentences we considered (3 × 10,000). We do this by multiplying the estimated conditional probability of each word by the estimated probability of the sentence it completes. For example, the estimated probability of the sentence “Me” was 75%, while the estimated conditional probability of the word “gustan” (given that the first word is “Me”) was 36%, so the estimated probability of the sentence “Me gustan” is 75% × 36% = 27%. After computing the probabilities of all 30,000 two-word sentences, we keep only the top 3. In this example they all start with the word “Me”: “Me gustan” (27%), “Me gusta” (24%), and “Me encanta” (12%). Right now, the sentence “Me gustan” is winning, but “Me gusta” has not been eliminated.

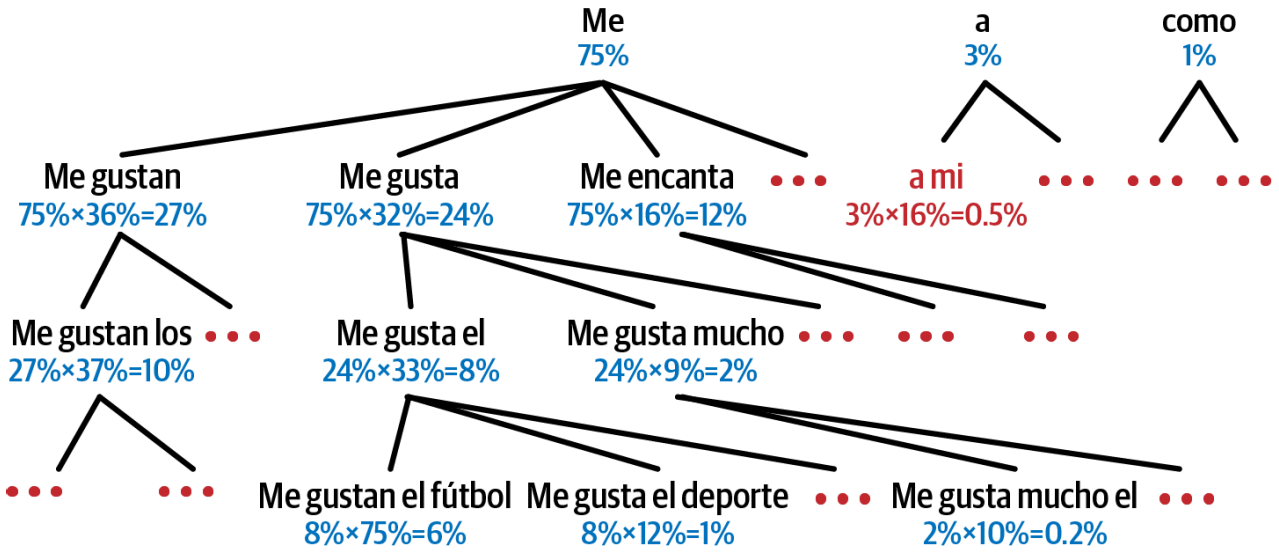


Figure 14-7. Beam search, with a beam width of three

Then we repeat the same process: we use the model to predict the next word in each of these three sentences, and we compute the probabilities of all 30,000 three-word sentences we considered. Perhaps the top 3 are now "Me gustan los" (10%), "Me gusta el" (8%), and "Me gusta mucho" (2%). At the next step we may get "Me gusta el fútbol" (6%), "Me gusta mucho el" (1%), and "Me gusta el deporte" (0.2%). Notice that "Me gustan" was eliminated, and the correct translation is now ahead. We boosted our encoder-decoder model's performance without any extra training, simply by using it more wisely.

The notebook for this chapter contains a very simple `beam_search()` function, if you're interested, but in general you will probably want to use the implementation provided by the `GenerationMixin` class in the Transformers library. This is where the text generation models from the Transformers library get their `generate()` method: it accepts a `num_beams` argument which you can set to the desired beam width if you want to use beam search. It also provides a `do_sample` argument that will randomly sample the next token using the probability distribution output by the model, just like we did earlier with our char-RNN model. Other generation strategies are also supported and can be combined (see *https://homl.info/hfgen* for more details).

