6.1 Working with text data

Text is one of the most widespread forms of sequence data. It can be understood as either a sequence of characters or a sequence of words, but it's most common to work at the level of words. The deep-learning sequence-processing models that we'll introduce in the following sections can use text to produce a basic form of natural language understanding, sufficient for applications including document classification, sentiment analysis, author identification, and even answering questions (in a constrained context). Of course, keep in mind throughout this chapter that none of these deep-learning models truly understand text in a human sense; rather, these models can map the statistical structure of written language, which is sufficient to solve many simple textual tasks. Deep learning for natural-language processing is pattern recognition applied to words, sentences, and paragraphs, in much the same way that computer vision is pattern recognition applied to pixels.

Like all other neural networks, deep-learning models don't take as input raw text: they only work with numeric tensors. *Vectorizing* text is the process of transforming text into numeric tensors. This can be done in multiple ways:

- Segment text into words, and transform each word into a vector.
- Segment text into characters, and transform each character into a vector.
- Extract N-grams of words or characters, and transform each N-gram into a vector. *N-grams* are overlapping groups of multiple consecutive words or characters.

Collectively, the different units into which you can break down text (words, characters, or N-grams) are called *tokens*, and breaking text into such tokens is called *tokenization*. All text-vectorization processes consist of applying some tokenization scheme and then associating numeric vectors with the generated tokens. These vectors, packed into sequence tensors, are fed into deep neural networks. There are multiple ways to associate a vector with a token. In this section, we'll present two major ones: *one-hot encoding* of tokens, and *token embedding* (typically used exclusively for words, and called *word embedding*). The remainder of this section explains these techniques and shows how to use them to go from raw text to a tensor that you can send to a Keras network.

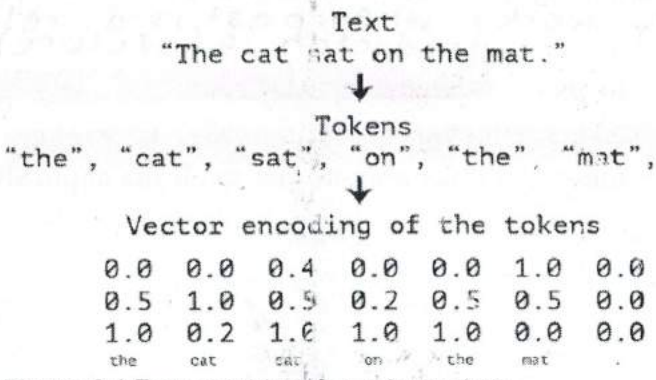

Figure 6.1 From text to tokens to vectors

NOTE **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", "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", "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*.

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 does away with this kind of rigid, brittle approach, replacing it with hierarchical feature learning. One-dimensional convnets and recurrent neural networks, introduced later in this chapter, 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. For this reason, we won't be covering N-grams any further in this book. But do keep in mind that they're a powerful, unavoidable feature-engineering tool when using lightweight, shallow text-processing models such as logistic regression and random forests.

6.1.1 One-hot encoding of words and characters

One-hot encoding is the most common, most basic way to turn a token into a vector. You saw it in action in the initial IMDB and Reuters examples in chapter 3 (done with words, in that case). It consists of associating a unique integer index with every word and then turning this integer index *i* into a binary vector of size *N* (the size of the vocabulary); the vector is all zeros except for the _i_th entry, which is 1.

Of course, one-hot encoding can be done at the character level, as well. To unambiguously drive home what one-hot encoding is and how to implement it, listings 6.1 and 6.2 show two toy examples: one for words, the other for characters.

Listing 6.1 Word-level one-hot encoding (toy example)

```
samples <- c("The cat sat on the mat.", "The dog ate my homework.")

token_index <- list()
for (sample in samples)
  for (word in strsplit(sample, " ")[[1]])
    if (!word %in% names(token_index))
      token_index[[word]] <- length(token_index) + 2

max_length <- 10

results <- array(0, dim = c(length(samples),
                            max_length,
                            max(as.integer(token_index))))

for (i in 1:length(samples)) {
  sample <- samples[[i]]
  words <- head(strsplit(sample, " ")[[1]], n = max_length)
  for (j in 1:length(words)) {
    index <- token_index[[words[[j]]]]
    results[[i, j, index]] <- 1
  }
}
```

1. Initial data: one entry per sample (in this example, a sample is a sentence, but it could be an entire document)
2. Builds an index of all tokens in the data
3. Tokenizes the samples via the strsplit function. In real life, you'd also strip punctuation and special characters from the samples.
4. Assigns a unique index to each unique word. Note that you don't attribute index 1 to anything.
5. Vectorizes the samples. You'll only consider the first max_length words in each sample.
6. This is where you store the results.

Listing 6.2 Character-level one-hot encoding (toy example)

```
samples <- c("The cat sat on the mat.", "The dog ate my homework.")

ascii_tokens <- c("", sapply(as.raw(c(32:126)), rawToChar))
token_index <- c(1:(length(ascii_tokens)))
names(token_index) <- ascii_tokens

max_length <- 50

results <- array(0, dim = c(length(samples), max_length, length(token_index)))

for (i in 1:length(samples)) {
  sample <- samples[[i]]
  characters <- strsplit(sample, "")[[1]]
  for (j in 1:length(characters)) {
    character <- characters[[j]]
    results[i, j, token_index[[character]]] <- 1
  }
}
```

Note that Keras has built-in utilities for doing one-hot encoding of text at the word level or character level, starting from raw text data. You should use these utilities, because they take care of a number of important features such as stripping special characters from strings and only taking into account the N most common words in your dataset (a common restriction, to avoid dealing with very large input vector spaces).

Listing 6.3 Using Keras for word-level one-hot encoding

```
library(keras)

samples <- c("The cat sat on the mat.", "The dog ate my homework.")

tokenizer <- text_tokenizer(num_words = 1000) %>%                    ❶
  fit_text_tokenizer(samples)                                        ❷

sequences <- texts_to_sequences(tokenizer, samples)                  ❸

one_hot_results <- texts_to_matrix(tokenizer, samples, mode = "binary")   ❹

word_index <- tokenizer$word_index                                   ❺
cat("Found", length(word_index), "unique tokens.\n")
```

❶ Creates a tokenizer, configured to only take into account the 1,000 most common words

❷ Builds the word index

❸ Turns strings into lists of integer indices

❹ You could also directly get the one-hot binary representations. Vectorization modes other than one-hot encoding are supported by this tokenizer.

❺ How you can recover the word index that was computed

A variant of one-hot encoding is the so-called *one-hot hashing trick*, which you can use when the number of unique tokens in your vocabulary is too large to handle explicitly. Instead of explicitly assigning an index to each word and keeping a reference of these indices in a dictionary, you can hash words into vectors of fixed size. This is typically done with a very lightweight hashing function. The main advantage of this method is that it does away with maintaining an explicit word index, which saves memory and allows online encoding of the data (you can generate token vectors right away, before you've seen all of the available data). The one drawback of this approach is that it's susceptible to *hash collisions*: two different words may end up with the same hash, and subsequently any machine-learning model looking at these hashes won't be able to tell the difference between these words. The likelihood of hash collisions decreases when the dimensionality of the hashing space is much larger than the total number of unique tokens being hashed.

Listing 6.4 Word-level one-hot encoding with hashing trick (toy example)

```
library(hashFunction)

samples <- c("The cat sat on the mat.", "The dog ate my homework.")

dimensionality <- 1000
max_length <- 10

results <- array(0, dim = c(length(samples), max_length, dimensionality))

for (i in 1:length(samples)) {
  sample <- samples[[i]]
  words <- head(strsplit(sample, " ")[[1]], n = max_length)
  for (j in 1:length(words)) {
    index <- abs(spooky.32(words[[j]])) %% dimensionality
    results[[i, j, index]] <- 1
  }
}
```

1. Stores the words as vectors of size 1,000. If you have close to 1,000 words (or more), you'll see many hash collisions, which will decrease the accuracy of this encoding method.
2. Use hashFunction::spooky.32() to hash the word into a random integer index between 0 and 1,000.

