11 Deep learning for text

This chapter covers

- Preprocessing text data for machine learning applications
- Bag-of-words approaches and sequence-modeling approaches for text processing
- The Transformer architecture
- Sequence-to-sequence learning

11.1 Natural language processing: The bird's eye view

In computer science, we refer to human languages, like English or Mandarin, as "natural" languages, to distinguish them from languages that were designed for machines, like Assembly, LISP, or XML. Every machine language was *designed*: its starting point was a human engineer writing down a set of formal rules to describe what statements you could make in that language and what they meant. Rules came first, and people only started using the language once the rule set was complete. With human language, it's the reverse: usage comes first, rules arise later. Natural language was shaped by an evolution process, much like biological organisms—that's what makes it "natural." Its "rules," like the grammar of English, were formalized after the fact and are often ignored or broken by its users. As a result, while

machine-readable language is highly structured and rigorous, using precise syntactic rules to weave together exactly defined concepts from a fixed vocabulary, natural language is messy—ambiguous, chaotic, sprawling, and constantly in flux.

Creating algorithms that can make sense of natural language is a big deal: language, and in particular text, underpins most of our communications and our cultural production. The internet is mostly text. Language is how we store almost all of our knowledge. Our very thoughts are largely built upon language. However, the ability to understand natural language has long eluded machines. Some people once naively thought that you could simply write down the "rule set of English," much like one can write down the rule set of LISP. Early attempts to build natural language processing (NLP) systems were thus made through the lens of "applied linguistics." Engineers and linguists would handcraft complex sets of rules to perform basic machine translation or create simple chatbots—like the famous ELIZA program from the 1960s, which used pattern matching to sustain very basic conversation. But language is a rebellious thing: it's not easily pliable to formalization. After several decades of effort, the capabilities of these systems remained disappointing.

Handcrafted rules held out as the dominant approach well into the 1990s. But starting in the late 1980s, faster computers and greater data availability started making a better alternative viable. When you find yourself building systems that are big piles of ad hoc rules, as a clever engineer, you're likely to start asking: "Could I use a corpus of data to automate the process of finding these rules? Could I search for the rules within some kind of rule space, instead of having to come up with them myself?" And just like that, you've graduated to doing machine learning. And so, in the late 1980s, we started seeing machine learning approaches to natural language processing. The earliest ones were based on decision trees—the intent was literally to automate the development of the kind of if/then/else rules of previous systems. Then statistical approaches started gaining speed, starting with logistic regression. Over time, learned parametric models fully took over, and linguistics came to be seen as more of a hindrance than a useful tool. Frederick Jelinek, an early speech recognition researcher, joked in the 1990s: "Every time I fire a linguist, the performance of the speech recognizer goes up."

That's what modern NLP is about: using machine learning and large datasets to give computers the ability not to *understand* language, which is a more lofty goal, but to ingest a piece of language as input and return something useful, like predicting the following:

- "What's the topic of this text?" (text classification)
- "Does this text contain abuse?" (content filtering)
- "Does this text sound positive or negative?" (sentiment analysis)
- "What should be the next word in this incomplete sentence?" (language modeling)
- "How would you say this in German?" (translation)
- "How would you summarize this article in one paragraph?" (summarization)
- etc.

Of course, keep in mind throughout this chapter that the text-processing models you will train won't possess a human-like understanding of language; rather, they simply look for statistical regularities in their input data, which turns out to be sufficient to perform well on many simple tasks. In much the same way that computer vision is pattern recognition applied to pixels, NLP is pattern recognition applied to words, sentences, and paragraphs.

The toolset of NLP—decision trees, logistic regression—only saw slow evolution from the 1990s to the early 2010s. Most of the research focus was on feature engineering. When I won my first NLP competition on Kaggle in 2013, my model was, you guessed it, based on decision trees and logistic regression. However, around 2014–2015, things started changing at last. Multiple researchers began to investigate the language-understanding capabilities of recurrent neural networks, in particular LSTM—a sequence-processing algorithm from the late 1990s that had stayed under the radar until then.

In early 2015, Keras made available the first open source, easy-to-use implementation of LSTM, just at the start of a massive wave of renewed interest in recurrent neural networks—until then, there had only been "research code" that couldn't be readily reused. Then from 2015 to 2017, recurrent neural networks dominated the booming NLP scene. Bidirectional LSTM models, in particular, set the state of the art on many important tasks, from summarization to question-answering to machine translation.

Finally, around 2017–2018, a new architecture rose to replace RNNs: the Transformer, which you will learn about in the second half of this chapter. Transformers unlocked considerable progress across the field in a short period of time, and today most NLP systems are based on them.

Let's dive into the details. This chapter will take you from the very basics to doing machine translation with a Transformer.

11.2 Preparing text data

Deep learning models, being differentiable functions, can only process numeric tensors: they can't take raw text as input. *Vectorizing* text is the process of transforming text into numeric tensors. Text vectorization processes come in many shapes and forms, but they all follow the same template (see figure 11.1):

- First, you *standardize* the text to make it easier to process, such as by converting it to lowercase or removing punctuation.
- You split the text into units (called *tokens*), such as characters, words, or groups of words. This is called *tokenization*.
- You convert each such token into a numerical vector. This will usually involve first *indexing* all tokens present in the data.

Let's review each of these steps.

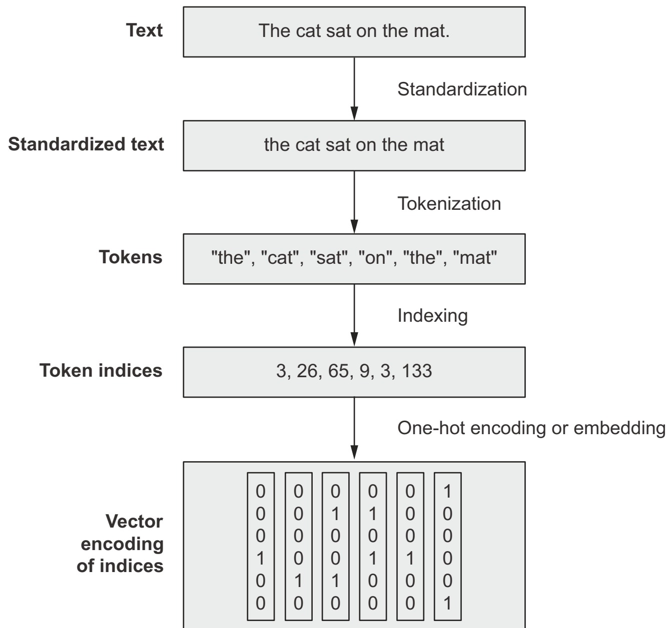


Figure 11.1 From raw text to vectors

11.2.1 Text standardization

Consider these two sentences:

- “sunset came. i was staring at the Mexico sky. Isnt nature splendid??”
- “Sunset came; I stared at the México sky. Isn’t nature splendid?”

They’re very similar—in fact, they’re almost identical. Yet, if you were to convert them to byte strings, they would end up with very different representations, because “i” and “I” are two different characters, “Mexico” and “México” are two different words, “isnt” isn’t “isn’t,” and so on. A machine learning model doesn’t know a priori that “i” and “I” are the same letter, that “é” is an “e” with an accent, or that “staring” and “stared” are two forms of the same verb.

Text standardization is a basic form of feature engineering that aims to erase encoding differences that you don’t want your model to have to deal with. It’s not exclusive to machine learning, either—you’d have to do the same thing if you were building a search engine.

One of the simplest and most widespread standardization schemes is “convert to lowercase and remove punctuation characters.” Our two sentences would become

- “sunset came i was staring at the mexico sky isnt nature splendid”
- “sunset came i stared at the méxico sky isnt nature splendid”

Much closer already. Another common transformation is to convert special characters to a standard form, such as replacing "é" with "e," "æ" with "ae," and so on. Our token "méxico" would then become "mexico".

Lastly, a much more advanced standardization pattern that is more rarely used in a machine learning context is *stemming*: converting variations of a term (such as different conjugated forms of a verb) into a single shared representation, like turning "caught" and "been catching" into "[catch]" or "cats" into "[cat]". With stemming, "was staring" and "stared" would become something like "[stare]", and our two similar sentences would finally end up with an identical encoding:

- "sunset came i [stare] at the mexico sky isnt nature splendid"

With these standardization techniques, your model will require less training data and will generalize better—it won't need abundant examples of both "Sunset" and "sunset" to learn that they mean the same thing, and it will be able to make sense of "México" even if it has only seen "mexico" in its training set. Of course, standardization may also erase some amount of information, so always keep the context in mind: for instance, if you're writing a model that extracts questions from interview articles, it should definitely treat "?" as a separate token instead of dropping it, because it's a useful signal for this specific task.

11.2.2 Text splitting (tokenization)

Once your text is standardized, you need to break it up into units to be vectorized (tokens), a step called *tokenization*. You could do this in three different ways:

- *Word-level tokenization*—Where tokens are space-separated (or punctuation-separated) substrings. A variant of this is to further split words into subwords when applicable—for instance, treating "staring" as "star+ing" or "called" as "call+ed."
- *N-gram tokenization*—Where tokens are groups of *N* consecutive words. For instance, "the cat" or "he was" would be 2-gram tokens (also called bigrams).
- *Character-level tokenization*—Where each character is its own token. In practice, this scheme is rarely used, and you only really see it in specialized contexts, like text generation or speech recognition.

In general, you'll always use either word-level or N-gram tokenization. There are two kinds of text-processing models: those that care about word order, called *sequence models*, and those that treat input words as a set, discarding their original order, called *bag-of-words models*. If you're building a sequence model, you'll use word-level tokenization, and if you're building a bag-of-words model, you'll use N-gram tokenization. N-grams are a way to artificially inject a small amount of local word order information into the model. Throughout this chapter, you'll learn more about each type of model and when to use them.

Understanding N-grams and bag-of-words

Word N-grams are groups of *N* (or fewer) consecutive words that you can extract from a sentence. The same concept may also be applied to characters instead of words.

Here's a simple example. Consider the sentence "the cat sat on the mat." It may be decomposed into the following set of 2-grams:

```
{"the", "the cat", "cat", "cat sat", "sat",
 "sat on", "on", "on the", "the mat", "mat"}
```

It may also be decomposed into the following set of 3-grams:

```
{"the", "the cat", "cat", "cat sat", "the cat sat",
 "sat", "sat on", "on", "cat sat on", "on the",
 "sat on the", "the mat", "mat", "on the mat"}
```

Such a set is called a *bag-of-2-grams* or *bag-of-3-grams*, respectively. The term "bag" here refers to the fact that you're dealing with a set of tokens rather than a list or sequence: the tokens have no specific order. This family of tokenization methods is called *bag-of-words* (or *bag-of-N-grams*).

Because bag-of-words isn't an order-preserving tokenization method (the tokens generated are understood as a set, not a sequence, and the general structure of the sentences is lost), it tends to be used in shallow language-processing models rather than in deep learning models. Extracting N-grams is a form of feature engineering, and deep learning sequence models do away with this manual approach, replacing it with hierarchical feature learning. One-dimensional convnets, recurrent neural networks, and Transformers are capable of learning representations for groups of words and characters without being explicitly told about the existence of such groups, by looking at continuous word or character sequences.

11.2.3 Vocabulary indexing

Once your text is split into tokens, you need to encode each token into a numerical representation. You could potentially do this in a stateless way, such as by hashing each token into a fixed binary vector, but in practice, the way you'd go about it is to build an index of all terms found in the training data (the "vocabulary"), and assign a unique integer to each entry in the vocabulary.

Something like this:

```
vocabulary = {}
for text in dataset:
    text = standardize(text)
    tokens = tokenize(text)
    for token in tokens:
        if token not in vocabulary:
            vocabulary[token] = len(vocabulary)
```

You can then convert that integer into a vector encoding that can be processed by a neural network, like a one-hot vector:

```
def one_hot_encode_token(token):
    vector = np.zeros((len(vocabulary),))
    token_index = vocabulary[token]
    vector[token_index] = 1
    return vector
```

Note that at this step it's common to restrict the vocabulary to only the top 20,000 or 30,000 most common words found in the training data. Any text dataset tends to feature an extremely large number of unique terms, most of which only show up once or twice—indexing those rare terms would result in an excessively large feature space, where most features would have almost no information content.

Remember when you were training your first deep learning models on the IMDB dataset in chapters 4 and 5? The data you were using from `keras.datasets.imdb` was already preprocessed into sequences of integers, where each integer stood for a given word. Back then, we used the setting `num_words=10000`, in order to restrict our vocabulary to the top 10,000 most common words found in the training data.

Now, there's an important detail here that we shouldn't overlook: when we look up a new token in our vocabulary index, it may not necessarily exist. Your training data may not have contained any instance of the word "cherimoya" (or maybe you excluded it from your index because it was too rare), so doing `token_index = vocabulary["cherimoya"]` may result in a `KeyError`. To handle this, you should use an "out of vocabulary" index (abbreviated as *OOV index*)—a catch-all for any token that wasn't in the index. It's usually index 1: you're actually doing `token_index = vocabulary.get(token, 1)`. When decoding a sequence of integers back into words, you'll replace 1 with something like "[UNK]" (which you'd call an "OOV token").

"Why use 1 and not 0?" you may ask. That's because 0 is already taken. There are two special tokens that you will commonly use: the OOV token (index 1), and the *mask token* (index 0). While the OOV token means "here was a word we did not recognize," the mask token tells us "ignore me, I'm not a word." You'd use it in particular to pad sequence data: because data batches need to be contiguous, all sequences in a batch of sequence data must have the same length, so shorter sequences should be padded to the length of the longest sequence. If you want to make a batch of data with the sequences `[5, 7, 124, 4, 89]` and `[8, 34, 21]`, it would have to look like this:

```
[[5,  7, 124, 4, 89]
 [8, 34,  21, 0,  0]]
```

The batches of integer sequences for the IMDB dataset that you worked with in chapters 4 and 5 were padded with zeros in this way.