With all this, you can get reasonably good translations for fairly short sentences. For example, the following translation is correct:

```
>>> beam_search(nmt_model, longer_text, beam_width=3)
' Me gusta jugar al fútbol con mis amigos . </s>'
```

Unfortunately, this model will still be pretty bad at translating long sentences:

```
>>> longest_text = "I like to play soccer with my friends at the beach."
>>> beam_search(nmt_model, longest_text, beam_width=3)
' Me gusta jugar con jugar con los jug adores de la playa . </s>'
```

This translates to "I like to play with play with the players on the beach". That's not quite right. Once again, the problem comes from the limited short-term memory of RNNs. *Attention mechanisms* are the game-changing innovation that addressed this problem.

Attention Mechanisms

Consider the path from the word "soccer" to its translation "fútbol" back in Figure 14-5: it is quite long! This means that a representation of this word (along with all the other words) needs to be carried over many steps before it is actually used. Can't we make this path shorter?

This was the core idea in a landmark 2014 paper[20] by Dzmitry Bahdanau et al., where the authors introduced a technique that allowed the decoder to focus on the appropriate words (as encoded by the encoder) at each time step. For example, at the time step where the decoder needs to output the word "fútbol", it will focus its attention on the word "soccer". This means

that the path from an input word to its translation is now much shorter, so the short-term memory limitations of RNNs have much less impact. Attention mechanisms revolutionized neural machine translation (and deep learning in general), allowing a significant improvement in the state of the art, especially for long sentences (e.g., over 30 words).

Figure 14-8 shows our encoder-decoder model with an added attention mechanism:

- On the left, you have the encoder and the decoder (I've made the encoder bidirectional in this figure, as it's generally a good idea).
- Instead of just sending the encoder's final hidden state to the decoder, as well as the previous target word at each time step (which is still done, although it is not shown in the figure), we now send all of the encoder's outputs to the decoder as well.
- Since the decoder cannot deal with all these encoder outputs at once, they need to be aggregated: at each time step, the decoder's memory cell computes a weighted sum of all the encoder outputs. This determines which words the decoder will focus on at this step.
- The weight $\alpha_{(t,i)}$ is the weight of the i^{th} encoder output at the t^{th} decoder time step. For example, if the weight $\alpha_{(3,2)}$ is much larger than the weights $\alpha_{(3,0)}$ and $\alpha_{(3,1)}$, then the decoder will pay much more attention to the encoder's output for word #2 ("soccer") than to the other two outputs, at least at this time step.
- The rest of the decoder works just like earlier: at each time step the memory cell receives the inputs we just discussed, plus the hidden state from the previous time step, and finally (although it is not represented in the diagram) it receives the target word from the previous time step (or at inference time, the output from the previous time step).