6.1.2 Using word embeddings

Another popular and powerful way to associate a vector with a word is the use of dense *word vectors*, also called *word embeddings*. Whereas the vectors obtained through one-hot encoding are binary, sparse (mostly made of zeros), and very high-dimensional (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 6.2. Unlike the word vectors obtained via one-hot encoding, word embeddings are learned from data. 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 token, in this case). So, word embeddings pack more information into far fewer dimensions.

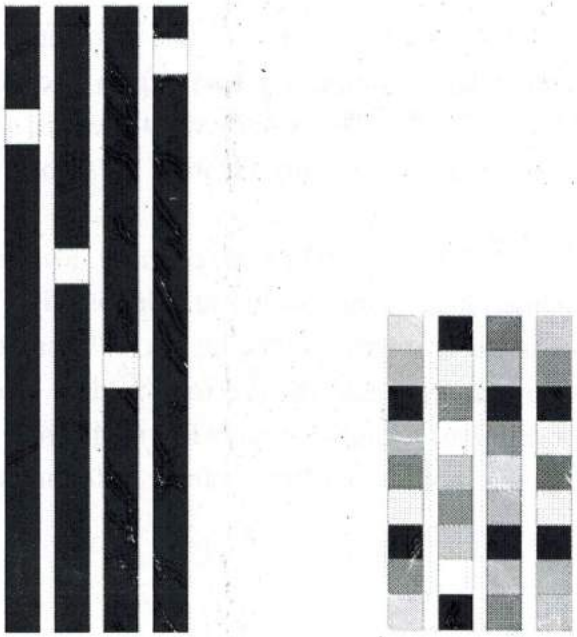

Figure 6.2 Whereas word representations obtained from one-hot encoding or hashing are sparse, high-dimensional, and hardcoded, word embeddings are dense, relatively low-dimensional, and learned from data.

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 look at both.

LEARNING WORD EMBEDDINGS WITH AN EMBEDDING LAYER

The simplest way to associate a dense vector with a word is to choose the vector at random. The problem with this approach is that the resulting embedding space has no structure: for instance, the words *accurate* and *exact* may end up with completely different embeddings, even though they're interchangeable in most sentences. It's difficult for a deep neural network to make sense of such a noisy, unstructured embedding space.

To get a bit more abstract, the geometric relationships between word vectors should reflect the semantic relationships between these words. Word embeddings are meant to map human language into a geometric space. For instance, in a reasonable embedding space, you would expect synonyms to be embedded into similar word vectors; and in general, you would expect the geometric distance (such as L2 distance) between any two

word vectors to relate to the semantic distance between the associated words (words meaning different things are embedded at points far away from each other, whereas related words are closer). In addition to distance, you may want specific *directions* in the embedding space to be meaningful. To make this clearer, let's look at a concrete example.

In figure 6.3, 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.

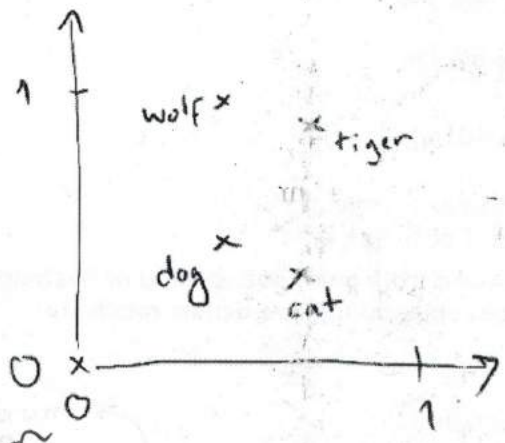

Figure 6.3 A toy example of a word-embedding space

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.

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, 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 using `layer_embedding`.

Listing 6.5 Instantiating an embedding layer

```
embedding_layer <- layer_embedding(input_dim = 1000, output_dim = 64)
```

The embedding layer takes at least two arguments: the number of possible tokens (here, 1,000) and the dimensionality of the embeddings (here, 64).

A `layer_embedding` is best understood as a dictionary that maps integer indices (which stand for specific words) to dense vectors. It takes integers as input, it looks up these integers in an internal dictionary, and it returns the associated vectors. It's effectively a dictionary lookup (see figure 6.4).

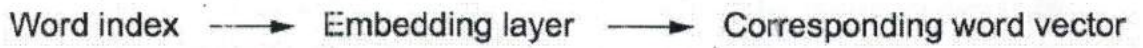

Figure 6.4 An embedding layer

An embedding layer takes as input a 2D tensor of integers, of shape `(samples, sequence_length)`, where each entry is a sequence of integers. It can embed sequences of variable lengths: for instance, you could feed into the embedding layer in the previous example batches with shapes `(32, 10)` (batch of 32 sequences of length 10) or `(64, 15)` (batch of 64 sequences of length 15). All sequences in a batch must have the same length, though (because you need to pack them into a single tensor), so sequences that are shorter than others should be padded with zeros, and sequences that are longer should be truncated.

This layer returns a 3D floating-point tensor, of shape `(samples, sequence_length, embedding_dimensionality)`. Such a 3D tensor can then be processed by an RNN layer or a 1D convolution layer (both will be introduced in the following sections).

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 were training your model.

Let's apply this idea to the IMDB movie-review sentiment-prediction task that you're already familiar with. First, you'll quickly prepare the data. You'll restrict the movie reviews to the top 10,000 most common words (as you did the first time you worked with this dataset) and cut off the reviews after only 20 words. The network will learn 8-dimensional embeddings for each of the 10,000 words, turn the input integer sequences (2D integer tensor) into embedded sequences (3D float tensor), flatten the tensor to 2D, and train a single dense layer on top for classification.

Listing 6.6 Loading the IMDB data for use with an embedding layer

```
max_features <- 10000                                    ❶
maxlen <- 20                                             ❷

imdb <- dataset_imdb(num_words = max_features)
c(c(x_train, y_train), c(x_test, y_test)) %<-% imdb      ❸

x_train <- pad_sequences(x_train, maxlen = maxlen)       ❹
x_test <- pad_sequences(x_test, maxlen = maxlen)
```

❶ Number of words to consider as features

❷ Cuts off the text after this number of words (among the max_features most common words)

❸ Loads the data as lists of integers

❹ Turns the lists of integers into a 2D integer tensor of shape (samples, maxlen)

Listing 6.7 Using an embedding layer and classifier on the IMDB data

```
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = 10000, output_dim = 8,     ❶
                  input_length = maxlen) %>%
  layer_flatten() %>%                                    ❷
  layer_dense(units = 1, activation = "sigmoid")         ❸

model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("acc")
)

summary(model)

history <- model %>% fit(
  x_train, y_train,
  epochs = 10,
  batch_size = 32,
  validation_split = 0.2
)
```

❶ Specifies the maximum input length to the embedding layer so you can later flatten the embedded inputs. After the embedding layer, the activations have shape (samples, maxlen, 8).