11.2.4 Using the TextVectorization layer

Every step I've introduced so far would be very easy to implement in pure Python. Maybe you could write something like this:

```
import string

class Vectorizer:
    def standardize(self, text):
        text = text.lower()
        return "".join(char for char in text
                       if char not in string.punctuation)

    def tokenize(self, text):
        text = self.standardize(text)
        return text.split()

    def make_vocabulary(self, dataset):
        self.vocabulary = {"": 0, "[UNK]": 1}
        for text in dataset:
            text = self.standardize(text)
            tokens = self.tokenize(text)
            for token in tokens:
                if token not in self.vocabulary:
                    self.vocabulary[token] = len(self.vocabulary)
        self.inverse_vocabulary = dict(
            (v, k) for k, v in self.vocabulary.items())

    def encode(self, text):
        text = self.standardize(text)
        tokens = self.tokenize(text)
        return [self.vocabulary.get(token, 1) for token in tokens]

    def decode(self, int_sequence):
        return " ".join(
            self.inverse_vocabulary.get(i, "[UNK]") for i in int_sequence)

vectorizer = Vectorizer()
dataset = [
    "I write, erase, rewrite",
    "Erase again, and then",
    "A poppy blooms.",
]
vectorizer.make_vocabulary(dataset)
```

Haiku by poet Hokushi

It does the job:

```
>>> test_sentence = "I write, rewrite, and still rewrite again"
>>> encoded_sentence = vectorizer.encode(test_sentence)
>>> print(encoded_sentence)
[2, 3, 5, 7, 1, 5, 6]
>>> decoded_sentence = vectorizer.decode(encoded_sentence)
>>> print(decoded_sentence)
"i write rewrite and [UNK] rewrite again"
```

However, using something like this wouldn't be very performant. In practice, you'll work with the Keras `TextVectorization` layer, which is fast and efficient and can be dropped directly into a `tf.data` pipeline or a Keras model.

This is what the `TextVectorization` layer looks like:

```
from tensorflow.keras.layers import TextVectorization
text_vectorization = TextVectorization(
    output_mode="int",
)
```

Configures the layer to return sequences of words encoded as integer indices. There are several other output modes available, which you will see in action in a bit.

By default, the `TextVectorization` layer will use the setting "convert to lowercase and remove punctuation" for text standardization, and "split on whitespace" for tokenization. But importantly, you can provide custom functions for standardization and tokenization, which means the layer is flexible enough to handle any use case. Note that such custom functions should operate on `tf.string` tensors, not regular Python strings! For instance, the default layer behavior is equivalent to the following:

```
import re
import string
import tensorflow as tf

def custom_standardization_fn(string_tensor):
    lowercase_string = tf.strings.lower(string_tensor)
    return tf.strings.regex_replace(
        lowercase_string, f"[{re.escape(string.punctuation)}]", "")

def custom_split_fn(string_tensor):
    return tf.strings.split(string_tensor)

text_vectorization = TextVectorization(
    output_mode="int",
    standardize=custom_standardization_fn,
    split=custom_split_fn,
)
```

Convert strings to lowercase.

Replace punctuation characters with the empty string.

Split strings on whitespace.

To index the vocabulary of a text corpus, just call the `adapt()` method of the layer with a `Dataset` object that yields strings, or just with a list of Python strings:

```
dataset = [
    "I write, erase, rewrite",
    "Erase again, and then",
    "A poppy blooms.",
]
text_vectorization.adapt(dataset)
```

Note that you can retrieve the computed vocabulary via `get_vocabulary()`—this can be useful if you need to convert text encoded as integer sequences back into words. The first two entries in the vocabulary are the mask token (index 0) and the OOV token (index 1). Entries in the vocabulary list are sorted by frequency, so with a real-world dataset, very common words like "the" or "a" would come first.

Listing 11.1 Displaying the vocabulary

```
>>> text_vectorization.get_vocabulary()
["", "[UNK]", "erase", "write", ...]
```

For a demonstration, let's try to encode and then decode an example sentence:

```
>>> vocabulary = text_vectorization.get_vocabulary()
>>> test_sentence = "I write, rewrite, and still rewrite again"
>>> encoded_sentence = text_vectorization(test_sentence)
>>> print(encoded_sentence)
tf.Tensor([ 7  3  5  9  1  5 10], shape=(7,), dtype=int64)
>>> inverse_vocab = dict(enumerate(vocabulary))
>>> decoded_sentence = " ".join(inverse_vocab[int(i)] for i in encoded_sentence)
>>> print(decoded_sentence)
"i write rewrite and [UNK] rewrite again"
```

Using the `TextVectorization` layer in a `tf.data` pipeline or as part of a model

Importantly, because `TextVectorization` is mostly a dictionary lookup operation, it can't be executed on a GPU (or TPU)—only on a CPU. So if you're training your model on a GPU, your `TextVectorization` layer will run on the CPU before sending its output to the GPU. This has important performance implications.

There are two ways we could use our `TextVectorization` layer. The first option is to put it in the `tf.data` pipeline, like this:

```
int_sequence_dataset = string_dataset.map(      # string_dataset would be a dataset that yields string tensors.
    text_vectorization,
    num_parallel_calls=4)                       # The num_parallel_calls argument is used to parallelize the map() call across multiple CPU cores.
```

The second option is to make it part of the model (after all, it's a Keras layer), like this:

```
text_input = keras.Input(shape=(), dtype="string")          # Create a symbolic input that expects strings.
vectorized_text = text_vectorization(text_input)            # Apply the text vectorization layer to it.
embedded_input = keras.layers.Embedding(...)(vectorized_text)
output = ...
model = keras.Model(text_input, output)
# You can keep chaining new layers on top—just your regular Functional API model.
```

There's an important difference between the two: if the vectorization step is part of the model, it will happen synchronously with the rest of the model. This means that at each training step, the rest of the model (placed on the GPU) will have to wait for the output of the `TextVectorization` layer (placed on the CPU) to be ready in order to get to work. Meanwhile, putting the layer in the `tf.data` pipeline enables you to

do asynchronous preprocessing of your data on CPU: while the GPU runs the model on one batch of vectorized data, the CPU stays busy by vectorizing the next batch of raw strings.

So if you're training the model on GPU or TPU, you'll probably want to go with the first option to get the best performance. This is what we will do in all practical examples throughout this chapter. When training on a CPU, though, synchronous processing is fine: you will get 100% utilization of your cores regardless of which option you go with.

Now, if you were to export our model to a production environment, you would want to ship a model that accepts raw strings as input, like in the code snippet for the second option above—otherwise you would have to reimplement text standardization and tokenization in your production environment (maybe in JavaScript?), and you would face the risk of introducing small preprocessing discrepancies that would hurt the model's accuracy. Thankfully, the `TextVectorization` layer enables you to include text preprocessing right into your model, making it easier to deploy—even if you were originally using the layer as part of a `tf.data` pipeline. In the sidebar "Exporting a model that processes raw strings," you'll learn how to export an inference-only trained model that incorporates text preprocessing.

You've now learned everything you need to know about text preprocessing—let's move on to the modeling stage.

11.3 Two approaches for representing groups of words: Sets and sequences

How a machine learning model should represent *individual words* is a relatively uncontroversial question: they're categorical features (values from a predefined set), and we know how to handle those. They should be encoded as dimensions in a feature space, or as category vectors (word vectors in this case). A much more problematic question, however, is how to encode *the way words are woven into sentences*: word order.

The problem of order in natural language is an interesting one: unlike the steps of a timeseries, words in a sentence don't have a natural, canonical order. Different languages order similar words in very different ways. For instance, the sentence structure of English is quite different from that of Japanese. Even within a given language, you can typically say the same thing in different ways by reshuffling the words a bit. Even further, if you fully randomize the words in a short sentence, you can still largely figure out what it was saying—though in many cases significant ambiguity seems to arise. Order is clearly important, but its relationship to meaning isn't straightforward.

How to represent word order is the pivotal question from which different kinds of NLP architectures spring. The simplest thing you could do is just discard order and treat text as an unordered set of words—this gives you *bag-of-words models*. You could also decide that words should be processed strictly in the order in which they appear, one at a time, like steps in a timeseries—you could then leverage the recurrent models from the last chapter. Finally, a hybrid approach is also possible: the Transformer

architecture is technically order-agnostic, yet it injects word-position information into the representations it processes, which enables it to simultaneously look at different parts of a sentence (unlike RNNs) while still being order-aware. Because they take into account word order, both RNNs and Transformers are called *sequence models*.

Historically, most early applications of machine learning to NLP just involved bag-of-words models. Interest in sequence models only started rising in 2015, with the rebirth of recurrent neural networks. Today, both approaches remain relevant. Let's see how they work, and when to leverage which.

We'll demonstrate each approach on a well-known text classification benchmark: the IMDB movie review sentiment-classification dataset. In chapters 4 and 5, you worked with a prevectorized version of the IMDB dataset; now, let's process the raw IMDB text data, just like you would do when approaching a new text-classification problem in the real world.

11.3.1 Preparing the IMDB movie reviews data

Let's start by downloading the dataset from the Stanford page of Andrew Maas and uncompressing it:

```
!curl -O https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
!tar -xf aclImdb_v1.tar.gz
```

You're left with a directory named aclImdb, with the following structure:

```
aclImdb/
...train/
......pos/
......neg/
...test/
......pos/
......neg/
```

For instance, the train/pos/ directory contains a set of 12,500 text files, each of which contains the text body of a positive-sentiment movie review to be used as training data. The negative-sentiment reviews live in the "neg" directories. In total, there are 25,000 text files for training and another 25,000 for testing.

There's also a train/unsup subdirectory in there, which we don't need. Let's delete it:

```
!rm -r aclImdb/train/unsup
```

Take a look at the content of a few of these text files. Whether you're working with text data or image data, remember to always inspect what your data looks like before you dive into modeling it. It will ground your intuition about what your model is actually doing:

```
!cat aclImdb/train/pos/4077_10.txt
```

Next, let's prepare a validation set by setting apart 20% of the training text files in a new directory, aclImdb/val:

```
import os, pathlib, shutil, random

base_dir = pathlib.Path("aclImdb")
val_dir = base_dir / "val"
train_dir = base_dir / "train"
for category in ("neg", "pos"):
    os.makedirs(val_dir / category)
    files = os.listdir(train_dir / category)
    random.Random(1337).shuffle(files)
    num_val_samples = int(0.2 * len(files))
    val_files = files[-num_val_samples:]
    for fname in val_files:
        shutil.move(train_dir / category / fname,
                    val_dir / category / fname)
```

Shuffle the list of training files using a seed, to ensure we get the same validation set every time we run the code.

Take 20% of the training files to use for validation.

Move the files to aclImdb/val/neg and aclImdb/val/pos.

Remember how, in chapter 8, we used the `image_dataset_from_directory` utility to create a batched `Dataset` of images and their labels for a directory structure? You can do the exact same thing for text files using the `text_dataset_from_directory` utility. Let's create three `Dataset` objects for training, validation, and testing:

```
from tensorflow import keras
batch_size = 32

train_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/train", batch_size=batch_size
)
val_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/val", batch_size=batch_size
)
test_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/test", batch_size=batch_size
)
```

Running this line should output "Found 20000 files belonging to 2 classes"; if you see "Found 70000 files belonging to 3 classes," it means you forgot to delete the aclImdb/train/unsup directory.

These datasets yield inputs that are TensorFlow `tf.string` tensors and targets that are `int32` tensors encoding the value "0" or "1."

Listing 11.2 Displaying the shapes and dtypes of the first batch

```
>>> for inputs, targets in train_ds:
>>>     print("inputs.shape:", inputs.shape)
>>>     print("inputs.dtype:", inputs.dtype)
>>>     print("targets.shape:", targets.shape)
>>>     print("targets.dtype:", targets.dtype)
>>>     print("inputs[0]:", inputs[0])
>>>     print("targets[0]:", targets[0])
>>>     break
inputs.shape: (32,)
inputs.dtype: <dtype: "string">
targets.shape: (32,)
targets.dtype: <dtype: "int32">
```

```
inputs[0]: tf.Tensor(b"This string contains the movie review.", shape=(),
     dtype=string)
targets[0]: tf.Tensor(1, shape=(), dtype=int32)
```

All set. Now let's try learning something from this data.

11.3.2 Processing words as a set: The bag-of-words approach

The simplest way to encode a piece of text for processing by a machine learning model is to discard order and treat it as a set (a "bag") of tokens. You could either look at individual words (unigrams), or try to recover some local order information by looking at groups of consecutive token (N-grams).

SINGLE WORDS (UNIGRAMS) WITH BINARY ENCODING

If you use a bag of single words, the sentence "the cat sat on the mat" becomes

```
{"cat", "mat", "on", "sat", "the"}
```

The main advantage of this encoding is that you can represent an entire text as a single vector, where each entry is a presence indicator for a given word. For instance, using binary encoding (multi-hot), you'd encode a text as a vector with as many dimensions as there are words in your vocabulary—with 0s almost everywhere and some 1s for dimensions that encode words present in the text. This is what we did when we worked with text data in chapters 4 and 5. Let's try this on our task.

First, let's process our raw text datasets with a `TextVectorization` layer so that they yield multi-hot encoded binary word vectors. Our layer will only look at single words (that is to say, *unigrams*).

Listing 11.3 Preprocessing our datasets with a `TextVectorization` layer

Limit the vocabulary to the 20,000 most frequent words. Otherwise we'd be indexing every word in the training data—potentially tens of thousands of terms that only occur once or twice and thus aren't informative. In general, 20,000 is the right vocabulary size for text classification.

Encode the output tokens as multi-hot binary vectors.

Prepare a dataset that only yields raw text inputs (no labels).

Use that dataset to index the dataset vocabulary via the adapt() method.

```
text_vectorization = TextVectorization(
    max_tokens=20000,
    output_mode="multi_hot",
)
text_only_train_ds = train_ds.map(lambda x, y: x)
text_vectorization.adapt(text_only_train_ds)

binary_1gram_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
binary_1gram_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
binary_1gram_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
```

Prepare processed versions of our training, validation, and test dataset. Make sure to specify num_parallel_calls to leverage multiple CPU cores.

You can try to inspect the output of one of these datasets.

Listing 11.4 Inspecting the output of our binary unigram dataset

```
>>> for inputs, targets in binary_1gram_train_ds:
>>>     print("inputs.shape:", inputs.shape)
>>>     print("inputs.dtype:", inputs.dtype)
>>>     print("targets.shape:", targets.shape)
>>>     print("targets.dtype:", targets.dtype)
>>>     print("inputs[0]:", inputs[0])
>>>     print("targets[0]:", targets[0])
>>>     break
inputs.shape: (32, 20000)
inputs.dtype: <dtype: "float32">
targets.shape: (32,)
targets.dtype: <dtype: "int32">
inputs[0]: tf.Tensor([1. 1. 1. ... 0. 0. 0.], shape=(20000,), dtype=float32)
targets[0]: tf.Tensor(1, shape=(), dtype=int32)
```

Inputs are batches of 20,000-dimensional vectors.

These vectors consist entirely of ones and zeros.

Next, let's write a reusable model-building function that we'll use in all of our experiments in this section.