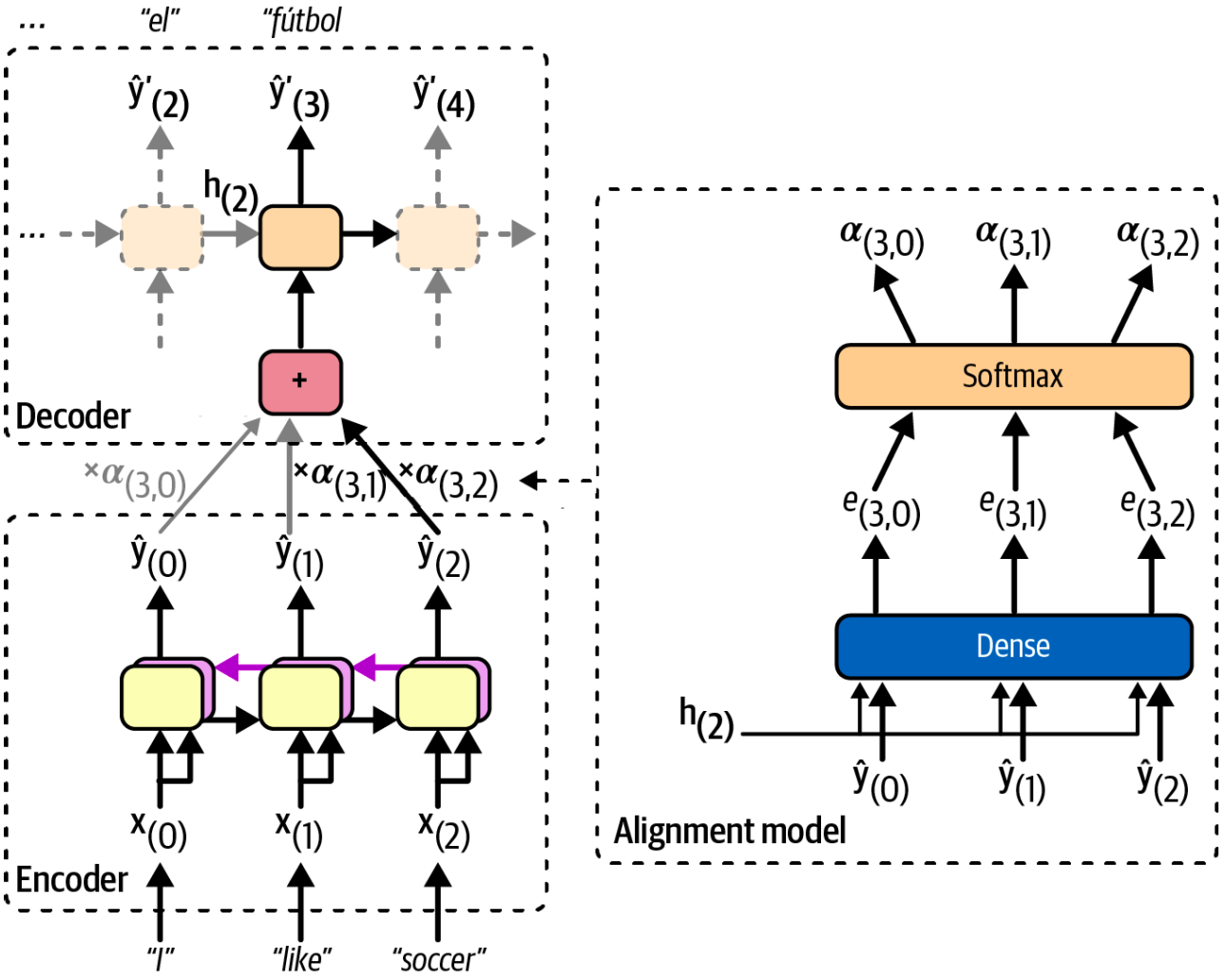


Figure 14-8. Neural machine translation using an encoder-decoder network with an attention model

But where do these $\alpha_{(t,i)}$ weights come from? Well, they are generated by a small neural network called an *alignment model* (or an *attention layer*), which is trained jointly with the rest of the encoder-decoder model. This alignment model is illustrated on the righthand side of Figure 14-8:

- It starts with a dense layer (i.e., `nn.Linear`) that takes as input each of the encoder's outputs, along with the decoder's previous hidden state (e.g., $\mathbf{h}_{(2)}$), and outputs a score (or energy) for each encoder output (e.g., $e_{(3,\ 2)}$). This score measures how well each encoder output is aligned with the decoder's previous hidden state.
  For example, in Figure 14-8, the model has already output "me gusta el" (meaning "I like"), so it's now expecting a noun. The word "soccer" is the one that best aligns with the current state, so it gets a high score.
- Finally, all the scores go through a softmax layer to get a final weight for each encoder output (e.g., $\alpha_{(3,2)}$). All the weights for a given decoder time step add up to 1.

This particular attention mechanism is called *Bahdanau attention* (named after the 2014 paper's first author). Since it concatenates the encoder output with the decoder's previous hidden state, it is sometimes called *concatenative attention* (or *additive attention*).