❷ Flattens the 3D tensor of embeddings into a 2D tensor of shape (samples, maxlen * 8)

❸ Adds the classifier on top

You get to a validation accuracy of ~76%, which is pretty good considering that you're only looking at the first 20 words in every review. But note that merely flattening the embedded sequences and training a single dense layer on top leads to a model that treats each word in the input sequence separately, without considering inter-word relationships and sentence structure (for example, this model would likely treat both "this movie is a bomb" and "this movie is the bomb" as being negative reviews). It's much

Read
Reread & ask questions
Reread & ask questions more clearly, try to find answers

6 Deep learning for text and sequences

This chapter covers

- Preprocessing text data into useful representations
- Working with recurrent neural networks
- Using 1D convnets for sequence processing

This chapter explores deep-learning models that can process text (understood as sequences of word or sequences of characters), timeseries, and sequence data in general. The two fundamental deep-learning algorithms for sequence processing are *recurrent neural networks* and *1D convnets*, the one-dimensional version of the 2D convnets that we covered in the previous chapters. We'll discuss both of these approaches in this chapter.

Applications of these algorithms include the following:

- Document classification and timeseries classification, such as identifying the topic of an article or the author of a book, fancier topic modelling & classify
- Timeseries comparisons, such as estimating how closely related two documents or two stock tickers are, similarity / relation calculation
- Sequence-to-sequence learning, such as decoding an English sentence into French, translation
- Sentiment analysis, such as classifying the sentiment of tweets or movie reviews as positive or negative, sentiment analysis
- Timeseries forecasting, such as predicting the future weather at a certain location, given recent weather data (only works when past is a reliable predictor of future)

This chapter's examples will focus on two narrow tasks: sentiment analysis on the IMDB dataset, a task we approached earlier in the book, and weather forecasting. But the techniques we'll demonstrate for these two tasks are relevant to all the applications we just listed, and many more.

better to add recurrent layers or 1D convolutional layers on top of the embedded sequences to learn features that take into account each sequence as a whole. That's what we'll focus on in the next few sections.

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. What do you do then?

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—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 the features that you need to be 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,[10] 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 (code.google.com/archive/p/word2vec), developed by Tomas Mikolov at Google in 2013. Word2vec dimensions capture specific semantic properties, such as gender.

Footnote 10 Yoshua Bengio et al., *Neural Probabilistic Language Models* (Springer, 2003).

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, 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 will of course be valid for Word2vec embeddings or any other word-embedding database. You'll also use this example to refresh the text-tokenization techniques we introduced a few paragraphs ago: you'll start from raw text and work your way up.

6.1.3 Putting it all together: from raw text to word embeddings

You'll be using a model similar to the one we just went over: embedding sentences in sequences of vectors, flattening them, and training a dense layer on top. But you'll do so using pretrained word embeddings; and instead of using the pretokenized IMDB data packaged in Keras, you'll start from scratch by downloading the original text data.

DOWNLOADING THE IMDB DATA AS RAW TEXT

First, head to ai.stanford.edu/~amaas/data/sentiment and download the raw IMDB dataset (if the URL isn't working anymore, Google "IMDB dataset"). Uncompress it.

Now, let's collect the individual training reviews into a list of strings, one string per review. You'll also collect the review labels (positive / negative) into a `labels` list.

Listing 6.8 Processing the labels of the raw IMDB data

```
imdb_dir <- "~/Downloads/aclImdb"
train_dir <- file.path(imdb_dir, "train")

labels <- c()
texts <- c()

for (label_type in c("neg", "pos")) {
  label <- switch(label_type, neg = 0, pos = 1)
  dir_name <- file.path(train_dir, label_type)
  for (fname in list.files(dir_name, pattern = glob2rx("*.txt"),
                           full.names = TRUE)) {
    texts <- c(texts, readChar(fname, file.info(fname)$size))
    labels <- c(labels, label)
  }
}
```

TOKENIZING THE DATA

Let's vectorize the text and prepare a training and validation split, using the concepts we introduced earlier in this section. Because pretrained word embeddings are meant to be particularly useful on problems where little training data is available (otherwise, task-specific embeddings are likely to outperform them), we'll add the following twist: restricting the training data to the first 200 samples. So you'll be learning to classify movie reviews after looking at just 200 examples.

Listing 6.9 Tokenizing the text of the raw IMDB data

```
library(keras)

maxlen <- 100                 # (1)
training_samples <- 200       # (2)
validation_samples <- 10000   # (3)
max_words <- 10000            # (4)

tokenizer <- text_tokenizer(num_words = max_words) %>%
  fit_text_tokenizer(texts)

sequences <- texts_to_sequences(tokenizer, texts)

word_index = tokenizer$word_index
```

```
cat("Found", length(word_index), "unique tokens.\n")

data <- pad_sequences(sequences, maxlen = maxlen)

labels <- as.array(labels)
cat("Shape of data tensor:", dim(data), "\n")
cat('Shape of label tensor:', dim(labels), "\n")

indices <- sample(1:nrow(data))                                    ❺
training_indices <- indices[1:training_samples]
validation_indices <- indices[(training_samples + 1):
                              (training_samples + validation_samples)]

x_train <- data[training_indices,]
y_train <- labels[training_indices]

x_val <- data[validation_indices,]
y_val <- labels[validation_indices]
```

❶ Cuts off reviews after 100 words

❷ Trains on 200 samples

❸ Validates on 10,000 samples

❹ Considers only the top 10,000 words in the dataset

❺ Splits the data into a training set and a validation set. But first shuffles the data, because you're starting with data in which samples are ordered (all negative first, then all positive).

DOWNLOADING THE GLOVE WORD EMBEDDINGS

Go to nlp.stanford.edu/projects/glove, and download the precomputed embeddings from 2014 English Wikipedia. It's a 822 MB zip file called glove.6B.zip, containing 100-dimensional embedding vectors for 400,000 words (or nonword tokens). Unzip it.

PREPROCESSING THE EMBEDDINGS

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

Listing 6.10 Parsing the GloVe word-embeddings file

```
glove_dir = '~/Downloads/glove.6B'
lines <- readLines(file.path(glove_dir, "glove.6B.100d.txt"))

embeddings_index <- new.env(hash = TRUE, parent = emptyenv())
for (i in 1:length(lines)) {
  line <- lines[[i]]
  values <- strsplit(line, " ")[[1]]
  word <- values[[1]]
  embeddings_index[[word]] <- as.double(values[-1])
}

cat("Found", length(embeddings_index), "word vectors.\n")
```

Next, you'll 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). Note that index 1 isn't supposed to stand for any word or token—it's a placeholder.

Listing 6.11 Preparing the GloVe word-embeddings matrix

```
embedding_dim <- 100

embedding_matrix <- array(0, c(max_words, embedding_dim))

for (word in names(word_index)) {
  index <- word_index[[word]]
  if (index < max_words) {
    embedding_vector <- embeddings_index[[word]]
    if (!is.null(embedding_vector))
      embedding_matrix[index+1,] <- embedding_vector    ❶
  }
}
```

❶ Words not found in the embedding index will be all zeros.

DEFINING A MODEL

You'll use the same model architecture as before.

Listing 6.12 Model definition

```
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = max_words, output_dim = embedding_dim,
                  input_length = maxlen) %>%
  layer_flatten() %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

summary(model)
```

LOADING THE GLOVE EMBEDDINGS IN THE MODEL

The embedding layer has a single weight matrix: a 2D float matrix where each entry *i* is the word vector meant to be associated with index *i*. Simple enough. Load the GloVe matrix you prepared into the embedding layer, the first layer in the model.

Listing 6.13 Loading pretrained word embeddings into the embedding layer

```
get_layer(model, index = 1) %>%
  set_weights(list(embedding_matrix)) %>%
  freeze_weights()
```

Additionally, you'll freeze the weights of the embedding layer, following the same rationale you're already familiar with in the context of pretrained convnet features: when parts of a model are pretrained (like your embedding layer) and parts are randomly initialized (like your classifier), the pretrained parts shouldn't be updated during training, to avoid forgetting what they already know. The large gradient updates triggered by the randomly initialized layers would be disruptive to the already-learned features.