Listing 11.5 Our model-building utility

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(max_tokens=20000, hidden_dim=16):
    inputs = keras.Input(shape=(max_tokens,))
    x = layers.Dense(hidden_dim, activation="relu")(inputs)
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(1, activation="sigmoid")(x)
    model = keras.Model(inputs, outputs)
    model.compile(optimizer="rmsprop",
                  loss="binary_crossentropy",
                  metrics=["accuracy"])
    return model
```

Finally, let's train and test our model.

Listing 11.6 Training and testing the binary unigram model

```
model = get_model()
model.summary()
callbacks = [
    keras.callbacks.ModelCheckpoint("binary_1gram.keras",
                                    save_best_only=True)
]
model.fit(binary_1gram_train_ds.cache(),
          validation_data=binary_1gram_val_ds.cache(),
          epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("binary_1gram.keras")
print(f"Test acc: {model.evaluate(binary_1gram_test_ds)[1]:.3f}")
```

We call cache() on the datasets to cache them in memory: this way, we will only do the preprocessing once, during the first epoch, and we'll reuse the preprocessed texts for the following epochs. This can only be done if the data is small enough to fit in memory.

This gets us to a test accuracy of 89.2%: not bad! Note that in this case, since the dataset is a balanced two-class classification dataset (there are as many positive samples as negative samples), the "naive baseline" we could reach without training an actual model would only be 50%. Meanwhile, the best score that can be achieved on this dataset without leveraging external data is around 95% test accuracy.

BIGRAMS WITH BINARY ENCODING

Of course, discarding word order is very reductive, because even atomic concepts can be expressed via multiple words: the term "United States" conveys a concept that is quite distinct from the meaning of the words "states" and "united" taken separately. For this reason, you will usually end up re-injecting local order information into your bag-of-words representation by looking at N-grams rather than single words (most commonly, bigrams).

With bigrams, our sentence becomes

```
{"the", "the cat", "cat", "cat sat", "sat",
 "sat on", "on", "on the", "the mat", "mat"}
```

The `TextVectorization` layer can be configured to return arbitrary N-grams: bigrams, trigrams, etc. Just pass an `ngrams=N` argument as in the following listing.

Listing 11.7 Configuring the `TextVectorization` layer to return bigrams

```
text_vectorization = TextVectorization(
    ngrams=2,
    max_tokens=20000,
    output_mode="multi_hot",
)
```

Let's test how our model performs when trained on such binary-encoded bags of bigrams.

Listing 11.8 Training and testing the binary bigram model

```
text_vectorization.adapt(text_only_train_ds)
binary_2gram_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
binary_2gram_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
binary_2gram_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)

model = get_model()
model.summary()
callbacks = [
    keras.callbacks.ModelCheckpoint("binary_2gram.keras",
                                    save_best_only=True)
]
```

```
model.fit(binary_2gram_train_ds.cache(),
          validation_data=binary_2gram_val_ds.cache(),
          epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("binary_2gram.keras")
print(f"Test acc: {model.evaluate(binary_2gram_test_ds)[1]:.3f}")
```

We're now getting 90.4% test accuracy, a marked improvement! Turns out local order is pretty important.

Bigrams with TF-IDF encoding

You can also add a bit more information to this representation by counting how many times each word or N-gram occurs, that is to say, by taking the histogram of the words over the text:

```
{"the": 2, "the cat": 1, "cat": 1, "cat sat": 1, "sat": 1,
 "sat on": 1, "on": 1, "on the": 1, "the mat: 1", "mat": 1}
```

If you're doing text classification, knowing how many times a word occurs in a sample is critical: any sufficiently long movie review may contain the word "terrible" regardless of sentiment, but a review that contains many instances of the word "terrible" is likely a negative one.

Here's how you'd count bigram occurrences with the `TextVectorization` layer.

Listing 11.9 Configuring the `TextVectorization` layer to return token counts

```
text_vectorization = TextVectorization(
    ngrams=2,
    max_tokens=20000,
    output_mode="count"
)
```

Now, of course, some words are bound to occur more often than others no matter what the text is about. The words "the," "a," "is," and "are" will always dominate your word count histograms, drowning out other words—despite being pretty much useless features in a classification context. How could we address this?

You already guessed it: via normalization. We could just normalize word counts by subtracting the mean and dividing by the variance (computed across the entire training dataset). That would make sense. Except most vectorized sentences consist almost entirely of zeros (our previous example features 12 non-zero entries and 19,988 zero entries), a property called "sparsity." That's a great property to have, as it dramatically reduces compute load and reduces the risk of overfitting. If we subtracted the mean from each feature, we'd wreck sparsity. Thus, whatever normalization scheme we use should be divide-only. What, then, should we use as the denominator? The best practice is to go with something called *TF-IDF normalization*—TF-IDF stands for "term frequency, inverse document frequency."

TF-IDF is so common that it's built into the `TextVectorization` layer. All you need to do to start using it is to switch the `output_mode` argument to `"tf_idf"`.

Understanding TF-IDF normalization

The more a given term appears in a document, the more important that term is for understanding what the document is about. At the same time, the frequency at which the term appears across all documents in your dataset matters too: terms that appear in almost every document (like "the" or "a") aren't particularly informative, while terms that appear only in a small subset of all texts (like "Herzog") are very distinctive, and thus important. TF-IDF is a metric that fuses these two ideas. It weights a given term by taking "term frequency," how many times the term appears in the current document, and dividing it by a measure of "document frequency," which estimates how often the term comes up across the dataset. You'd compute it as follows:

```
def tfidf(term, document, dataset):
    term_freq = document.count(term)
    doc_freq = math.log(sum(doc.count(term) for doc in dataset) + 1)
    return term_freq / doc_freq
```

Listing 11.10 Configuring `TextVectorization` to return TF-IDF-weighted outputs

```
text_vectorization = TextVectorization(
    ngrams=2,
    max_tokens=20000,
    output_mode="tf_idf",
)
```

Let's train a new model with this scheme.

Listing 11.11 Training and testing the TF-IDF bigram model

```
text_vectorization.adapt(text_only_train_ds)    <-- The adapt() call will learn the TF-IDF weights in addition to the vocabulary.

tfidf_2gram_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
tfidf_2gram_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
tfidf_2gram_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)

model = get_model()
model.summary()
callbacks = [
    keras.callbacks.ModelCheckpoint("tfidf_2gram.keras",
                                    save_best_only=True)
]
model.fit(tfidf_2gram_train_ds.cache(),
          validation_data=tfidf_2gram_val_ds.cache(),
          epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("tfidf_2gram.keras")
print(f"Test acc: {model.evaluate(tfidf_2gram_test_ds)[1]:.3f}")
```

This gets us an 89.8% test accuracy on the IMDB classification task: it doesn't seem to be particularly helpful in this case. However, for many text-classification datasets, it would be typical to see a one-percentage-point increase when using TF-IDF compared to plain binary encoding.

Exporting a model that processes raw strings

In the preceding examples, we did our text standardization, splitting, and indexing as part of the `tf.data` pipeline. But if we want to export a standalone model independent of this pipeline, we should make sure that it incorporates its own text preprocessing (otherwise, you'd have to reimplement in the production environment, which can be challenging or can lead to subtle discrepancies between the training data and the production data). Thankfully, this is easy.

Just create a new model that reuses your `TextVectorization` layer and adds to it the model you just trained:

```
inputs = keras.Input(shape=(1,), dtype="string")     # One input sample would be one string.
processed_inputs = text_vectorization(inputs)        # Apply text preprocessing.
outputs = model(processed_inputs)                    # Apply the previously trained model.
inference_model = keras.Model(inputs, outputs)       # Instantiate the end-to-end model.
```

The resulting model can process batches of raw strings:

```
import tensorflow as tf
raw_text_data = tf.convert_to_tensor([
    ["That was an excellent movie, I loved it."],
])
predictions = inference_model(raw_text_data)
print(f"{float(predictions[0] * 100):.2f} percent positive")
```

11.3.3 Processing words as a sequence: The sequence model approach

These past few examples clearly show that word order matters: manual engineering of order-based features, such as bigrams, yields a nice accuracy boost. Now remember: the history of deep learning is that of a move away from manual feature engineering, toward letting models learn their own features from exposure to data alone. What if, instead of manually crafting order-based features, we exposed the model to raw word sequences and let it figure out such features on its own? This is what *sequence models* are about.

To implement a sequence model, you'd start by representing your input samples as sequences of integer indices (one integer standing for one word). Then, you'd map each integer to a vector to obtain vector sequences. Finally, you'd feed these sequences of vectors into a stack of layers that could cross-correlate features from adjacent vectors, such as a 1D convnet, a RNN, or a Transformer.

For some time around 2016–2017, bidirectional RNNs (in particular, bidirectional LSTMs) were considered to be the state of the art for sequence modeling. Since you're

already familiar with this architecture, this is what we'll use in our first sequence model examples. However, nowadays sequence modeling is almost universally done with Transformers, which we will cover shortly. Oddly, one-dimensional convnets were never very popular in NLP, even though, in my own experience, a residual stack of depthwise-separable 1D convolutions can often achieve comparable performance to a bidirectional LSTM, at a greatly reduced computational cost.

A FIRST PRACTICAL EXAMPLE

Let's try out a first sequence model in practice. First, let's prepare datasets that return integer sequences.

Listing 11.12 Preparing integer sequence datasets

```
from tensorflow.keras import layers

max_length = 600
max_tokens = 20000
text_vectorization = layers.TextVectorization(
    max_tokens=max_tokens,
    output_mode="int",
    output_sequence_length=max_length,
)
text_vectorization.adapt(text_only_train_ds)

int_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y)),
    num_parallel_calls=4)
int_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
int_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
```

In order to keep a manageable input size, we'll truncate the inputs after the first 600 words. This is a reasonable choice, since the average review length is 233 words, and only 5% of reviews are longer than 600 words.

Next, let's make a model. The simplest way to convert our integer sequences to vector sequences is to one-hot encode the integers (each dimension would represent one possible term in the vocabulary). On top of these one-hot vectors, we'll add a simple bidirectional LSTM.

Listing 11.13 A sequence model built on one-hot encoded vector sequences

```
import tensorflow as tf
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = tf.one_hot(inputs, depth=max_tokens)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()
```

One input is a sequence of integers.

Encode the integers into binary 20,000-dimensional vectors.

Add a bidirectional LSTM.

Finally, add a classification layer.

Now, let's train our model.

Listing 11.14 Training a first basic sequence model

```
callbacks = [
    keras.callbacks.ModelCheckpoint("one_hot_bidir_lstm.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("one_hot_bidir_lstm.keras")
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

A first observation: this model trains very slowly, especially compared to the lightweight model of the previous section. This is because our inputs are quite large: each input sample is encoded as a matrix of size `(600, 20000)` (600 words per sample, 20,000 possible words). That's 12,000,000 floats for a single movie review. Our bidirectional LSTM has a lot of work to do. Second, the model only gets to 87% test accuracy—it doesn't perform nearly as well as our (very fast) binary unigram model.

Clearly, using one-hot encoding to turn words into vectors, which was the simplest thing we could do, wasn't a great idea. There's a better way: *word embeddings*.

UNDERSTANDING WORD EMBEDDINGS

Crucially, when you encode something via one-hot encoding, you're making a feature-engineering decision. You're injecting into your model a fundamental assumption about the structure of your feature space. That assumption is that *the different tokens you're encoding are all independent from each other*: indeed, one-hot vectors are all orthogonal to one another. And in the case of words, that assumption is clearly wrong. Words form a structured space: they share information with each other. The words "movie" and "film" are interchangeable in most sentences, so the vector that represents "movie" should not be orthogonal to the vector that represents "film"—they should be the same vector, or close enough.

To get a bit more abstract, the *geometric relationship* between two word vectors should reflect the *semantic relationship* between these words. For instance, in a reasonable word vector space, you would expect synonyms to be embedded into similar word vectors, and in general, you would expect the geometric distance (such as the cosine distance or L2 distance) between any two word vectors to relate to the "semantic distance" between the associated words. Words that mean different things should lie far away from each other, whereas related words should be closer.

Word embeddings are vector representations of words that achieve exactly this: they map human language into a structured geometric space.

Whereas the vectors obtained through one-hot encoding are binary, sparse (mostly made of zeros), and very high-dimensional (the same dimensionality as the number of words in the vocabulary), word embeddings are low-dimensional floating-point vectors (that is, dense vectors, as opposed to sparse vectors); see figure 11.2. It's common to see word embeddings that are 256-dimensional, 512-dimensional, or 1,024-dimensional

when dealing with very large vocabularies. On the other hand, one-hot encoding words generally leads to vectors that are 20,000-dimensional or greater (capturing a vocabulary of 20,000 tokens, in this case). So, word embeddings pack more information into far fewer dimensions.

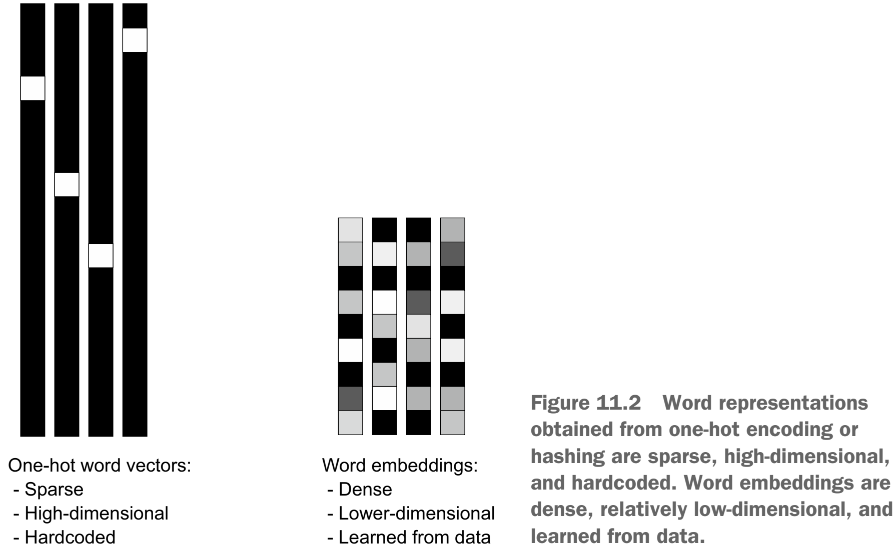


Figure 11.2 Word representations obtained from one-hot encoding or hashing are sparse, high-dimensional, and hardcoded. Word embeddings are dense, relatively low-dimensional, and learned from data.

Besides being *dense* representations, word embeddings are also *structured* representations, and their structure is learned from data. Similar words get embedded in close locations, and further, specific *directions* in the embedding space are meaningful. To make this clearer, let's look at a concrete example.