In short, the attention mechanism provides a way to focus the attention of the model on part of the inputs. That said, there's another way to think of this whole process: it acts as a differentiable memory retrieval mechanism. For example, let's suppose the encoder analyzed the input sentence "I like soccer", and it managed to understand that the word "I" is the subject, the word "like" is the verb, and the word "soccer" is the noun, so it encoded this information in its outputs for these words. Now suppose the decoder has already translated "I like", and it thinks that it should translate the noun next. For this, it needs to fetch the noun from the input sentence. This is analogous to a dictionary lookup: it's as if the encoder had created a dictionary {"subject": "I", "verb": "like", "noun": "soccer"} and the decoder wanted to look up the value that corresponds to the key "noun".

However, the model does not have discrete tokens to represent the keys (like "subject", "verb", or "noun"); instead, it has vectorized representations of these concepts that it learned during training, so the query it will use for the lookup will not perfectly match any key in the dictionary. One solution is to compute a similarity measure between the query and each key in the dictionary, and then use the softmax function to convert these similarity scores to weights that add up to 1. As we just saw, that's exactly what the attention layer does. If the key that represents the noun is by far the most similar to the query, then that key's weight will be close to 1. Next, the attention layer computes a weighted sum of the corresponding values: if the weight of the "noun" key is close to 1, then the weighted sum will be very close to the representation of the word "soccer". In short, the decoder queried for a noun and the attention mechanism retrieved it.

In most modern implementations of attention mechanisms, the arguments are named `query`, `key`, and `value`. In our example, the query is the decoder's hidden states, the key is the en-

coder's outputs (this is used to compute the weights), and the value is also the encoder's outputs (this is used to compute the final weighted sum).

NOTE

If the input sentence is n words long, and assuming the output sentence is about as long, then the attention mechanism will need to compute about n^2 weights. This quadratic computational complexity becomes untractable when the sentences are too long.

Another common attention mechanism, known as *Luong attention* or *multiplicative attention*, was proposed shortly after, in 2015,[21] by Minh-Thang Luong et al. Because the goal of the alignment model is to measure the similarity between one of the encoder's outputs and the decoder's previous hidden state, the authors proposed to simply compute the dot product (see Chapter 4) of these two vectors, as this is often a fairly good similarity measure, and modern hardware can compute it very efficiently. For this to be possible, both vectors must have the same dimensionality. The dot product gives a score, and all the scores (at a given decoder time step) go through a softmax layer to give the final weights, just like in Bahdanau attention.

Luong et al. also proposed to use the decoder's hidden state at the current time step rather than at the previous time step (i.e., $\mathbf{h}_{(t)}$ rather than $\mathbf{h}_{(t-1)}$) to compute the attention vector (noted). This attention vector is then concatenated with the decoder's hidden state to form an attentional hidden state, which is then used to predict the next token. This simplifies and speeds up the process by allowing the encoder and decoder to operate independently before attention is applied, rather than interweaving attention into the decoder's recurrence.

The researchers also proposed a variant of the dot product mechanism where the encoder outputs first go through a fully connected layer (without a bias term) before the dot products are computed. This is called the "general" dot product approach. The researchers compared both dot product approaches with the concatenative attention mechanism (adding a rescaling parameter vector $\mathbf{v}$), and they observed that the dot product variants performed better than concatenative attention. For this reason, concatenative attention is much less used now. The equations for these three attention mechanisms are summarized in Equation 14-2.

Equation 14-2. Attention mechanisms

Let's add Luong attention to our encoder–decoder model. Since PyTorch does not include a Luong attention function, we need to write our own. Luckily, it's pretty short:

```
def attention(query, key, value):
    scores = query @ key.transpose(1, 2)  # [B, Lq, D] @ [B, D, Lk]= [B, Lq, Lk]
    weights = torch.softmax(scores, dim=-1)  # [B, Lq, Lk]
    return weights @ value  # [B, Lq, Lk] @ [B, Lk, Dv] = [B, Lq, Dv]
```