TRAINING AND EVALUATING THE MODEL

Compile and train the model.

Listing 6.14 Training and evaluation

```
model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("acc")
)

history <- model %>% fit(
  x_train, y_train,
  epochs = 20,
  batch_size = 32,
  validation_data = list(x_val, y_val)
)

save_model_weights_hdf5(model, "pre_trained_glove_model.h5")
```

Now, plot the model's performance over time (see figure 6.5).

Listing 6.15 Plotting the results

```
plot(history)
```

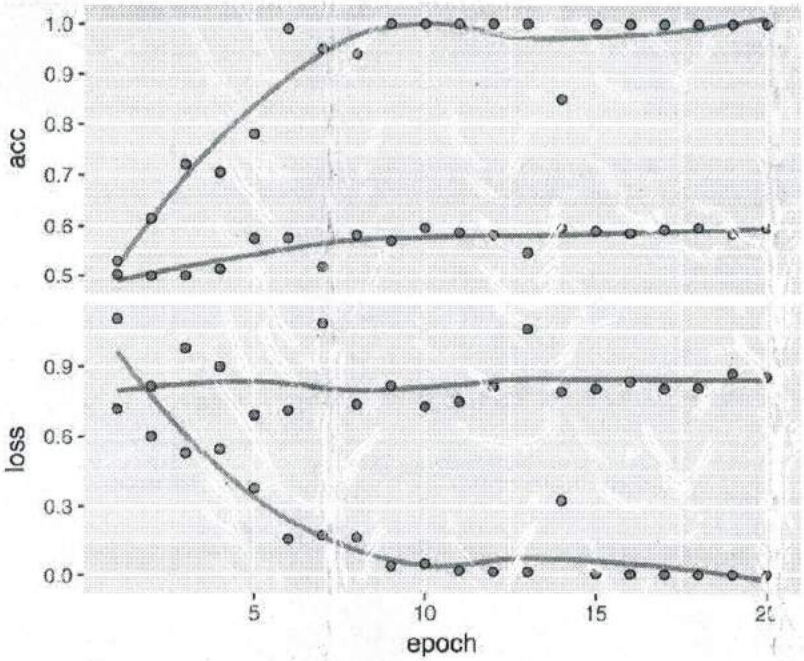

Figure 6.5 Training and validation metrics when using pretrained word embeddings

Remember: learning curves (evaluations) show changes in learning performance over time, in terms of experience.

We use it to diagnose a model as: · underfit, · overfit, OR · well fit

A plot of learning curves show overfitting if: · the plot of training loss continues to decrease w/ experience AND · the plot of validation loss ↓ to a point then

The model quickly starts overfitting, which is unsurprising given the small number of training samples. Validation accuracy has high variance for the same reason, but it seems to reach the high 50s.

Note that your mileage may vary: because you have so few training samples, performance is heavily dependent on exactly which 200 samples you choose—and you're

choosing them at random. If this works poorly for you, try choosing a different random set of 200 samples, for the sake of the exercise (in real life, you don't get to choose your training data).

You can also train the same model without loading the pretrained word embeddings and without freezing the embedding layer. In that case, you'll be learning a task-specific embedding of the input tokens, which is generally more powerful than pretrained word embeddings when lots of data is available. But in this case, you have only 200 training samples. Let's try it (see figure 6.6).

Listing 6.16 Training the same model without pretrained word embeddings

```
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = max_words, output_dim = embedding_dim,
                  input_length = maxlen) %>%
  layer_flatten() %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("acc")
)

history <- model %>% fit(
  x_train, y_train,
  epochs = 20,
  batch_size = 32,
  validation_data = list(x_val, y_val)
)
```

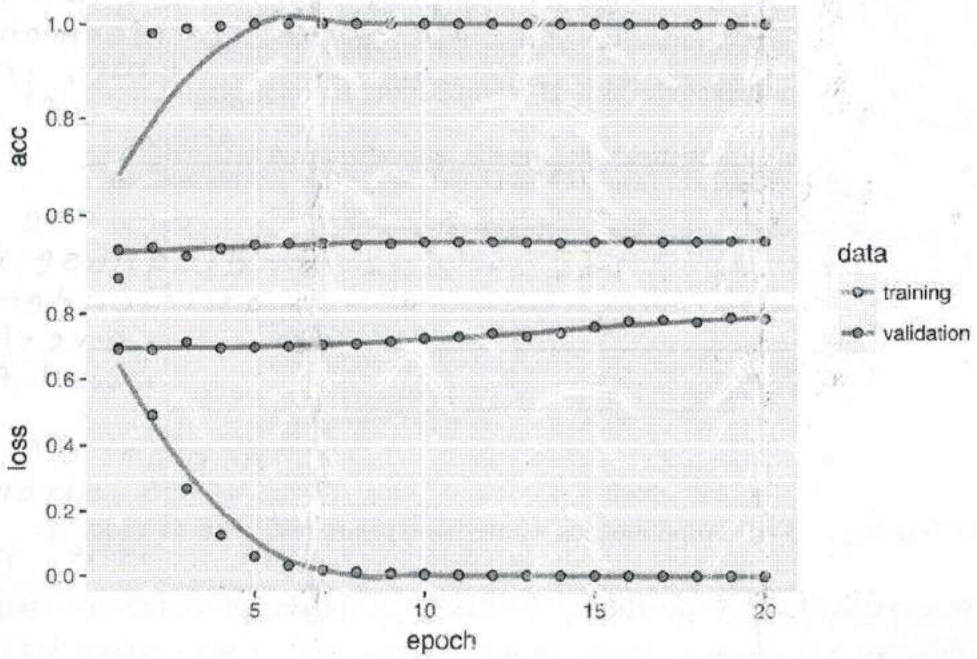

Figure 6.6 Training and validation metrics without using pretrained word embeddings

Validation accuracy stalls in the mid 50s. So in this case, pretrained word embeddings outperform jointly learned embeddings. If you increase the number of training samples,

this will quickly stop being the case—try it as an exercise.

Finally, let's evaluate the model on the test data. First, you need to tokenize the test data.

Listing 6.17 Tokenizing the data of the test set

```
test_dir <- file.path(imdb_dir, "test")

labels <- c()
texts <- c()
                    negative + positive
for (label_type in c("neg", "pos")) {
  label <- switch(label_type, neg = 0, pos = 1)
  dir_name <- file.path(test_dir, label_type)
  for (fname in list.files(dir_name, pattern = glob2rx("*.txt"),
                           full.names = TRUE)) {
    texts <- c(texts, readChar(fname, file.info(fname)$size))
    labels <- c(labels, label)
  }
}

sequences <- texts_to_sequences(tokenizer, texts)
x_test <- pad_sequences(sequences, maxlen = maxlen)
y_test <- as.array(labels)
```

need to think critically about how they got this series of ops. so I can design my own when needed.

Next, load and evaluate the first model.

Listing 6.18 Evaluating the model on the test set

```
model %>%
  load_model_weights_hdf5("pre_trained_glove_model.h5") %>%
  evaluate(x_test, y_test)
```

You get an appalling test accuracy of 58%. Working with just a handful of training samples is difficult!

6.1.4 Wrapping up

Now you're able to do the following:

- Turn raw text into something a neural network can process
- Use an embedding layer in a Keras model to learn task-specific token embeddings
- Use pretrained word embeddings to get an extra boost on small natural-language-processing problems

6.2 Understanding recurrent neural networks

A major characteristic of all neural networks you've seen so far, such as densely connected networks and convnets, is that they have no memory. Each input shown to them is processed independently, with no state kept in between inputs. With such networks, in order to process a sequence or a temporal series of data points, you have to show the entire sequence to the network at once: that is, turn it into a single data point. For instance, this is what you did in the IMDB example: an entire movie review was transformed into a single large vector and processed in one go. Such networks are called *feedforward networks*.