In figure 11.3, four words are embedded on a 2D plane: *cat*, *dog*, *wolf*, and *tiger*. With the vector representations we chose here, some semantic relationships between these words can be encoded as geometric transformations. For instance, the same vector allows us to go from *cat* to *tiger* and from *dog* to *wolf*: this vector could be interpreted as the "from pet to wild animal" vector. Similarly, another vector lets us go from *dog* to *cat* and from *wolf* to *tiger*, which could be interpreted as a "from canine to feline" vector.

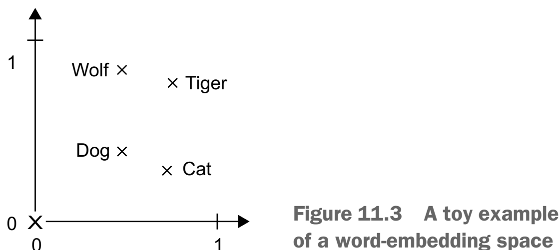


Figure 11.3 A toy example of a word-embedding space

In real-world word-embedding spaces, common examples of meaningful geometric transformations are "gender" vectors and "plural" vectors. For instance, by adding a "female" vector to the vector "king," we obtain the vector "queen." By adding a "plural" vector, we obtain "kings." Word-embedding spaces typically feature thousands of such interpretable and potentially useful vectors.

Let's look at how to use such an embedding space in practice. There are two ways to obtain word embeddings:

- Learn word embeddings jointly with the main task you care about (such as document classification or sentiment prediction). In this setup, you start with random word vectors and then learn word vectors in the same way you learn the weights of a neural network.
- Load into your model word embeddings that were precomputed using a different machine learning task than the one you're trying to solve. These are called *pretrained word embeddings*.

Let's review each of these approaches.

LEARNING WORD EMBEDDINGS WITH THE EMBEDDING LAYER

Is there some ideal word-embedding space that would perfectly map human language and could be used for any natural language processing task? Possibly, but we have yet to compute anything of the sort. Also, there is no such a thing as *human language*—there are many different languages, and they aren't isomorphic to one another, because a language is the reflection of a specific culture and a specific context. But more pragmatically, what makes a good word-embedding space depends heavily on your task: the perfect word-embedding space for an English-language movie-review sentiment-analysis model may look different from the perfect embedding space for an English-language legal-document classification model, because the importance of certain semantic relationships varies from task to task.

It's thus reasonable to *learn* a new embedding space with every new task. Fortunately, backpropagation makes this easy, and Keras makes it even easier. It's about learning the weights of a layer: the `Embedding` layer.

Listing 11.15 Instantiating an `Embedding` layer

```
embedding_layer = layers.Embedding(input_dim=max_tokens, output_dim=256)
```

The Embedding layer takes at least two arguments: the number of possible tokens and the dimensionality of the embeddings (here, 256).

The `Embedding` layer is best understood as a dictionary that maps integer indices (which stand for specific words) to dense vectors. It takes integers as input, looks up these integers in an internal dictionary, and returns the associated vectors. It's effectively a dictionary lookup (see figure 11.4).

Word index ——► Embedding layer ——► Corresponding word vector

Figure 11.4 The `Embedding` layer

The `Embedding` layer takes as input a rank-2 tensor of integers, of shape `(batch_size, sequence_length)`, where each entry is a sequence of integers. The layer then returns a 3D floating-point tensor of shape `(batch_size, sequence_length, embedding_dimensionality)`.

When you instantiate an `Embedding` layer, its weights (its internal dictionary of token vectors) are initially random, just as with any other layer. During training, these word vectors are gradually adjusted via backpropagation, structuring the space into something the downstream model can exploit. Once fully trained, the embedding space will show a lot of structure—a kind of structure specialized for the specific problem for which you're training your model.

Let's build a model that includes an `Embedding` layer and benchmark it on our task.

Listing 11.16 Model that uses an `Embedding` layer trained from scratch

```
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = layers.Embedding(input_dim=max_tokens, output_dim=256)(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()

callbacks = [
    keras.callbacks.ModelCheckpoint("embeddings_bidir_gru.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("embeddings_bidir_gru.keras")
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

It trains much faster than the one-hot model (since the LSTM only has to process 256-dimensional vectors instead of 20,000-dimensional), and its test accuracy is comparable (87%). However, we're still some way off from the results of our basic bigram model. Part of the reason why is simply that the model is looking at slightly less data: the bigram model processed full reviews, while our sequence model truncates sequences after 600 words.

UNDERSTANDING PADDING AND MASKING

One thing that's slightly hurting model performance here is that our input sequences are full of zeros. This comes from our use of the `output_sequence_length=max_length` option in `TextVectorization` (with `max_length` equal to 600): sentences longer than 600 tokens are truncated to a length of 600 tokens, and sentences shorter than 600 tokens are padded with zeros at the end so that they can be concatenated together with other sequences to form contiguous batches.

We're using a bidirectional RNN: two RNN layers running in parallel, with one processing the tokens in their natural order, and the other processing the same tokens in reverse. The RNN that looks at the tokens in their natural order will spend its last iterations seeing only vectors that encode padding—possibly for several hundreds of iterations if the original sentence was short. The information stored in the internal state of the RNN will gradually fade out as it gets exposed to these meaningless inputs.

We need some way to tell the RNN that it should skip these iterations. There's an API for that: *masking*.

The `Embedding` layer is capable of generating a "mask" that corresponds to its input data. This mask is a tensor of ones and zeros (or True/False booleans), of shape `(batch_size, sequence_length)`, where the entry `mask[i, t]` indicates where timestep `t` of sample `i` should be skipped or not (the timestep will be skipped if `mask[i, t]` is 0 or False, and processed otherwise).

By default, this option isn't active—you can turn it on by passing `mask_zero=True` to your `Embedding` layer. You can retrieve the mask with the `compute_mask()` method:

```
>>> embedding_layer = Embedding(input_dim=10, output_dim=256, mask_zero=True)
>>>  some_input = [
...   [4, 3, 2, 1, 0, 0, 0],
...   [5, 4, 3, 2, 1, 0, 0],
...   [2, 1, 0, 0, 0, 0, 0]]
>>> mask = embedding_layer.compute_mask(some_input)
<tf.Tensor: shape=(3, 7), dtype=bool, numpy=
array([[ True,  True,  True,  True, False, False, False],
       [ True,  True,  True,  True,  True, False, False],
       [ True,  True, False, False, False, False, False]])>
```

In practice, you will almost never have to manage masks by hand. Instead, Keras will automatically pass on the mask to every layer that is able to process it (as a piece of metadata attached to the sequence it represents). This mask will be used by RNN layers to skip masked steps. If your model returns an entire sequence, the mask will also be used by the loss function to skip masked steps in the output sequence.

Let's try retraining our model with masking enabled.

Listing 11.17 Using an `Embedding` layer with masking enabled

```
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = layers.Embedding(
    input_dim=max_tokens, output_dim=256, mask_zero=True)(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()
```

```
callbacks = [
    keras.callbacks.ModelCheckpoint("embeddings_bidir_gru_with_masking.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("embeddings_bidir_gru_with_masking.keras")
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

This time we get to 88% test accuracy—a small but noticeable improvement.

USING PRETRAINED WORD EMBEDDINGS

Sometimes you have so little training data available that you can't use your data alone to learn an appropriate task-specific embedding of your vocabulary. In such cases, instead of learning word embeddings jointly with the problem you want to solve, you can load embedding vectors from a precomputed embedding space that you know is highly structured and exhibits useful properties—one that captures generic aspects of language structure. The rationale behind using pretrained word embeddings in natural language processing is much the same as for using pretrained convnets in image classification: you don't have enough data available to learn truly powerful features on your own, but you expect that the features you need are fairly generic—that is, common visual features or semantic features. In this case, it makes sense to reuse features learned on a different problem.

Such word embeddings are generally computed using word-occurrence statistics (observations about what words co-occur in sentences or documents), using a variety of techniques, some involving neural networks, others not. The idea of a dense, low-dimensional embedding space for words, computed in an unsupervised way, was initially explored by Bengio et al. in the early 2000s,[1] but it only started to take off in research and industry applications after the release of one of the most famous and successful word-embedding schemes: the Word2Vec algorithm (https://code.google.com/archive/p/word2vec), developed by Tomas Mikolov at Google in 2013. Word2Vec dimensions capture specific semantic properties, such as gender.

There are various precomputed databases of word embeddings that you can download and use in a Keras `Embedding` layer. Word2vec is one of them. Another popular one is called Global Vectors for Word Representation (GloVe, https://nlp.stanford.edu/projects/glove), which was developed by Stanford researchers in 2014. This embedding technique is based on factorizing a matrix of word co-occurrence statistics. Its developers have made available precomputed embeddings for millions of English tokens, obtained from Wikipedia data and Common Crawl data.

Let's look at how you can get started using GloVe embeddings in a Keras model. The same method is valid for Word2Vec embeddings or any other word-embedding database. We'll start by downloading the GloVe files and parse them. We'll then load the word vectors into a Keras `Embedding` layer, which we'll use to build a new model.

[1] Yoshua Bengio et al., "A Neural Probabilistic Language Model," *Journal of Machine Learning Research* (2003).

First, let's download the GloVe word embeddings precomputed on the 2014 English Wikipedia dataset. It's an 822 MB zip file containing 100-dimensional embedding vectors for 400,000 words (or non-word tokens).

```
!wget http://nlp.stanford.edu/data/glove.6B.zip
!unzip -q glove.6B.zip
```

Let's parse the unzipped file (a .txt file) to build an index that maps words (as strings) to their vector representation.

Listing 11.18 Parsing the GloVe word-embeddings file

```
import numpy as np
path_to_glove_file = "glove.6B.100d.txt"

embeddings_index = {}
with open(path_to_glove_file) as f:
    for line in f:
        word, coefs = line.split(maxsplit=1)
        coefs = np.fromstring(coefs, "f", sep=" ")
        embeddings_index[word] = coefs

print(f"Found {len(embeddings_index)} word vectors.")
```

Next, let's build an embedding matrix that you can load into an `Embedding` layer. It must be a matrix of shape `(max_words, embedding_dim)`, where each entry *i* contains the `embedding_dim`-dimensional vector for the word of index *i* in the reference word index (built during tokenization).

Listing 11.19 Preparing the GloVe word-embeddings matrix

```
embedding_dim = 100

vocabulary = text_vectorization.get_vocabulary()
word_index = dict(zip(vocabulary, range(len(vocabulary))))

embedding_matrix = np.zeros((max_tokens, embedding_dim))
for word, i in word_index.items():
    if i < max_tokens:
        embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
```

Retrieve the vocabulary indexed by our previous TextVectorization layer.

Use it to create a mapping from words to their index in the vocabulary.

Prepare a matrix that we'll fill with the GloVe vectors.

Fill entry *i* in the matrix with the word vector for index *i*. Words not found in the embedding index will be all zeros.

Finally, we use a `Constant` initializer to load the pretrained embeddings in an `Embedding` layer. So as not to disrupt the pretrained representations during training, we freeze the layer via `trainable=False`:

```
embedding_layer = layers.Embedding(
    max_tokens,
    embedding_dim,
    embeddings_initializer=keras.initializers.Constant(embedding_matrix),
```

```
    trainable=False,
    mask_zero=True,
)
```

We're now ready to train a new model—identical to our previous model, but leveraging the 100-dimensional pretrained GloVe embeddings instead of 128-dimensional learned embeddings.