Just like in Equation 14-2, we first compute the attention scores, then we convert them to attention weights using the softmax function, and lastly we compute the attention output by multiplying the attention weights with the value (i.e., the encoder outputs). This implementation efficiently runs all these computations for the whole batch at once. The `query` argument corresponds to $\mathbf{h}_{(t)}$ in Equation 14-2 (i.e., the decoder's hidden states), and the `key` argument corresponds to (i.e., the encoder's outputs), but only for the computation of the attention scores. The `value` argument also corresponds to , but only for the final computation of the weighted sum. The `key` and `value` arguments are generally identical, but there are a few scenarios where they can differ (e.g., some models use compressed keys to save memory and speed up the score computation). The shapes are shown in the comments: `B` is the batch size; `Lq` is the length of the longest query in the batch; `Lk` is the length of the longest key in the batch (note that each value must have the same length as its corresponding key); `D` is the query's embedding size, which must be the same as the key's embedding size; and `Dv` is the value's embedding size.

TIP

Since all arguments are 3D tensors, we could replace the `@` matrix multiplication operator with the *batch matrix multiplication* function: `torch.bmm()`. This function only works with batches of matrices (i.e., 3D tensors), but it's optimized for this use case so it runs faster. The result is the same: each matrix in the first tensor gets multiplied by the corresponding matrix in the second tensor.

Now let's update our NMT model. The constructor needs just one modification: the output layer's input size must be doubled, since we will concatenate the attention vectors to the decoder outputs:

```
self.output = nn.Linear(2 * hidden_dim, vocab_size)
```

Next, let's add attention to the `forward()` method:

```
def forward(self, pair):
    src_embeddings = self.embed(pair.src_token_ids)  # same as earlier
    tgt_embeddings = self.embed(pair.tgt_token_ids)  # same
    src_lengths = pair.src_mask.sum(dim=1)  # same
    src_packed = pack_padded_sequence(src_embeddings, [...])  # same
    encoder_outputs_packed, hidden_states = self.encoder(src_packed)
    decoder_outputs, _ = self.decoder(tgt_embeddings, hidden_states)  # same
    encoder_outputs, _ = pad_packed_sequence(encoder_outputs_packed,
                                             batch_first=True)
    attn_output = attention(query=decoder_outputs,
                            key=encoder_outputs, value=encoder_outputs)
    combined_output = torch.cat((attn_output, decoder_outputs), dim=-1)
    return self.output(combined_output).permute(0, 2, 1)
```

Let's go through this code:

- We compute the English and Spanish embeddings, the English sequence lengths, and we pack the English embeddings, just like earlier.
- We then run the encoder like earlier, but we no longer ignore its outputs since we will need them for the attention function.
- Next, we run the decoder, just like earlier.
- Since the encoder's inputs are represented as a packed sequence, its outputs are also represented as a packed sequence. Not many operations support packed sequences, so we must convert the encoder's outputs to a padded tensor using the `pad_packed_sequence()` function.
- And now we can call our `attention()` function. Note that we pass the decoder outputs instead of the hidden states because the decoder only returns the last hidden states. That's OK because the `nn.GRU` layer's outputs are equal to its top-layer hidden states.
- Lastly, we concatenate the attention output and the decoder outputs along the last dimension, and we pass the result through the output layer. As earlier, we also permute the last two dimensions of the result.

WARNING

Our attention mechanism doesn't ignore padding tokens. The model learns to ignore them during training, but it's preferable to mask them entirely. We will see how in Chapter 15.

And that's it! If you train this model, you will find that it now handles much longer sentences. For example:

```
>>> translate(nmt_attn_model, longest_text)
' Me gusta jugar fu tbol con mis amigos en la playa . </s>'
```

Perfect! We didn't even have to use beam search. In fact, attention mechanisms turned out to be so powerful that some Google researchers tried getting rid of recurrent layers altogether, only using feedforward layers and attention. Surprisingly, it worked like a charm. This led the researchers to name their paper "Attention is all you need", introducing the Transformer architecture to the world. This was the start of a huge revolution in NLP and beyond. In the next chapter, we will explore the Transformer architecture and see how it revolutionized deep learning.

Exercises

1. What are the pros and cons of using a stateful RNN versus a stateless RNN?
2. Why do people use encoder-decoder RNNs rather than plain sequence-to-sequence RNNs for automatic translation?