In contrast, as you're reading the present sentence, you're processing it word by word—or rather, eye saccade by eye saccade—while keeping memories of what came before; this gives you a fluid representation of the meaning we're conveying with this sentence. Biological intelligence processes information incrementally while maintaining an internal model of what it's processing, built from past information and constantly updated as new information comes in.

A *recurrent neural network* (RNN) adopts the same principle, albeit in an extremely simplified version: it processes sequences by iterating through the sequence elements and maintaining a *state* containing information relative to what it has seen so far. In effect, an RNN is a type of neural network that has an internal loop (see figure 6.9). The state of the RNN is reset between processing two different, independent sequences (such as two different IMDB reviews), so you still consider one sequence a single data point: a single input to the network. What changes is that this data point is no longer processed in a single step; rather, the network internally loops over sequence elements.

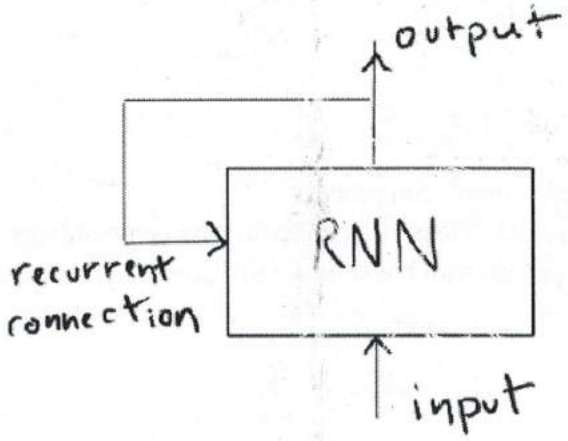

Figure 6.7 A recurrent network: a network with a loop

To make these notions of *loop* and *state* clear, let's implement the forward pass of a toy RNN in R. This RNN takes as input a sequence of vectors, which you'll encode as a 2D tensor of size `(timesteps, input_features)`. It loops over timesteps, and at each timestep, it considers its current state at `t` and the input at `t` (of shape `(input_features)`, and combines them to obtain the output at `t`. You'll then set the

state for the next step to be this previous output. For the first timestep, the previous output isn't defined; hence there is no current state. So, you'll initialize the state as an all-zero vector called the *initial state* of the network.

In pseudocode, this is the RNN:

Listing 6.19 Pseudocode RNN

```
state_t = 0                                  (1)
for (input_t in input_sequence) {            (2)
  output_t <- f(input_t, state_t)
  state_t <- output_t                        (3)
}
```

(1) The state at t

(2) Iterates over sequence elements

(3) The previous output becomes the new state.

You can even flesh out the function `f`: the transformation of the input and state into an output will be parametrized by two matrices, `W` and `U`, and a bias vector. It's similar to the transformation operated by a densely connected layer in a feedforward network.

Listing 6.20 More detailed pseudocode for the RNN

```
state_t <- 0
for (input_t in input_sequence) {
  output_t <- activation(dot(W, input_t) + dot(U, state_t) + b)
  state_t <- output_t
}
```

To make these notions absolutely unambiguous, let's write a naive R implementation of the forward pass of the simple RNN.

Listing 6.21 R implementation of a simple RNN

```
timesteps <- 100                             (1)
input_features <- 32                         (2)
output_features <- 64                        (3)

random_array <- function(dim) {
  array(runif(prod(dim)), dim = dim)
}

inputs <- random_array(dim = c(timesteps, input_features))   (4)
state_t <- rep_len(0, length = c(output_features))           (5)

W <- random_array(dim = c(output_features, input_features))  (6)
U <- random_array(dim = c(output_features, output_features)) (7)
b <- random_array(dim = c(output_features, 1))               (8)
```

```
output_sequence <- array(0, dim = c(timesteps, output_features))
for (i in 1:nrow(inputs)) {
  input_t <- inputs[i,]
  output_t <- tanh(as.numeric((W %*% input_t) + (U %*% state_t) + b))
  output_sequence[i,] <- as.numeric(output_t)
  state_t <- output_t
}
```

1. Number of timesteps in the input sequence
2. Dimensionality of the input feature space
3. Dimensionality of the output feature space
4. Input data: random noise for the sake of the example
5. Initial state: an all-zero vector
6. Creates random weight matrices
7. input_t is a vector of shape (input_features).
8. Combines the input with the current state (the previous output) to obtain the current output
9. Updates the result matrix
10. Updates the state of the network for the next timestep

Easy enough: in summary, an RNN is a `for` loop that reuses quantities computed during the previous iteration of the loop. Nothing more. Of course, there are many different RNNs fitting this definition that you could build—this example is one of the simplest RNN formulations. RNNs are characterized by their step function, such as the following function in this case (see figure 6.8):

```
output_t <- tanh(as.numeric((W %*% input_t) + (U %*% state_t) + b))
```

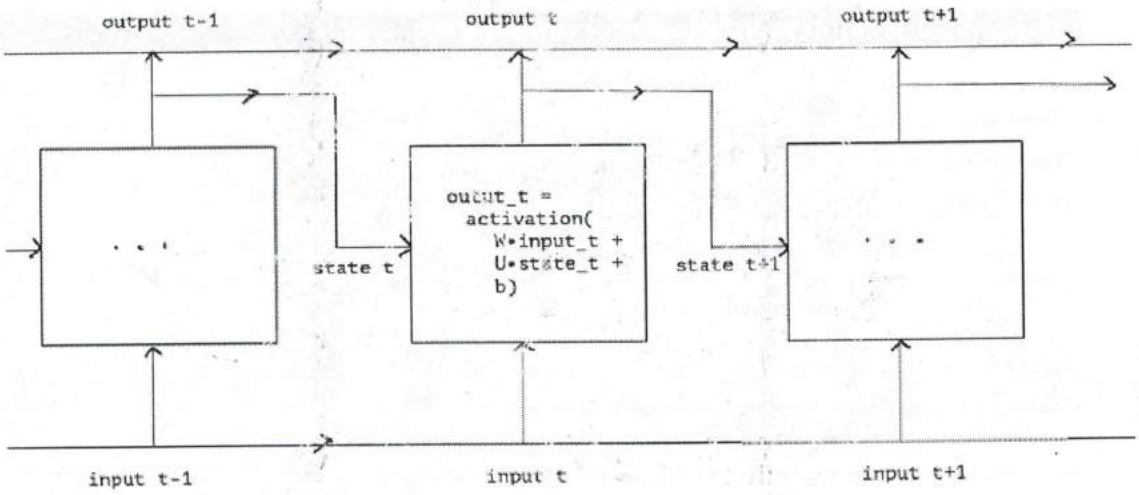

Figure 6.8 A simple RNN, unrolled over time

NOTE In this example, the final output is a 2D tensor of shape `(timesteps, output_features)`, where each timestep is the output of the loop at time `t`. Each timestep `t` in the output tensor contains information about timesteps `1` to `t` in the input sequence—about the entire past. For this reason, in many cases, you don't need this full sequence of outputs; you just need the last output (`output_t` at the end of the loop), because it already contains information about the entire sequence.

6.2.1 A recurrent layer in Keras

The process you just naively implemented in R corresponds to an actual Keras layer—`layer_simple_rnn`.

```
layer_simple_rnn(units = 32)
```

There is one minor difference: `layer_simple_rnn` processes batches of sequences, like all other Keras layers, not a single sequence as in the R example. This means it takes inputs of shape `(batch_size, timesteps, input_features)`, rather than `(timesteps, input_features)`.