Listing 11.20 Model that uses a pretrained Embedding layer

```
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = embedding_layer(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()

callbacks = [
    keras.callbacks.ModelCheckpoint("glove_embeddings_sequence_model.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10,
          callbacks=callbacks)
model = keras.models.load_model("glove_embeddings_sequence_model.keras")
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

You'll find that on this particular task, pretrained embeddings aren't very helpful, because the dataset contains enough samples that it is possible to learn a specialized enough embedding space from scratch. However, leveraging pretrained embeddings can be very helpful when you're working with a smaller dataset.

11.4 The Transformer architecture

Starting in 2017, a new model architecture started overtaking recurrent neural networks across most natural language processing tasks: the Transformer.

Transformers were introduced in the seminal paper "Attention is all you need" by Vaswani et al.[2] The gist of the paper is right there in the title: as it turned out, a simple mechanism called "neural attention" could be used to build powerful sequence models that didn't feature any recurrent layers or convolution layers.

This finding unleashed nothing short of a revolution in natural language processing—and beyond. Neural attention has fast become one of the most influential ideas in deep learning. In this section, you'll get an in-depth explanation of how it works and why it has proven so effective for sequence data. We'll then leverage self-attention

[2] Ashish Vaswani et al., "Attention is all you need" (2017), https://arxiv.org/abs/1706.03762.

to create a Transformer encoder, one of the basic components of the Transformer architecture, and we'll apply it to the IMDB movie review classification task.

11.4.1 Understanding self-attention

As you're going through this book, you may be skimming some parts and attentively reading others, depending on what your goals or interests are. What if your models did the same? It's a simple yet powerful idea: not all input information seen by a model is equally important to the task at hand, so models should "pay more attention" to some features and "pay less attention" to other features.

Does that sound familiar? You've already encountered a similar concept twice in this book:

- Max pooling in convnets looks at a pool of features in a spatial region and selects just one feature to keep. That's an "all or nothing" form of attention: keep the most important feature and discard the rest.
- TF-IDF normalization assigns importance scores to tokens based on how much information different tokens are likely to carry. Important tokens get boosted while irrelevant tokens get faded out. That's a continuous form of attention.

There are many different forms of attention you could imagine, but they all start by computing importance scores for a set of features, with higher scores for more relevant features and lower scores for less relevant ones (see figure 11.5). How these scores should be computed, and what you should do with them, will vary from approach to approach.

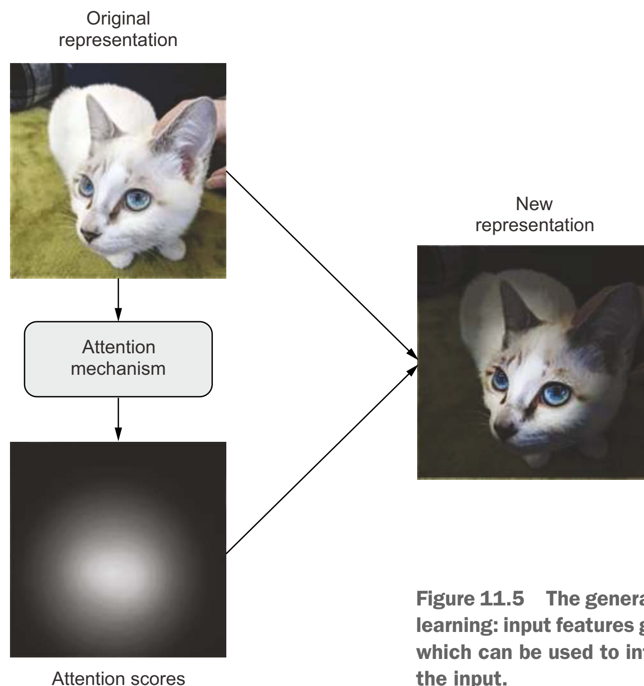


Figure 11.5 The general concept of "attention" in deep learning: input features get assigned "attention scores," which can be used to inform the next representation of the input.

Crucially, this kind of attention mechanism can be used for more than just highlighting or erasing certain features. It can be used to make features *context-aware*. You've just learned about word embeddings—vector spaces that capture the "shape" of the semantic relationships between different words. In an embedding space, a single word has a fixed position—a fixed set of relationships with every other word in the space. But that's not quite how language works: the meaning of a word is usually context-specific. When you mark the date, you're not talking about the same "date" as when you go on a date, nor is it the kind of date you'd buy at the market. When you say, "I'll see you soon," the meaning of the word "see" is subtly different from the "see" in "I'll see this project to its end" or "I see what you mean." And, of course, the meaning of pronouns like "he," "it," "in," etc., is entirely sentence-specific and can even change multiple times within a single sentence.

Clearly, a smart embedding space would provide a different vector representation for a word depending on the other words surrounding it. That's where *self-attention* comes in. The purpose of self-attention is to modulate the representation of a token by using the representations of related tokens in the sequence. This produces context-aware token representations. Consider an example sentence: "The train left the station on time." Now, consider one word in the sentence: station. What kind of station are we talking about? Could it be a radio station? Maybe the International Space Station? Let's figure it out algorithmically via self-attention (see figure 11.6).

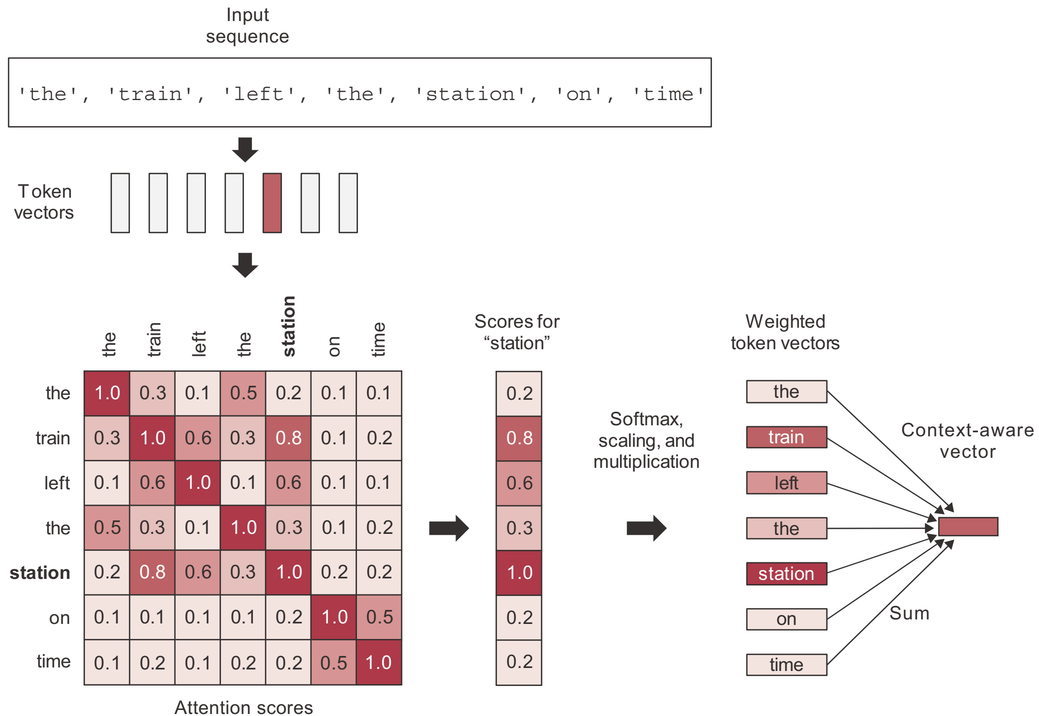


Figure 11.6 Self-attention: attention scores are computed between "station" and every other word in the sequence, and they are then used to weight a sum of word vectors that becomes the new "station" vector.

Step 1 is to compute relevancy scores between the vector for "station" and every other word in the sentence. These are our "attention scores." We're simply going to use the dot product between two word vectors as a measure of the strength of their relationship. It's a very computationally efficient distance function, and it was already the standard way to relate two word embeddings to each other long before Transformers. In practice, these scores will also go through a scaling function and a softmax, but for now, that's just an implementation detail.

Step 2 is to compute the sum of all word vectors in the sentence, weighted by our relevancy scores. Words closely related to "station" will contribute more to the sum (including the word "station" itself), while irrelevant words will contribute almost nothing. The resulting vector is our new representation for "station": a representation that incorporates the surrounding context. In particular, it includes part of the "train" vector, clarifying that it is, in fact, a "train station."

You'd repeat this process for every word in the sentence, producing a new sequence of vectors encoding the sentence. Let's see it in NumPy-like pseudocode:

```
def self_attention(input_sequence):
    output = np.zeros(shape=input_sequence.shape)
    for i, pivot_vector in enumerate(input_sequence):
        scores = np.zeros(shape=(len(input_sequence),))
        for j, vector in enumerate(input_sequence):
            scores[j] = np.dot(pivot_vector, vector.T)
        scores /= np.sqrt(input_sequence.shape[1])
        scores = softmax(scores)
        new_pivot_representation = np.zeros(shape=pivot_vector.shape)
        for j, vector in enumerate(input_sequence):
            new_pivot_representation += vector * scores[j]
        output[i] = new_pivot_representation
    return output
```

Iterate over each token in the input sequence.

Compute the dot product (attention score) between the token and every other token.

Scale by a normalization factor, and apply a softmax.

Take the sum of all tokens weighted by the attention scores.

That sum is our output.

Of course, in practice you'd use a vectorized implementation. Keras has a built-in layer to handle it: the `MultiHeadAttention` layer. Here's how you would use it:

```
num_heads = 4
embed_dim = 256
mha_layer = MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)
outputs = mha_layer(inputs, inputs, inputs)
```

Reading this, you're probably wondering

- Why are we passing the inputs to the layer *three* times? That seems redundant.
- What are these "multiple heads" we're referring to? That sounds intimidating—do they also grow back if you cut them?

Both of these questions have simple answers. Let's take a look.

GENERALIZED SELF-ATTENTION: THE QUERY-KEY-VALUE MODEL

So far, we have only considered one input sequence. However, the Transformer architecture was originally developed for machine translation, where you have to deal with two input sequences: the source sequence you're currently translating (such as "How's the weather today?"), and the target sequence you're converting it to (such as "¿Qué tiempo hace hoy?"). A Transformer is a *sequence-to-sequence* model: it was designed to convert one sequence into another. You'll learn about sequence-to-sequence models in depth later in this chapter.

Now let's take a step back. The self-attention mechanism as we've introduced it performs the following, schematically:

```
outputs = sum(inputs * pairwise_scores(inputs, inputs))
               ↑                        ↑       ↑
               C                        A       B
```

This means "for each token in `inputs` (A), compute how much the token is related to every token in `inputs` (B), and use these scores to weight a sum of tokens from `inputs` (C)." Crucially, there's nothing that requires A, B, and C to refer to the same input sequence. In the general case, you could be doing this with three different sequences. We'll call them "query," "keys," and "values." The operation becomes "for each element in the query, compute how much the element is related to every key, and use these scores to weight a sum of values":

```
outputs = sum(values * pairwise_scores(query, keys))
```

This terminology comes from search engines and recommender systems (see figure 11.7). Imagine that you're typing up a query to retrieve a photo from your collection—"dogs on the beach." Internally, each of your pictures in the database is described by a set of keywords—"cat," "dog," "party," etc. We'll call those "keys." The search engine will start by comparing your query to the keys in the database. "Dog" yields a match of 1, and "cat" yields a match of 0. It will then rank those keys by strength of match—relevance—and it will return the pictures associated with the top *N* matches, in order of relevance.

Conceptually, this is what Transformer-style attention is doing. You've got a reference sequence that describes something you're looking for: the query. You've got a body of knowledge that you're trying to extract information from: the values. Each value is assigned a key that describes the value in a format that can be readily compared to a query. You simply match the query to the keys. Then you return a weighted sum of values.

In practice, the keys and the values are often the same sequence. In machine translation, for instance, the query would be the target sequence, and the source sequence would play the roles of both keys and values: for each element of the target (like

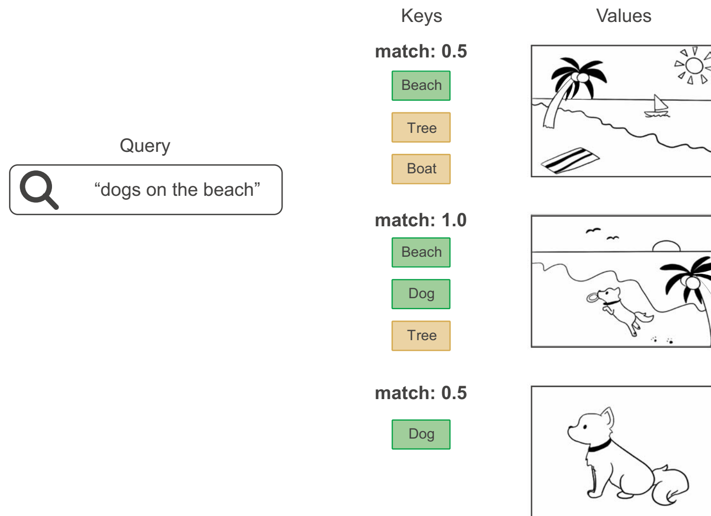


Figure 11.7 Retrieving images from a database: the "query" is compared to a set of "keys," and the match scores are used to rank "values" (images).

"tiempo"), you want to go back to the source ("How's the weather today?") and identify the different bits that are related to it ("tiempo" and "weather" should have a strong match). And naturally, if you're just doing sequence classification, then query, keys, and values are all the same: you're comparing a sequence to itself, to enrich each token with context from the whole sequence.

That explains why we needed to pass `inputs` three times to our `MultiHeadAttention` layer. But why "multi-head" attention?

11.4.2 Multi-head attention

"Multi-head attention" is an extra tweak to the self-attention mechanism, introduced in "Attention is all you need." The "multi-head" moniker refers to the fact that the output space of the self-attention layer gets factored into a set of independent subspaces, learned separately: the initial query, key, and value are sent through three independent sets of dense projections, resulting in three separate vectors. Each vector is processed via neural attention, and the three outputs are concatenated back together into a single output sequence. Each such subspace is called a "head." The full picture is shown in figure 11.8.

The presence of the learnable dense projections enables the layer to actually learn something, as opposed to being a purely stateless transformation that would require additional layers before or after it to be useful. In addition, having independent heads helps the layer learn different groups of features for each token, where features within one group are correlated with each other but are mostly independent from features in a different group.

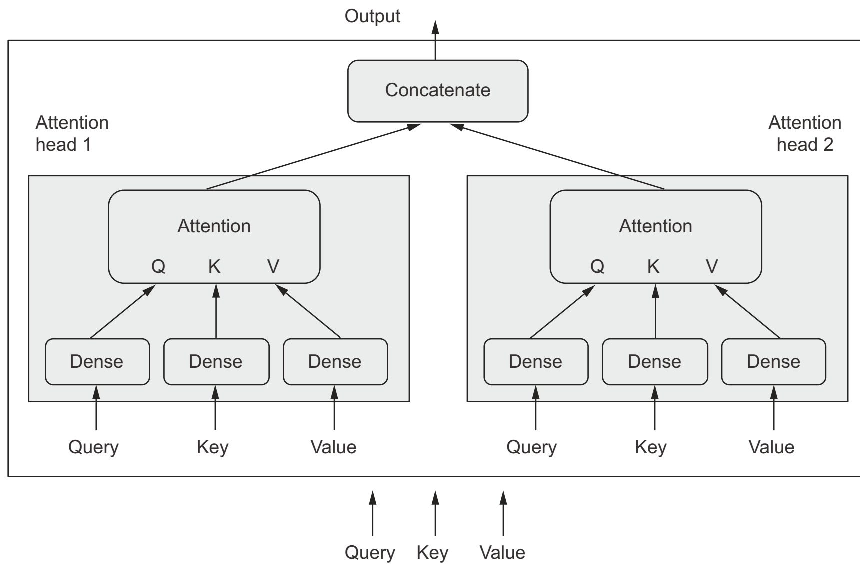


Figure 11.8 The `MultiHeadAttention` layer

This is similar in principle to what makes depthwise separable convolutions work: in a depthwise separable convolution, the output space of the convolution is factored into many subspaces (one per input channel) that get learned independently. The "Attention is all you need" paper was written at a time when the idea of factoring feature spaces into independent subspaces had been shown to provide great benefits for computer vision models—both in the case of depthwise separable convolutions, and in the case of a closely related approach, *grouped convolutions*. Multi-head attention is simply the application of the same idea to self-attention.

11.4.3 The Transformer encoder

If adding extra dense projections is so useful, why don't we also apply one or two to the output of the attention mechanism? Actually, that's a great idea—let's do that. And our model is starting to do a lot, so we might want to add residual connections to make sure we don't destroy any valuable information along the way—you learned in chapter 9 that they're a must for any sufficiently deep architecture. And there's another thing you learned in chapter 9: normalization layers are supposed to help gradients flow better during backpropagation. Let's add those too.

That's roughly the thought process that I imagine unfolded in the minds of the inventors of the Transformer architecture at the time. Factoring outputs into multiple independent spaces, adding residual connections, adding normalization layers—all of these are standard architecture patterns that one would be wise to leverage in any complex model. Together, these bells and whistles form the *Transformer encoder*—one of two critical parts that make up the Transformer architecture (see figure 11.9).

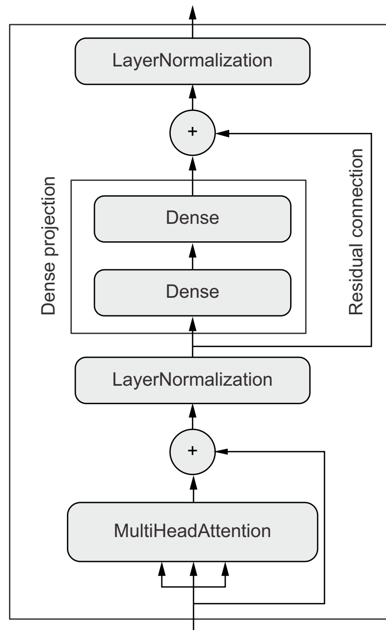


Figure 11.9 The `TransformerEncoder` chains a `MultiHeadAttention` layer with a dense projection and adds normalization as well as residual connections.

The original Transformer architecture consists of two parts: a *Transformer encoder* that processes the source sequence, and a *Transformer decoder* that uses the source sequence to generate a translated version. You'll learn about about the decoder part in a minute.

Crucially, the encoder part can be used for text classification—it's a very generic module that ingests a sequence and learns to turn it into a more useful representation. Let's implement a Transformer encoder and try it on the movie review sentiment classification task.

Listing 11.21 Transformer encoder implemented as a subclassed `Layer`

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

class TransformerEncoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim          # Size of the input token vectors
        self.dense_dim = dense_dim          # Size of the inner dense layer
        self.num_heads = num_heads          # Number of attention heads
        self.attention = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
        self.dense_proj = keras.Sequential(
            [layers.Dense(dense_dim, activation="relu"),
             layers.Dense(embed_dim),]
        )
```

```
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()

    def call(self, inputs, mask=None):
        if mask is not None:
            mask = mask[:, tf.newaxis, :]
        attention_output = self.attention(
            inputs, inputs, attention_mask=mask)
        proj_input = self.layernorm_1(inputs + attention_output)
        proj_output = self.dense_proj(proj_input)
        return self.layernorm_2(proj_input + proj_output)

    def get_config(self):
        config = super().get_config()
        config.update({
            "embed_dim": self.embed_dim,
            "num_heads": self.num_heads,
            "dense_dim": self.dense_dim,
        })
        return config
```

Computation goes in call().