3. How can you deal with variable-length input sequences? What about variable-length output sequences?
4. What is beam search, and why would you use it? What tool can you use to implement it?
5. What is an attention mechanism? How does it help?
6. When would you need to use sampled softmax?
7. *Embedded Reber grammars* were used by Hochreiter and Schmidhuber in their paper about LSTMs. They are artificial grammars that produce strings such as "BPBTSXXVPSEPE". Check out Jenny Orr's nice introduction to this topic, then choose a particular embedded Reber grammar (such as the one represented on Orr's page), and train an RNN to identify whether a string respects that grammar or not. You will first need to write a function capable of generating a training batch containing about 50% strings that respect the grammar, and 50% that don't.
8. Train an encoder-decoder model that can convert a date string from one format to another (e.g., from "April 22, 2019" to "2019-04-22").

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab-p*.

[1] Alan Turing, "Computing Machinery and Intelligence", *Mind* 49 (1950): 433–460.

[2] Of course, the word *chatbot* came much later. Turing called his test the *imitation game*: machine A and human B chat with human interrogator C via text messages; the interrogator asks questions to figure out which one is the machine (A or B). The machine passes the test if it can fool the interrogator, while the human B must try to help the interrogator.

[3] Tomáš Mikolov et al., "Distributed Representations of Words and Phrases and Their Compositionality", *Proceedings of the 26th International Conference on Neural Information Processing Systems* 2 (2013): 3111–3119.

[4] Malvina Nissim et al., "Fair Is Better Than Sensational: Man Is to Doctor as Woman Is to Doctor", arXiv preprint arXiv:1905.09866 (2019).

[5] It's a convention in Python to name unused variables with an underscore prefix.

[6] Another technique to capture longer patterns is to use a stateful RNN. It's a bit more complex and not used as much, but if you're interested I've included a section in this chapter's notebook.

[7] Alec Radford et al., "Learning to Generate Reviews and Discovering Sentiment", arXiv preprint arXiv:1704.01444 (2017).

[8] Rico Sennrich et al., "Neural Machine Translation of Rare Words with Subword Units", *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics* 1 (2016): 1715–1725.

[9] Yonghui Wu et al., "Google's Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation", arXiv preprint arXiv:1609.08144 (2016).

[10] Taku Kudo, "Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates", arXiv preprint arXiv:1804.10959 (2018).

Subword Candidates", arXiv preprint arXiv:1804.10959 (2018).

[11] Taku Kudo and John Richardson, "SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing", arXiv preprint arXiv:1808.06226 (2018).

[12] *Nested tensors* serve a similar purpose and are more convenient to use, but they are still in prototype stage at the time of writing. See *https://pytorch.org/docs/stable/nested.html* for more details.

[13] Matthew Peters et al., "Deep Contextualized Word Representations", *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* 1 (2018): 2227–2237.

[14] Jeremy Howard and Sebastian Ruder, "Universal Language Model Fine-Tuning for Text Classification", *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics* 1 (2018): 328–339.

[15] Ilya Sutskever et al., "Sequence to Sequence Learning with Neural Networks", arXiv preprint, arXiv:1409.3215 (2014).

[16] Samy Bengio et al., "Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks", arXiv preprint arXiv:1506.03099 (2015).

[17] Sébastien Jean et al., "On Using Very Large Target Vocabulary for Neural Machine Translation", *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing* 1 (2015): 1–10.

[18] Edouard Grave et al., "Efficient softmax approximation for GPUs", arXiv preprint arXiv:1609.04309 (2016).

[19] Ofir Press, Lior Wolf, "Using the Output Embedding to Improve Language Models", arXiv preprint arXiv:1608.05859 (2016).

[20] Dzmitry Bahdanau et al., "Neural Machine Translation by Jointly Learning to Align and Translate", arXiv preprint arXiv:1409.0473 (2014).

[21] Minh-Thang Luong et al., "Effective Approaches to Attention-Based Neural Machine Translation", *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (2015): 1412–1421.