Like all recurrent layers in Keras, `layer_simple_rnn` can be run in two different modes: it can return either the full sequences of successive outputs for each timestep (a 3D tensor of shape `(batch_size, timesteps, output_features)`) or only the last output for each input sequence (a 2D tensor of shape `(batch_size, output_features)`). These two modes are controlled by the `return_sequences` constructor argument. Let's look at an example that uses `layer_simple_rnn` and returns the last state:

```
library(keras)
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = 10000, output_dim = 32) %>%
  layer_simple_rnn(units = 32)

> summary(model)

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_22 (Embedding)     (None, None, 32)          320000
_________________________________________________________________
simplernn_10 (SimpleRNN)     (None, 32)                2080
=================================================================
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0
```

The following example returns the full state sequence:

```
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = 10000, output_dim = 32) %>%
  layer_simple_rnn(units = 32, return_sequences = TRUE)
```

```
> summary(model)
_____________________________________________________________________
Layer (type)                 Output Shape              Param #
=====================================================================
embedding_23 (Embedding)     (None, None, 32)          320000
_____________________________________________________________________
simplernn_11 (SimpleRNN)     (None, None, 32)          2080
=====================================================================
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0
```

It's sometimes useful to stack several recurrent layers one after the other in order to increase the representational power of a network. In such a setup, you have to get all of the intermediate layers to return full sequences:

```
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = 10000, output_dim = 32) %>%
  layer_simple_rnn(units = 32, return_sequences = TRUE) %>%
  layer_simple_rnn(units = 32, return_sequences = TRUE) %>%
  layer_simple_rnn(units = 32, return_sequences = TRUE) %>%
  layer_simple_rnn(units = 32)                                    ❶

> summary(model)
_____________________________________________________________________
Layer (type)                 Output Shape              Param #
=====================================================================
embedding_24 (Embedding)     (None, None, 32)          320000
_____________________________________________________________________
simplernn_12 (SimpleRNN)     (None, None, 32)          2080
_____________________________________________________________________
simplernn_13 (SimpleRNN)     (None, None, 32)          2080
_____________________________________________________________________
simplernn_14 (SimpleRNN)     (None, None, 32)          2080
_____________________________________________________________________
simplernn_15 (SimpleRNN)     (None, 32)                2080
=====================================================================
Total params: 328,320
Trainable params: 328,320
Non-trainable params: 0
```

❶ Last layer only returns the last outputs

Now, let's use such a model on the IMDB movie-review-classification problem. First, preprocess the data.

Listing 6.22 Preparing the IMDB data

```
library(keras)

max_features <- 10000                                             ❶
maxlen <- 500                                                     ❷
batch_size <- 32

cat("Loading data...\n")
imdb <- dataset_imdb(num_words = max_features)
c(c(input_train, y_train), c(input_test, y_test)) %<-% imdb
cat(length(input_train), "train sequences\n")
cat(length(input_test), "test sequences")
```

```
cat("Pad sequences (samples x time)\n")
input_train <- pad_sequences(input_train, maxlen = maxlen)
input_test <- pad_sequences(input_test, maxlen = maxlen)
cat("input_train shape:", dim(input_train), "\n")
cat("input_test shape:", dim(input_test), "\n")
```

1. Number of words to consider as features
2. Cuts off texts after this many words (among the max_features most common words)

Let's train a simple recurrent network using a `layer_embedding` and `layer_simple_rnn`.

Listing 6.23 Training the model with embedding and simple RNN layers

```
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = max_features, output_dim = 32) %>%
  layer_simple_rnn(units = 32) %>%
  layer_dense(units = 1, activation = "sigmoid")

model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("acc")
)

history <- model %>% fit(
  input_train, y_train,
  epochs = 10,
  batch_size = 128,
  validation_split = 0.2
)
```

Now, let's display the training and validation loss and accuracy (see figure 6.9).

Listing 6.24 Plotting results

```
plot(history)
```

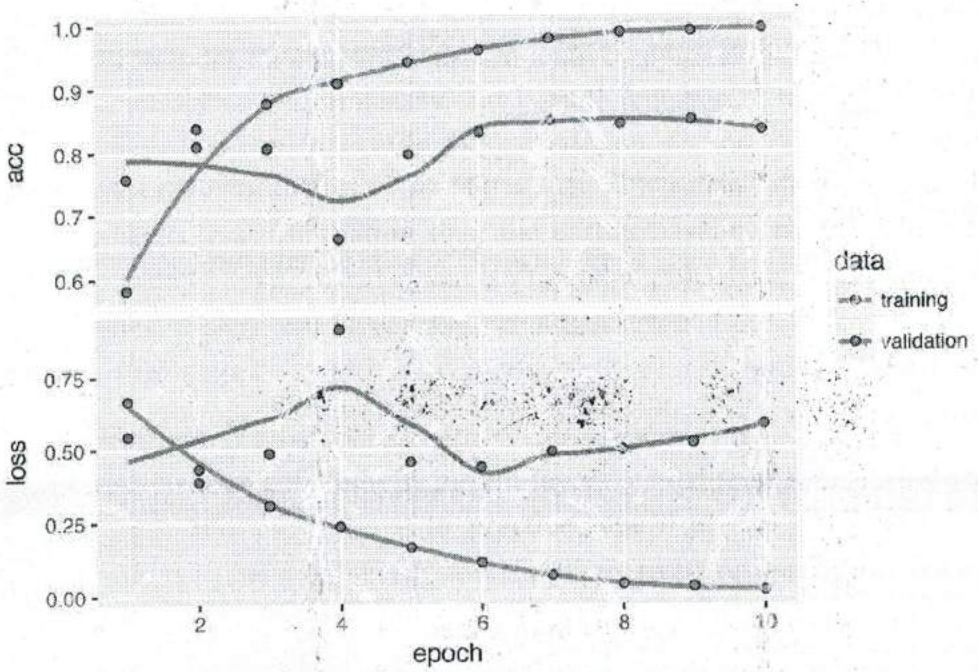

Figure 6.9 Training and validation metrics on IMDB with layer_simple_rnn

As a reminder, in chapter 3, the first naive approach to this dataset got you to a test accuracy of 88%. Unfortunately, this small recurrent network doesn't perform well compared to this baseline (only 84% validation accuracy). Part of the problem is that your inputs only consider the first 500 words, rather than full sequences—hence the RNN has access to less information than the earlier baseline model. The remainder of the problem is that `layer_simple_rnn` isn't good at processing long sequences, such as text. Other types of recurrent layers perform much better. Let's look at some more advanced layers.

6.2.2 Understanding the LSTM and GRU layers

Simple RNNs aren't the only recurrent layers available in Keras. There are two others: `layer_lstm` and `layer_gru`. In practice, you'll always use one of these, because `layer_simple_rnn` is generally too simplistic to be of real use. One major issue with `layer_simple_rnn` is that although it should theoretically be able to retain at time `t` information about inputs seen many timesteps before, in practice, such long-term dependencies are impossible to learn. This is due to the *vanishing gradient problem*, an effect that is similar to what is observed with non-recurrent networks (feedforward networks) that are many layers deep: as you keep adding layers to a network, the network eventually becomes untrainable. The theoretical reasons for this effect were studied by Hochreiter, Schmidhuber, and Bengio in the early 1990s.[11] The `LSTM` and `GRU` layers are designed to solve this problem.

Footnote 11 See, for example, Yoshua Bengio, Patrice Simar, and Paolo Frasconi, "Learning Long-Term Dependencies with Gradient Descent Is Difficult," *IEEE Transactions on Neural Networks* 5, no. 2 (1994).