The mask that will be generated by the Embedding layer will be 2D, but the attention layer expects to be 3D or 4D, so we expand its rank.

Implement serialization so we can save the model.

Saving custom layers

When you write custom layers, make sure to implement the `get_config` method: this enables the layer to be reinstantiated from its config dict, which is useful during model saving and loading. The method should return a Python dict that contains the values of the constructor arguments used to create the layer.

All Keras layers can be serialized and deserialized as follows:

```
config = layer.get_config()
new_layer = layer.__class__.from_config(config)
```

The config does not contain weight values, so all weights in the layer get initialized from scratch.

For instance:

```
layer = PositionalEmbedding(sequence_length, input_dim, output_dim)
config = layer.get_config()
new_layer = PositionalEmbedding.from_config(config)
```

When saving a model that contains custom layers, the savefile will contain these config dicts. When loading the model from the file, you should provide the custom layer classes to the loading process, so that it can make sense of the config objects:

```
model = keras.models.load_model(
    filename, custom_objects={"PositionalEmbedding": PositionalEmbedding})
```

You'll note that the normalization layers we're using here aren't `BatchNormalization` layers like those we've used before in image models. That's because `BatchNormalization` doesn't work well for sequence data. Instead, we're using the `LayerNormalization` layer, which normalizes each sequence independently from other sequences in the batch. Like this, in NumPy-like pseudocode:

```
def layer_normalization(batch_of_sequences):
    mean = np.mean(batch_of_sequences, keepdims=True, axis=-1)
    variance = np.var(batch_of_sequences, keepdims=True, axis=-1)
    return (batch_of_sequences - mean) / variance
```

Input shape: (batch_size, sequence_length, embedding_dim)

To compute mean and variance, we only pool data over the last axis (axis -1).

Compare to `BatchNormalization` (during training):

```
def batch_normalization(batch_of_images):
    mean = np.mean(batch_of_images, keepdims=True, axis=(0, 1, 2))
    variance = np.var(batch_of_images, keepdims=True, axis=(0, 1, 2))
    return (batch_of_images - mean) / variance
```

Input shape: (batch_size, height, width, channels)

Pool data over the batch axis (axis 0), which creates interactions between samples in a batch.

While `BatchNormalization` collects information from many samples to obtain accurate statistics for the feature means and variances, `LayerNormalization` pools data within each sequence separately, which is more appropriate for sequence data.

Now that we've implemented our `TransformerEncoder`, we can use it to assemble a text-classification model similar to the GRU-based one you've seen previously.

Listing 11.22 Using the Transformer encoder for text classification

```
vocab_size = 20000
embed_dim = 256
num_heads = 2
dense_dim = 32

inputs = keras.Input(shape=(None,), dtype="int64")
x = layers.Embedding(vocab_size, embed_dim)(inputs)
x = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()
```

Since TransformerEncoder returns full sequences, we need to reduce each sequence to a single vector for classification via a global pooling layer.

Let's train it. It gets to 87.5% test accuracy—slightly worse than the GRU model.

Listing 11.23 Training and evaluating the Transformer encoder based model

```
callbacks = [
    keras.callbacks.ModelCheckpoint("transformer_encoder.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=20,
          callbacks=callbacks)
```

```
model = keras.models.load_model(
    "transformer_encoder.keras",
    custom_objects={"TransformerEncoder": TransformerEncoder})
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

Provide the custom TransformerEncoder class to the model-loading process.

At this point, you should start to feel a bit uneasy. Something's off here. Can you tell what it is?

This section is ostensibly about "sequence models." I started off by highlighting the importance of word order. I said that Transformer was a sequence-processing architecture, originally developed for machine translation. And yet . . . the Transformer encoder you just saw in action wasn't a sequence model at all. Did you notice? It's composed of dense layers that process sequence tokens independently from each other, and an attention layer that looks at the tokens *as a set*. You could change the order of the tokens in a sequence, and you'd get the exact same pairwise attention scores and the exact same context-aware representations. If you were to completely scramble the words in every movie review, the model wouldn't notice, and you'd still get the exact same accuracy. Self-attention is a set-processing mechanism, focused on the relationships between pairs of sequence elements (see figure 11.10)—it's blind to whether these elements occur at the beginning, at the end, or in the middle of a sequence. So why do we say that Transformer is a sequence model? And how could it possibly be good for machine translation if it doesn't look at word order?

	Word order awareness	Context awareness (cross-words interactions)
Bag-of-unigrams	No	No
Bag-of-bigrams	Very limited	No
RNN	Yes	No
Self-attention	No	Yes
Transformer	Yes	Yes

Figure 11.10 Features of different types of NLP models

I hinted at the solution earlier in the chapter: I mentioned in passing that Transformer was a hybrid approach that is technically order-agnostic, but that manually injects order information in the representations it processes. This is the missing ingredient! It's called *positional encoding*. Let's take a look.

USING POSITIONAL ENCODING TO RE-INJECT ORDER INFORMATION

The idea behind positional encoding is very simple: to give the model access to word-order information, we're going to add the word's position in the sentence to each word embedding. Our input word embeddings will have two components: the usual word

vector, which represents the word independently of any specific context, and a position vector, which represents the position of the word in the current sentence. Hopefully, the model will then figure out how to best leverage this additional information.

The simplest scheme you could come up with would be to concatenate the word's position to its embedding vector. You'd add a "position" axis to the vector and fill it with 0 for the first word in the sequence, 1 for the second, and so on.

That may not be ideal, however, because the positions can potentially be very large integers, which will disrupt the range of values in the embedding vector. As you know, neural networks don't like very large input values, or discrete input distributions.

The original "Attention is all you need" paper used an interesting trick to encode word positions: it added to the word embeddings a vector containing values in the range `[-1, 1]` that varied cyclically depending on the position (it used cosine functions to achieve this). This trick offers a way to uniquely characterize any integer in a large range via a vector of small values. It's clever, but it's not what we're going to use in our case. We'll do something simpler and more effective: we'll learn position-embedding vectors the same way we learn to embed word indices. We'll then proceed to add our position embeddings to the corresponding word embeddings, to obtain a position-aware word embedding. This technique is called "positional embedding." Let's implement it.

Listing 11.24 Implementing positional embedding as a subclassed layer

```
class PositionalEmbedding(layers.Layer):
    def __init__(self, sequence_length, input_dim, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.token_embeddings = layers.Embedding(
            input_dim=input_dim, output_dim=output_dim)
        self.position_embeddings = layers.Embedding(
            input_dim=sequence_length, output_dim=output_dim)
        self.sequence_length = sequence_length
        self.input_dim = input_dim
        self.output_dim = output_dim

    def call(self, inputs):
        length = tf.shape(inputs)[-1]
        positions = tf.range(start=0, limit=length, delta=1)
        embedded_tokens = self.token_embeddings(inputs)
        embedded_positions = self.position_embeddings(positions)
        return embedded_tokens + embedded_positions

    def compute_mask(self, inputs, mask=None):
        return tf.math.not_equal(inputs, 0)

    def get_config(self):
        config = super().get_config()
        config.update({
            "output_dim": self.output_dim,
```

A downside of position embeddings is that the sequence length needs to be known in advance.

Prepare an Embedding layer for the token indices.

And another one for the token positions

Add both embedding vectors together.

Like the Embedding layer, this layer should be able to generate a mask so we can ignore padding 0s in the inputs. The compute_mask method will called automatically by the framework, and the mask will get propagated to the next layer.

Implement serialization so we can save the model.

```
            "sequence_length": self.sequence_length,
            "input_dim": self.input_dim,
        })
        return config
```

You would use this PositionEmbedding layer just like a regular Embedding layer. Let's see it in action!

PUTTING IT ALL TOGETHER: A TEXT-CLASSIFICATION TRANSFORMER

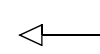
All you have to do to start taking word order into account is swap the old Embedding layer with our position-aware version.

Listing 11.25 Combining the Transformer encoder with positional embedding

```
vocab_size = 20000
sequence_length = 600
embed_dim = 256
num_heads = 2
dense_dim = 32

inputs = keras.Input(shape=(None,), dtype="int64")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(inputs)
x = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()

callbacks = [
    keras.callbacks.ModelCheckpoint("full_transformer_encoder.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=20,
     callbacks=callbacks)
model = keras.models.load_model(
    "full_transformer_encoder.keras",
    custom_objects={"TransformerEncoder": TransformerEncoder,
                    "PositionalEmbedding": PositionalEmbedding})
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```


Look here!

We get to 88.3% test accuracy, a solid improvement that clearly demonstrates the value of word order information for text classification. This is our best sequence model so far! However, it's still one notch below the bag-of-words approach.

11.4.4 When to use sequence models over bag-of-words models

You may sometimes hear that bag-of-words methods are outdated, and that Transformer-based sequence models are the way to go, no matter what task or dataset you're looking at. This is definitely not the case: a small stack of `Dense` layers on top of a bag-of-bigrams remains a perfectly valid and relevant approach in many cases. In fact, among the various techniques that we've tried on the IMDB dataset throughout this chapter, the best performing so far was the bag-of-bigrams!

So, when should you prefer one approach over the other?

In 2017, my team and I ran a systematic analysis of the performance of various text-classification techniques across many different types of text datasets, and we discovered a remarkable and surprising rule of thumb for deciding whether to go with a bag-of-words model or a sequence model (http://mng.bz/AOzK)—a golden constant of sorts.

It turns out that when approaching a new text-classification task, you should pay close attention to the ratio between the number of samples in your training data and the mean number of words per sample (see figure 11.11). If that ratio is small—less than 1,500—then the bag-of-bigrams model will perform better (and as a bonus, it will be much faster to train and to iterate on too). If that ratio is higher than 1,500, then you should go with a sequence model. In other words, sequence models work best when lots of training data is available and when each sample is relatively short.

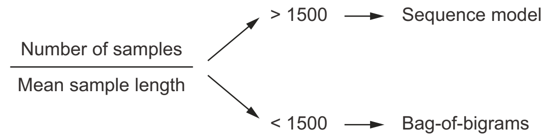


Figure 11.11 A simple heuristic for selecting a text-classification model: the ratio between the number of training samples and the mean number of words per sample

So if you're classifying 1,000-word long documents, and you have 100,000 of them (a ratio of 100), you should go with a bigram model. If you're classifying tweets that are 40 words long on average, and you have 50,000 of them (a ratio of 1,250), you should also go with a bigram model. But if you increase your dataset size to 500,000 tweets (a ratio of 12,500), go with a Transformer encoder. What about the IMDB movie review classification task? We had 20,000 training samples and an average word count of 233, so our rule of thumb points toward a bigram model, which confirms what we found in practice.

This intuitively makes sense: the input of a sequence model represents a richer and more complex space, and thus it takes more data to map out that space; meanwhile, a plain set of terms is a space so simple that you can train a logistic regression on top using just a few hundreds or thousands of samples. In addition, the shorter a sample is, the less the model can afford to discard any of the information it contains—in particular, word order becomes more important, and discarding it can create ambiguity. The sentences "this movie is the bomb" and "this movie was a bomb" have very

close unigram representations, which could confuse a bag-of-words model, but a sequence model could tell which one is negative and which one is positive. With a longer sample, word statistics would become more reliable and the topic or sentiment would be more apparent from the word histogram alone.

Now, keep in mind that this heuristic rule was developed specifically for text classification. It may not necessarily hold for other NLP tasks—when it comes to machine translation, for instance, Transformer shines especially for very long sequences, compared to RNNs. Our heuristic is also just a rule of thumb, rather than a scientific law, so expect it to work most of the time, but not necessarily every time.

11.5 Beyond text classification: Sequence-to-sequence learning

You now possess all of the tools you will need to tackle most natural language processing tasks. However, you've only seen these tools in action on a single problem: text classification. This is an extremely popular use case, but there's a lot more to NLP than classification. In this section, you'll deepen your expertise by learning about *sequence-to-sequence models*.

A sequence-to-sequence model takes a sequence as input (often a sentence or paragraph) and translates it into a different sequence. This is the task at the heart of many of the most successful applications of NLP:

- *Machine translation*—Convert a paragraph in a source language to its equivalent in a target language.
- *Text summarization*—Convert a long document to a shorter version that retains the most important information.
- *Question answering*—Convert an input question into its answer.
- *Chatbots*—Convert a dialogue prompt into a reply to this prompt, or convert the history of a conversation into the next reply in the conversation.
- *Text generation*—Convert a text prompt into a paragraph that completes the prompt.
- Etc.

The general template behind sequence-to-sequence models is described in figure 11.12. During training,

- An *encoder* model turns the source sequence into an intermediate representation.
- A *decoder* is trained to predict the next token `i` in the target sequence by looking at both previous tokens (`0` to `i - 1`) and the encoded source sequence.

During inference, we don't have access to the target sequence—we're trying to predict it from scratch. We'll have to generate it one token at a time:

1. We obtain the encoded source sequence from the encoder.
2. The decoder starts by looking at the encoded source sequence as well as an initial "seed" token (such as the string `"[start]"`), and uses them to predict the first real token in the sequence.