Let's consider the `LSTM` layer. The underlying Long Short-Term Memory (LSTM) algorithm was developed by Hochreiter and Schmidhuber in 1997;[12] it was the

culmination of their research on the vanishing gradient problem.

Footnote 12 Sepp Hochreiter and Jürgen Schmidhuber, "Long Short-Term Memory," *Neural Computation* 9, no. 8 (1997).

This layer is a variant of the `layer_simple_rnn` you already know about; it adds a way to carry information across many timesteps. Imagine a conveyor belt running parallel to the sequence you're processing. Information from the sequence can jump onto the conveyor belt at any point, be transported to a later timestep, and jump off, intact, when you need it. This is essentially what LSTM does: it saves information for later, thus preventing older signals from gradually vanishing during processing.

To understand this in detail, let's start from the simple RNN cell (see figure 6.10). Because you'll have a lot of weight matrices, you'll index the `W` and `U` matrices in the cell with the letter `o` (`Wo` and `Uo`) for *output*.

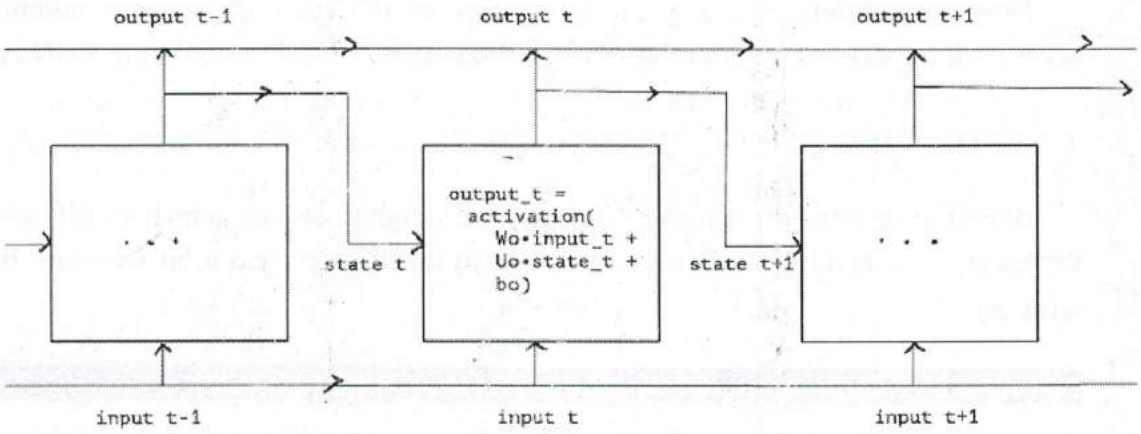

Figure 6.10 The starting point of an LSTM layer: a simple RNN

Let's add to this picture an additional data flow that carries information across timesteps. You'll call its values at different timesteps `Ct`, where *C* stands for *carry*. This information will have the following impact on the cell: it will be combined with the input connection and the recurrent connection (via a dense transformation: a dot product with a weight matrix followed by a bias add and the application of an activation function), and it will affect the state being sent to the next timestep (via an activation function an a multiplication operation). Conceptually, the carry dataflow is a way to modulate the next output and the next state (see figure 6.11). Simple so far.

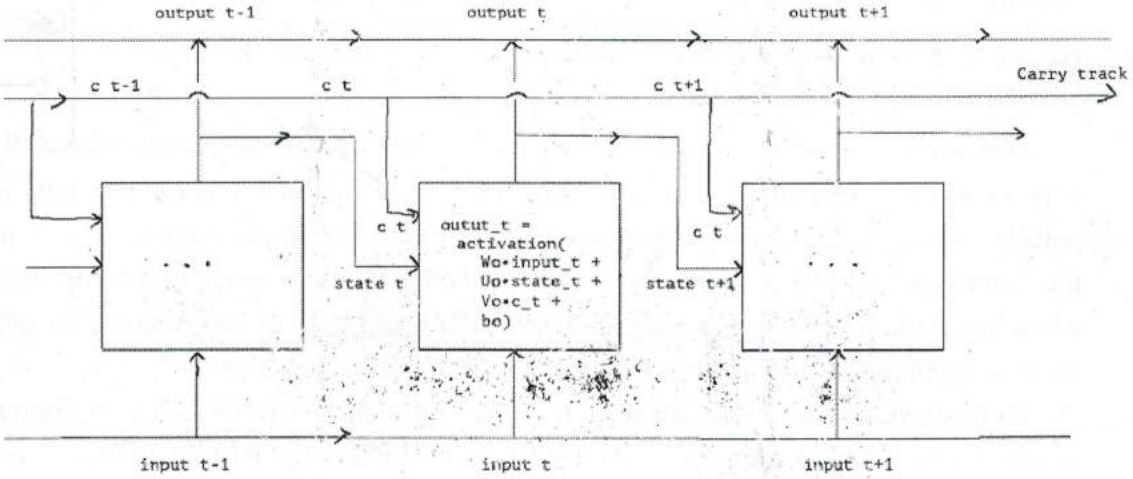

Figure 6.11 Going from a simple RNN to an LSTM: adding a carry track

Now the subtlety: the way the next value of the carry dataflow is computed. It involves three distinct transformations. All three have the form of a simple RNN cell:

```
y = activation(dot(state_t, U) + dot(input_t, W) + b)
```

But all three transformations have their own weight matrices, which you'll index with the letters `i`, `f`, and `k`. Here's what you have so far (it may seem a bit arbitrary, but bear with us).

Listing 6.25 Pseudocode details of the LSTM architecture (1/2)

```
output_t = activation(dot(state_t, Uo) + dot(input_t, Wo) + dot(C_t, Vo) + bo)

i_t = activation(dot(state_t, Ui) + dot(input_t, Wi) + bi)
f_t = activation(dot(state_t, Uf) + dot(input_t, Wf) + bf)
k_t = activation(dot(state_t, Uk) + dot(input_t, Wk) + bk)
```

You obtain the new carry state (the next `c_t`) by combining `i_t`, `f_t`, and `k_t`.

Listing 6.26 Pseudocode details of the LSTM architecture (2/2)

```
c_t+1 = i_t * k_t + c_t * f_t
```

Add this as shown in figure 6.12. And that's it. Not so complicated—merely a tad complex.

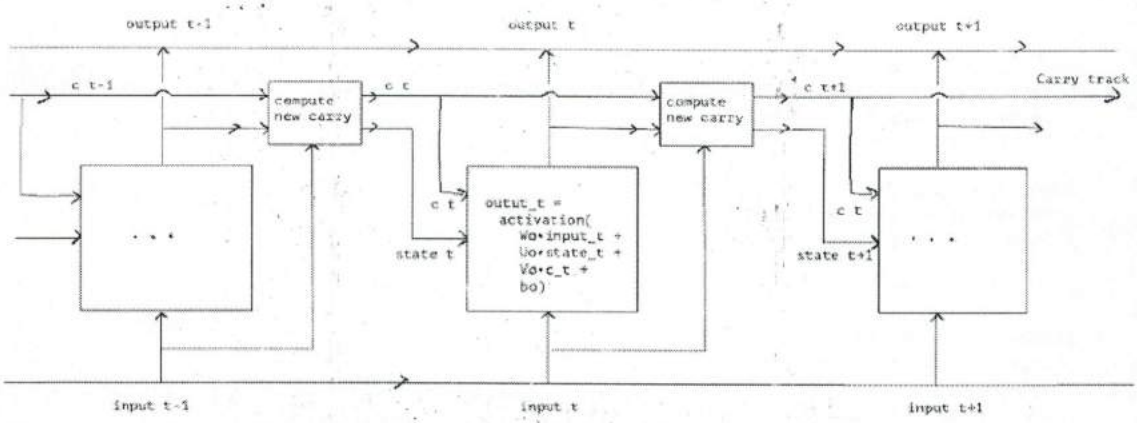

Figure 6.12 Anatomy of an LSTM

If you want to get philosophical, you can interpret what each of these operations is meant to do. For instance, you can say that multiplying `c_t` and `f_t` is a way to deliberately forget irrelevant information in the carry dataflow. Meanwhile, `i_t` and `k_t` provide information about the present, updating the carry track with new information. But at the end of the day, these interpretations don't mean much, because what these operations *actually* do is determined by the contents of the weights parametrizing them; and the weights are learned in an end-to-end fashion, starting over with each training round, making it impossible to credit this or that operation with a specific purpose. The specification of a RNN cell (as we just described) determines your hypothesis space—the space in which you'll search for a good model configuration during training—but it doesn't determine what the cell does; that is up to the cell weights. The same cell with different weights can be doing very different things. So the combination of operations making up a RNN cell is better interpreted as set of *constraints* on your search, not as a *design* in an engineering sense.