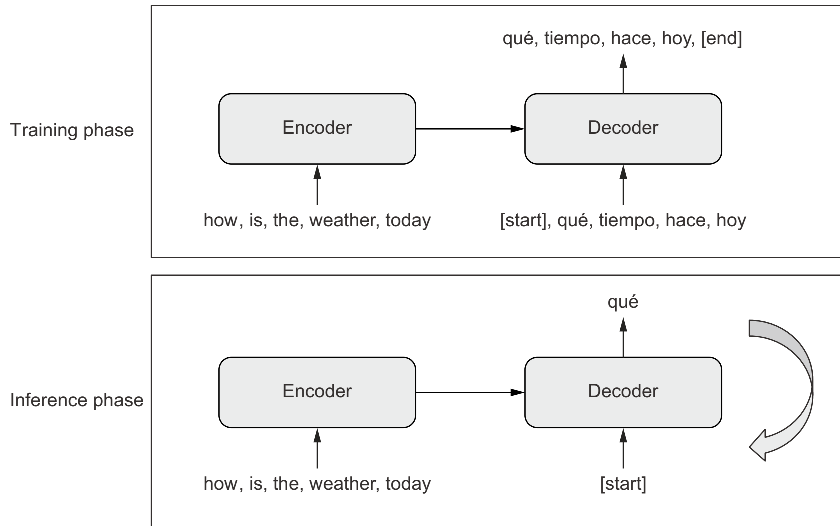


Figure 11.12 Sequence-to-sequence learning: the source sequence is processed by the encoder and is then sent to the decoder. The decoder looks at the target sequence so far and predicts the target sequence offset by one step in the future. During inference, we generate one target token at a time and feed it back into the decoder.

3. The predicted sequence so far is fed back into the decoder, which generates the next token, and so on, until it generates a stop token (such as the string `"[end]"`).

Everything you've learned so far can be repurposed to build this new kind of model. Let's dive in.

11.5.1 A machine translation example

We'll demonstrate sequence-to-sequence modeling on a machine translation task. Machine translation is precisely what Transformer was developed for! We'll start with a recurrent sequence model, and we'll follow up with the full Transformer architecture.

We'll be working with an English-to-Spanish translation dataset available at www.manythings.org/anki/. Let's download it:

```
!wget http://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip
!unzip -q spa-eng.zip
```

The text file contains one example per line: an English sentence, followed by a tab character, followed by the corresponding Spanish sentence. Let's parse this file.

```
text_file = "spa-eng/spa.txt"
with open(text_file) as f:
    lines = f.read().split("\n")[:-1]
text_pairs = []
```

```
for line in lines:
    english, spanish = line.split("\t")
    spanish = "[start] " + spanish + " [end]"
    text_pairs.append((english, spanish))
```

Iterate over the lines in the file.

Each line contains an English phrase and its Spanish translation, tab-separated.

We prepend "[start]" and append "[end]" to the Spanish sentence, to match the template from figure 11.12.

Our text_pairs look like this:

```
>>> import random
>>> print(random.choice(text_pairs))
("Soccer is more popular than tennis.",
 "[start] El fútbol es más popular que el tenis. [end]")
```

Let's shuffle them and split them into the usual training, validation, and test sets:

```
import random
random.shuffle(text_pairs)
num_val_samples = int(0.15 * len(text_pairs))
num_train_samples = len(text_pairs) - 2 * num_val_samples
train_pairs = text_pairs[:num_train_samples]
val_pairs = text_pairs[num_train_samples:num_train_samples + num_val_samples]
test_pairs = text_pairs[num_train_samples + num_val_samples:]
```

Next, let's prepare two separate TextVectorization layers: one for English and one for Spanish. We're going to need to customize the way strings are preprocessed:

- We need to preserve the "[start]" and "[end]" tokens that we've inserted. By default, the characters [and] would be stripped, but we want to keep them around so we can tell apart the word "start" and the start token "[start]".
- Punctuation is different from language to language! In the Spanish TextVectorization layer, if we're going to strip punctuation characters, we need to also strip the character ¿.

Note that for a non-toy translation model, we would treat punctuation characters as separate tokens rather than stripping them, since we would want to be able to generate correctly punctuated sentences. In our case, for simplicity, we'll get rid of all punctuation.

Listing 11.26 Vectorizing the English and Spanish text pairs

```
import tensorflow as tf
import string
import re

strip_chars = string.punctuation + "¿"
strip_chars = strip_chars.replace("[", "")
strip_chars = strip_chars.replace("]", "")

def custom_standardization(input_string):
    lowercase = tf.strings.lower(input_string)
    return tf.strings.regex_replace(
        lowercase, f"[{re.escape(strip_chars)}]", "")
```

Prepare a custom string standardization function for the Spanish TextVectorization layer: it preserves [and] but strips ¿ (as well as all other characters from strings.punctuation).

```
vocab_size = 15000
sequence_length = 20

source_vectorization = layers.TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length,
)
target_vectorization = layers.TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length + 1,
    standardize=custom_standardization,
)
train_english_texts = [pair[0] for pair in train_pairs]
train_spanish_texts = [pair[1] for pair in train_pairs]
source_vectorization.adapt(train_english_texts)
target_vectorization.adapt(train_spanish_texts)
```

To keep things simple, we'll only look at the top 15,000 words in each language, and we'll restrict sentences to 20 words.

The English layer

The Spanish layer

Generate Spanish sentences that have one extra token, since we'll need to offset the sentence by one step during training.

Learn the vocabulary of each language.

Finally, we can turn our data into a `tf.data` pipeline. We want it to return a tuple `(inputs, target)` where `inputs` is a dict with two keys, "encoder_inputs" (the English sentence) and "decoder_inputs" (the Spanish sentence), and `target` is the Spanish sentence offset by one step ahead.

Listing 11.27 Preparing datasets for the translation task

```
batch_size = 64

def format_dataset(eng, spa):
    eng = source_vectorization(eng)
    spa = target_vectorization(spa)
    return ({
        "english": eng,
        "spanish": spa[:, :-1],
    }, spa[:, 1:])

def make_dataset(pairs):
    eng_texts, spa_texts = zip(*pairs)
    eng_texts = list(eng_texts)
    spa_texts = list(spa_texts)
    dataset = tf.data.Dataset.from_tensor_slices((eng_texts, spa_texts))
    dataset = dataset.batch(batch_size)
    dataset = dataset.map(format_dataset, num_parallel_calls=4)
    return dataset.shuffle(2048).prefetch(16).cache()

train_ds = make_dataset(train_pairs)
val_ds = make_dataset(val_pairs)
```

The input Spanish sentence doesn't include the last token to keep inputs and targets at the same length.

The target Spanish sentence is one step ahead. Both are still the same length (20 words).

Use in-memory caching to speed up preprocessing.

Here's what our dataset outputs look like:

```
>>> for inputs, targets in train_ds.take(1):
>>>     print(f"inputs['english'].shape: {inputs['english'].shape}")
>>>     print(f"inputs['spanish'].shape: {inputs['spanish'].shape}")
```

```
>>>     print(f"targets.shape: {targets.shape}")
inputs["encoder_inputs"].shape: (64, 20)
inputs["decoder_inputs"].shape: (64, 20)
targets.shape: (64, 20)
```

The data is now ready—time to build some models. We'll start with a recurrent sequence-to-sequence model before moving on to a Transformer.

11.5.2 Sequence-to-sequence learning with RNNs

Recurrent neural networks dominated sequence-to-sequence learning from 2015–2017 before being overtaken by Transformer. They were the basis for many real-world machine-translation systems—as mentioned in chapter 10, Google Translate circa 2017 was powered by a stack of seven large LSTM layers. It's still worth learning about this approach today, as it provides an easy entry point to understanding sequence-to-sequence models.

The simplest, naive way to use RNNs to turn a sequence into another sequence is to keep the output of the RNN at each time step. In Keras, it would look like this:

```
inputs = keras.Input(shape=(sequence_length,), dtype="int64")
x = layers.Embedding(input_dim=vocab_size, output_dim=128)(inputs)
x = layers.LSTM(32, return_sequences=True)(x)
outputs = layers.Dense(vocab_size, activation="softmax")(x)
model = keras.Model(inputs, outputs)
```

However, there are two major issues with this approach:

- The target sequence must always be the same length as the source sequence. In practice, this is rarely the case. Technically, this isn't critical, as you could always pad either the source sequence or the target sequence to make their lengths match.
- Due to the step-by-step nature of RNNs, the model will only be looking at tokens $0...N$ in the source sequence in order to predict token N in the target sequence. This constraint makes this setup unsuitable for most tasks, and particularly translation. Consider translating "The weather is nice today" to French—that would be "Il fait beau aujourd'hui." You'd need to be able to predict "Il" from just "The," "Il fait" from just "The weather," etc., which is simply impossible.

If you're a human translator, you'd start by reading the entire source sentence before starting to translate it. This is especially important if you're dealing with languages that have wildly different word ordering, like English and Japanese. And that's exactly what standard sequence-to-sequence models do.

In a proper sequence-to-sequence setup (see figure 11.13), you would first use an RNN (the encoder) to turn the entire source sequence into a single vector (or set of vectors). This could be the last output of the RNN, or alternatively, its final internal state vectors. Then you would use this vector (or vectors) as the *initial state* of another

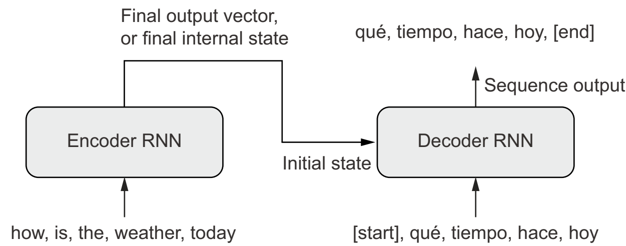


Figure 11.13 A sequence-to-sequence RNN: an RNN encoder is used to produce a vector that encodes the entire source sequence, which is used as the initial state for an RNN decoder.

RNN (the decoder), which would look at elements 0...*N* in the target sequence, and try to predict step *N*+1 in the target sequence.

Let's implement this in Keras with GRU-based encoders and decoders. The choice of GRU rather than LSTM makes things a bit simpler, since GRU only has a single state vector, whereas LSTM has multiple. Let's start with the encoder.

Listing 11.28 GRU-based encoder

```
from tensorflow import keras
from tensorflow.keras import layers

embed_dim = 256
latent_dim = 1024

source = keras.Input(shape=(None,), dtype="int64", name="english")
x = layers.Embedding(vocab_size, embed_dim, mask_zero=True)(source)
encoded_source = layers.Bidirectional(
    layers.GRU(latent_dim), merge_mode="sum")(x)
```

- Don't forget masking: it's critical in this setup.
- The English source sentence goes here. Specifying the name of the input enables us to fit() the model with a dict of inputs.
- Our encoded source sentence is the last output of a bidirectional GRU.

Next, let's add the decoder—a simple GRU layer that takes as its initial state the encoded source sentence. On top of it, we add a `Dense` layer that produces for each output step a probability distribution over the Spanish vocabulary.

Listing 11.29 GRU-based decoder and the end-to-end model

```
past_target = keras.Input(shape=(None,), dtype="int64", name="spanish")
x = layers.Embedding(vocab_size, embed_dim, mask_zero=True)(past_target)
decoder_gru = layers.GRU(latent_dim, return_sequences=True)
x = decoder_gru(x, initial_state=encoded_source)
x = layers.Dropout(0.5)(x)
target_next_step = layers.Dense(vocab_size, activation="softmax")(x)
seq2seq_rnn = keras.Model([source, past_target], target_next_step)
```

- The Spanish target sentence goes here.
- Don't forget masking.
- The encoded source sentence serves as the initial state of the decoder GRU.
- Predicts the next token
- End-to-end model: maps the source sentence and the target sentence to the target sentence one step in the future

During training, the decoder takes as input the entire target sequence, but thanks to the step-by-step nature of RNNs, it only looks at tokens 0...*N* in the input to predict

token *N* in the output (which corresponds to the next token in the sequence, since the output is intended to be offset by one step). This means we only use information from the past to predict the future, as we should; otherwise we'd be cheating, and our model would not work at inference time.

Let's start training.

Listing 11.30 Training our recurrent sequence-to-sequence model

```
seq2seq_rnn.compile(
    optimizer="rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])
seq2seq_rnn.fit(train_ds, epochs=15, validation_data=val_ds)
```

We picked accuracy as a crude way to monitor validation-set performance during training. We get to 64% accuracy: on average, the model predicts the next word in the Spanish sentence correctly 64% of the time. However, in practice, next-token accuracy isn't a great metric for machine translation models, in particular because it makes the assumption that the correct target tokens from 0 to *N* are already known when predicting token *N*+1. In reality, during inference, you're generating the target sentence from scratch, and you can't rely on previously generated tokens being 100% correct. If you work on a real-world machine translation system, you will likely use "BLEU scores" to evaluate your models—a metric that looks at entire generated sequences and that seems to correlate well with human perception of translation quality.

At last, let's use our model for inference. We'll pick a few sentences in the test set and check how our model translates them. We'll start from the seed token, `"[start]"`, and feed it into the decoder model, together with the encoded English source sentence. We'll retrieve a next-token prediction, and we'll re-inject it into the decoder repeatedly, sampling one new target token at each iteration, until we get to `"[end]"` or reach the maximum sentence length.

Listing 11.31 Translating new sentences with our RNN encoder and decoder

```
import numpy as np
spa_vocab = target_vectorization.get_vocabulary()
spa_index_lookup = dict(zip(range(len(spa_vocab)), spa_vocab))
max_decoded_sentence_length = 20

def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization([decoded_sentence])
        next_token_predictions = seq2seq_rnn.predict(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(next_token_predictions[0, i, :])
```

Prepare a dict to convert token index predictions to string tokens.

Seed token

Sample the next token.

```
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence

test_eng_texts = [pair[0] for pair in test_pairs]
for _ in range(20):
    input_sentence = random.choice(test_eng_texts)
    print("-")
    print(input_sentence)
    print(decode_sequence(input_sentence))
```

Convert the next token prediction to a string and append it to the generated sentence.

Exit condition: either hit max length or sample a stop character

Note that this inference setup, while very simple, is rather inefficient, since we reprocess the entire source sentence and the entire generated target sentence every time we sample a new word. In a practical application, you'd factor the encoder and the decoder as two separate models, and your decoder would only run a single step at each token-sampling iteration, reusing its previous internal state.

Here are our translation results. Our model works decently well for a toy model, though it still makes many basic mistakes.

Listing 11.32 Some sample results from the recurrent translation model

```
Who is in this room?
[start] quién está en esta habitación [end]
-
That doesn't sound too dangerous.
[start] eso no es muy difícil [end]
-
No one will stop me.
[start] nadie me va a hacer [end]
-
Tom is friendly.
[start] tom es un buen [UNK] [end]
```

There are many ways this toy model could be improved: We could use a deep stack of recurrent layers for both the encoder and the decoder (note that for the decoder, this makes state management a bit more involved). We could use an LSTM instead of a GRU. And so on. Beyond such tweaks, however, the RNN approach to sequence-to-sequence learning has a few fundamental limitations:

- The source sequence representation has to be held entirely in the encoder state vector(s), which puts significant limitations on the size and complexity of the sentences you can translate. It's a bit as if a human were translating a sentence entirely from memory, without looking twice at the source sentence while producing the translation.
- RNNs have trouble dealing with very long sequences, since they tend to progressively forget about the past—by the time you've reached the 100th token in either sequence, little information remains about the start of the sequence.

That means RNN-based models can't hold onto long-term context, which can be essential for translating long documents.

These limitations are what has led the machine learning community to embrace the Transformer architecture for sequence-to-sequence problems. Let's take a look.

11.5.3 Sequence-to-sequence learning with Transformer

Sequence-to-sequence learning is the task where Transformer really shines. Neural attention enables Transformer models to successfully process sequences that are considerably longer and more complex than those RNNs can handle.

As a human translating English to Spanish, you're not going to read the English sentence one word at a time, keep its meaning in memory, and then generate the Spanish sentence one word at a time. That may work for a five-word sentence, but it's unlikely to work for an entire paragraph. Instead, you'll probably want to go back and forth between the source sentence and your translation in progress, and pay attention to different words in the source as you're writing down different parts of your translation.

That's exactly what you can achieve with neural attention and Transformers. You're already familiar with the Transformer encoder, which uses self-attention to produce context-aware representations of each token in an input sequence. In a sequence-to-sequence Transformer, the Transformer encoder would naturally play the role of the encoder, which reads the source sequence and produces an encoded representation of it. Unlike our previous RNN encoder, though, the Transformer encoder keeps the encoded representation in a sequence format: it's a sequence of context-aware embedding vectors.

The second half of the model is the *Transformer decoder*. Just like the RNN decoder, it reads tokens 0…*N* in the target sequence and tries to predict token *N*+1. Crucially, while doing this, it uses neural attention to identify which tokens in the encoded source sentence are most closely related to the target token it's currently trying to predict—perhaps not unlike what a human translator would do. Recall the query-key-value model: in a Transformer decoder, the target sequence serves as an attention "query" that is used to to pay closer attention to different parts of the source sequence (the source sequence plays the roles of both keys and values).

THE TRANSFORMER DECODER

Figure 11.14 shows the full sequence-to-sequence Transformer. Look at the decoder internals: you'll recognize that it looks very similar to the Transformer encoder, except that an extra attention block is inserted between the self-attention block applied to the target sequence and the dense layers of the exit block.

Let's implement it. Like for the `TransformerEncoder`, we'll use a `Layer` subclass. Before we focus on the `call()`, method, where the action happens, let's start by defining the class constructor, containing the layers we're going to need.

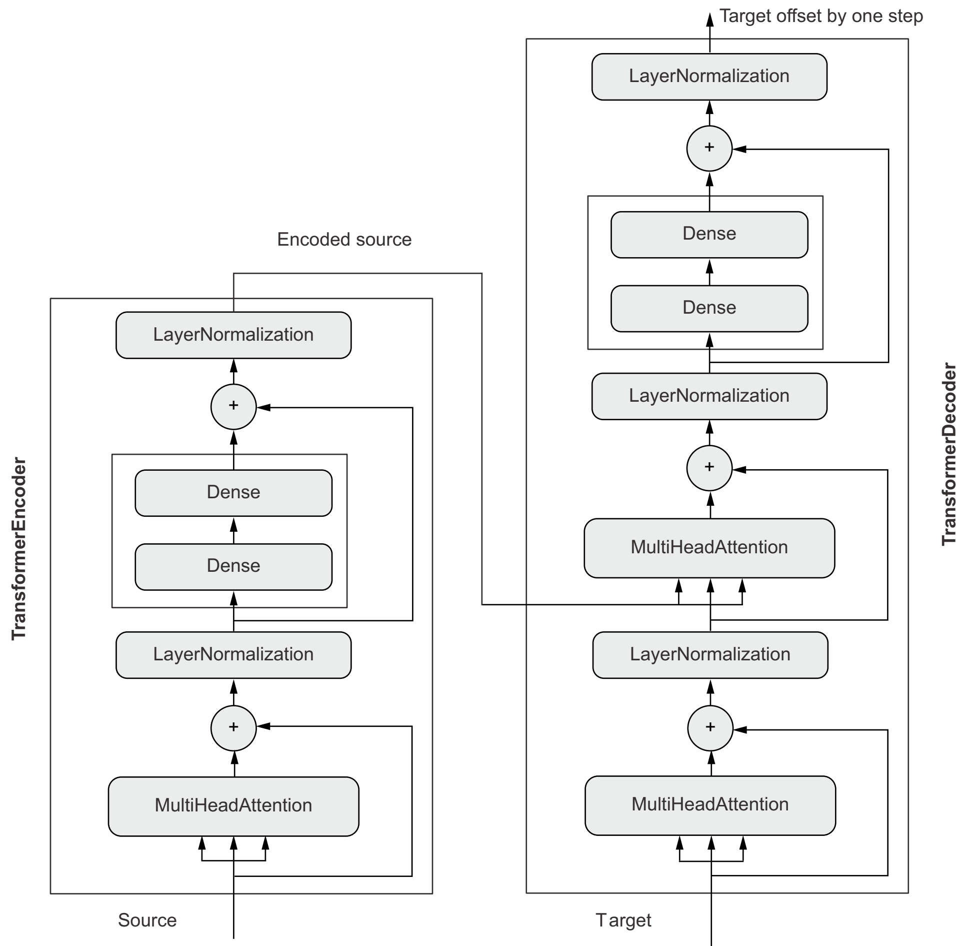


Figure 11.14 The `TransformerDecoder` is similar to the `TransformerEncoder`, except it features an additional attention block where the keys and values are the source sequence encoded by the `TransformerEncoder`. Together, the encoder and the decoder form an end-to-end Transformer.

Listing 11.33 The `TransformerDecoder`

```
class TransformerDecoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        self.attention_1 = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
        self.attention_2 = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
        self.dense_proj = keras.Sequential(
```

```
            [layers.Dense(dense_dim, activation="relu"),
             layers.Dense(embed_dim),]
        )
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()
        self.layernorm_3 = layers.LayerNormalization()
        self.supports_masking = True

    def get_config(self):
        config = super().get_config()
        config.update({
            "embed_dim": self.embed_dim,
            "num_heads": self.num_heads,
            "dense_dim": self.dense_dim,
        })
        return config
```

This attribute ensures that the layer will propagate its input mask to its outputs; masking in Keras is explicitly opt-in. If you pass a mask to a layer that doesn't implement compute_mask() and that doesn't expose this supports_masking attribute, that's an error.

The `call()` method is almost a straightforward rendering of the connectivity diagram from figure 11.14. But there's an additional detail we need to take into account: *causal padding*. Causal padding is absolutely critical to successfully training a sequence-to-sequence Transformer. Unlike an RNN, which looks at its input one step at a time, and thus will only have access to steps 0...*N* to generate output step *N* (which is token *N*+1 in the target sequence), the `TransformerDecoder` is order-agnostic: it looks at the entire target sequence at once. If it were allowed to use its entire input, it would simply learn to copy input step *N*+1 to location *N* in the output. The model would thus achieve perfect training accuracy, but of course, when running inference, it would be completely useless, since input steps beyond *N* aren't available.

The fix is simple: we'll mask the upper half of the pairwise attention matrix to prevent the model from paying any attention to information from the future—only information from tokens 0...*N* in the target sequence should be used when generating target token *N*+1. To do this, we'll add a `get_causal_attention_mask(self, inputs)` method to our `TransformerDecoder` to retrieve an attention mask that we can pass to our `MultiHeadAttention` layers.

Listing 11.34 `TransformerDecoder` method that generates a causal mask

Generate matrix of shape (sequence_length, sequence_length) with 1s in one half and 0s in the other.

```
    def get_causal_attention_mask(self, inputs):
        input_shape = tf.shape(inputs)
        batch_size, sequence_length = input_shape[0], input_shape[1]
        i = tf.range(sequence_length)[:, tf.newaxis]
        j = tf.range(sequence_length)
        mask = tf.cast(i >= j, dtype="int32")
        mask = tf.reshape(mask, (1, input_shape[1], input_shape[1]))
        mult = tf.concat(
            [tf.expand_dims(batch_size, -1),
             tf.constant([1, 1], dtype=tf.int32)], axis=0)
        return tf.tile(mask, mult)
```

Replicate it along the batch axis to get a matrix of shape (batch_size, sequence_length, sequence_length).

Now we can write down the full `call()` method implementing the forward pass of the decoder.

Listing 11.35 The forward pass of the `TransformerDecoder`

```
    def call(self, inputs, encoder_outputs, mask=None):
        causal_mask = self.get_causal_attention_mask(inputs)
        if mask is not None:
            padding_mask = tf.cast(
                mask[:, tf.newaxis, :], dtype="int32")
            padding_mask = tf.minimum(padding_mask, causal_mask)
        attention_output_1 = self.attention_1(
            query=inputs,
            value=inputs,
            key=inputs,
            attention_mask=causal_mask)
        attention_output_1 = self.layernorm_1(inputs + attention_output_1)
        attention_output_2 = self.attention_2(
            query=attention_output_1,
            value=encoder_outputs,
            key=encoder_outputs,
            attention_mask=padding_mask,
        )
        attention_output_2 = self.layernorm_2(
            attention_output_1 + attention_output_2)
        proj_output = self.dense_proj(attention_output_2)
        return self.layernorm_3(attention_output_2 + proj_output)
```

Annotations:
- Retrieve the causal mask.
- Prepare the input mask (that describes padding locations in the target sequence).
- Merge the two masks together.
- Pass the causal mask to the first attention layer, which performs self-attention over the target sequence.
- Pass the combined mask to the second attention layer, which relates the source sequence to the target sequence.

Putting it all together: A Transformer for machine translation

The end-to-end Transformer is the model we'll be training. It maps the source sequence and the target sequence to the target sequence one step in the future. It straightforwardly combines the pieces we've built so far: `PositionalEmbedding` layers, the `TransformerEncoder`, and the `TransformerDecoder`. Note that both the `TransformerEncoder` and the `TransformerDecoder` are shape-invariant, so you could be stacking many of them to create a more powerful encoder or decoder. In our example, we'll stick to a single instance of each.

Listing 11.36 End-to-end Transformer

```
embed_dim = 256
dense_dim = 2048
num_heads = 8

encoder_inputs = keras.Input(shape=(None,), dtype="int64", name="english")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(encoder_inputs)
encoder_outputs = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)

decoder_inputs = keras.Input(shape=(None,), dtype="int64", name="spanish")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(decoder_inputs)
x = TransformerDecoder(embed_dim, dense_dim, num_heads)(x, encoder_outputs)
x = layers.Dropout(0.5)(x)
```

Annotations:
- Encode the source sentence.
- Encode the target sentence and combine it with the encoded source sentence.

```
decoder_outputs = layers.Dense(vocab_size, activation="softmax")(x)
transformer = keras.Model([encoder_inputs, decoder_inputs], decoder_outputs)
```

Predict a word for each output position.

We're now ready to train our model—we get to 67% accuracy, a good deal above the GRU-based model.

Listing 11.37 Training the sequence-to-sequence Transformer

```
transformer.compile(
    optimizer="rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])
transformer.fit(train_ds, epochs=30, validation_data=val_ds)
```

Finally, let's try using our model to translate never-seen-before English sentences from the test set. The setup is identical to what we used for the sequence-to-sequence RNN model.

Listing 11.38 Translating new sentences with our Transformer model

```
import numpy as np
spa_vocab = target_vectorization.get_vocabulary()
spa_index_lookup = dict(zip(range(len(spa_vocab)), spa_vocab))
max_decoded_sentence_length = 20

def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization(
            [decoded_sentence])[:, :-1]
        predictions = transformer(
            [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence

test_eng_texts = [pair[0] for pair in test_pairs]
for _ in range(20):
    input_sentence = random.choice(test_eng_texts)
    print("-")
    print(input_sentence)
    print(decode_sequence(input_sentence))
```

Sample the next token.

Convert the next token prediction to a string, and append it to the generated sentence.

Exit condition

Subjectively, the Transformer seems to perform significantly better than the GRU-based translation model. It's still a toy model, but it's a better toy model.

Listing 11.39 Some sample results from the Transformer translation model

```
This is a song I learned when I was a kid.
[start] esta es una canción que aprendí cuando era chico [end]
-
She can play the piano.
[start] ella puede tocar piano [end]
-
I'm not who you think I am.
[start] no soy la persona que tú creo que soy [end]
-
It may have rained a little last night.
[start] puede que llueve un poco el pasado [end]
```

While the source sentence wasn't gendered, this translation assumes a male speaker. Keep in mind that translation models will often make unwarranted assumptions about their input data, which leads to algorithmic bias. In the worst cases, a model might hallucinate memorized information that has nothing to do with the data it's currently processing.

That concludes this chapter on natural language processing—you just went from the very basics to a fully fledged Transformer that can translate from English to Spanish. Teaching machines to make sense of language is the latest superpower you can add to your collection.

Summary

- There are two kinds of NLP models: *bag-of-words models* that process sets of words or N-grams without taking into account their order, and *sequence models* that process word order. A bag-of-words model is made of `Dense` layers, while a sequence model could be an RNN, a 1D convnet, or a Transformer.
- When it comes to text classification, the ratio between the number of samples in your training data and the mean number of words per sample can help you determine whether you should use a bag-of-words model or a sequence model.
- *Word embeddings* are vector spaces where semantic relationships between words are modeled as distance relationships between vectors that represent those words.
- *Sequence-to-sequence learning* is a generic, powerful learning framework that can be applied to solve many NLP problems, including machine translation. A sequence-to-sequence model is made of an encoder, which processes a source sequence, and a decoder, which tries to predict future tokens in target sequence by looking at past tokens, with the help of the encoder-processed source sequence.
- *Neural attention* is a way to create context-aware word representations. It's the basis for the Transformer architecture.
- The *Transformer* architecture, which consists of a `TransformerEncoder` and a `TransformerDecoder`, yields excellent results on sequence-to-sequence tasks. The first half, the `TransformerEncoder`, can also be used for text classification or any sort of single-input NLP task.