To a researcher, it seems that the choice of such constraints—the question of how to implement RNN cells—is better left to optimization algorithms (like genetic algorithms or reinforcement learning processes) than to human engineers. And in the future, that's how we'll build networks. In summary: you don't need to understand anything about the specific architecture of an LSTM cell; as a human, it shouldn't be your job to understand it. Just keep in mind what the LSTM cell is meant to do: allow past information to be reinjected at a later time, thus fighting the vanishing-gradient problem.

6.2.3 A concrete LSTM example in Keras

Now let's switch to more practical concerns: you'll set up a model using `layer_lstm` and train it on the IMDB data (see figure 6.13). The network is similar to the one with `layer_simple_rnn` that we just presented. You only specify the output dimensionality of the `layer_lstm`; leave every other argument (there are many) at the Keras defaults. Keras has good defaults, and things will almost always "just work" without you having to spend time tuning parameters by hand.

Listing 6.27 Using the LSTM layer in Keras

```
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = max_features, output_dim = 32) %>%
  layer_lstm(units = 32) %>%
  layer_dense(units = 1, activation = "sigmoid")

model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("acc")
)

history <- model %>% fit(
  input_train, y_train,
  epochs = 10,
  batch_size = 128,
  validation_split = 0.2
)
```

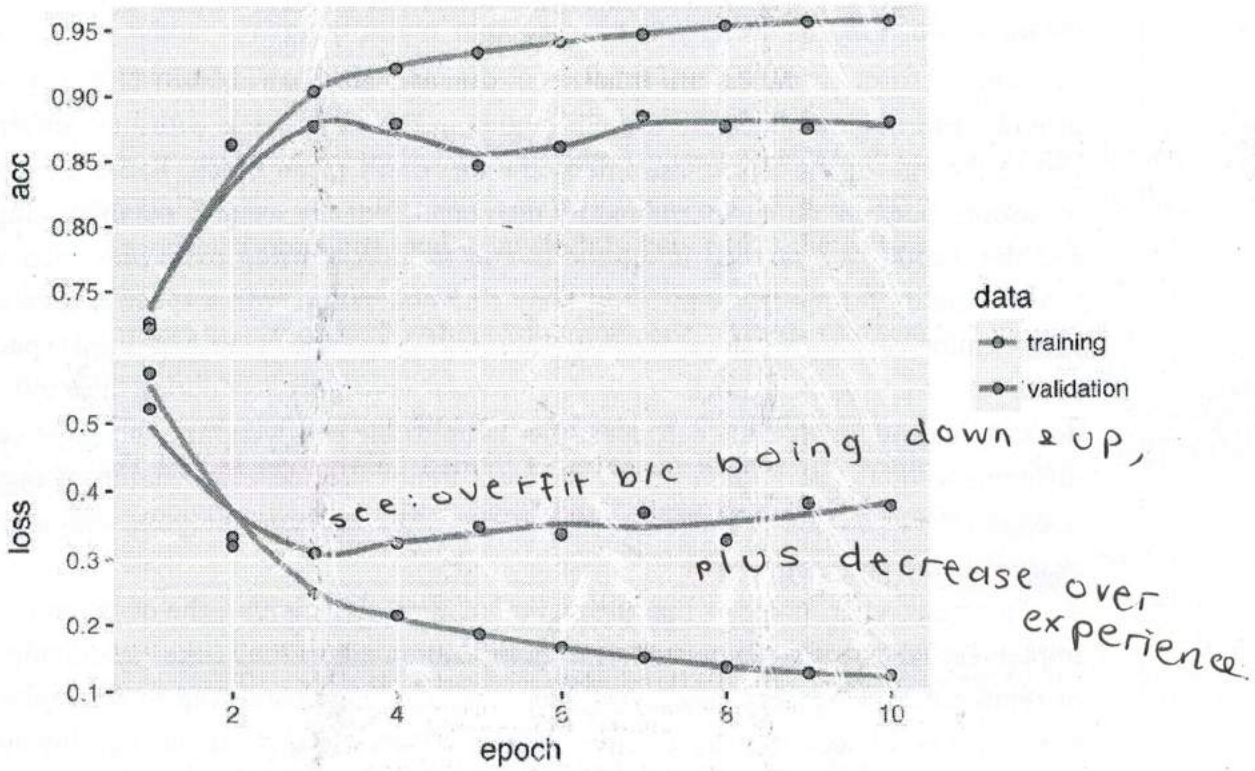

Figure 6.13 Training and validation metrics on IMDB with LSTM

This time, you achieve up to 88% validation accuracy. Not bad: certainly much better than the simple RNN network—that's largely because LSTM suffers much less from the vanishing-gradient problem—and even slightly better than the fully connected approach from chapter 3, even though you're looking at less data than you were in chapter 3. You're truncating sequences after 500 timesteps, whereas in chapter 3, you were considering full sequences.

But this result isn't groundbreaking for such a computationally intensive approach. Why isn't LSTM performing better? One reason is that you made no effort to tune hyperparameters such as the embeddings dimensionality or the LSTM output dimensionality. Another may be lack of regularization. But honestly, the primary reason is that analyzing the global, long-term structure of the reviews (what LSTM is good at)

isn't helpful for a sentiment-analysis problem. Such a basic problem is well solved by looking at what words occur in each review, and at what frequency. That's what the first fully connected approach looked at. But there are far more difficult natural-language-processing problems out there, where the strength of LSTM will become apparent: in particular, answering questions and machine translation.

6.2.4 Wrapping up

Now you understand the following:

- What RNNs are and how they work
- What LSTM is, and why it works better on long sequences than a naive RNN
- How to use Keras RNN layers to process sequence data

Next, we'll review a number of more advanced features of RNNs, which can help you get the most out of your deep-learning sequence models.

6.3 Advanced use of recurrent neural networks

In this section, we'll review three advanced techniques for improving the performance and generalization power of recurrent neural networks. By the end of the section, you'll know most of what there is to know about using recurrent networks with Keras. We'll demonstrate all three concepts on a weather-forecasting problem, where you have access to a timeseries of data points coming from sensors installed on the roof of a building, such as temperature, air pressure, and humidity, which you use to predict what the temperature will be 24 hours after the last data point. This is a fairly challenging problem that exemplifies many common difficulties encountered when working with timeseries.

We'll cover the following techniques:

- *Recurrent dropout*—This is a specific, built-in way to use dropout to fight overfitting in recurrent layers.
- *Stacking recurrent layers*—This increases the representational power of the network (at the cost of higher computational loads).
- *Bidirectional recurrent layers*—These present the same information to a recurrent network in different ways, increasing accuracy and mitigating forgetting issues.

6.3.1 A temperature-forecasting problem

Until now, the only sequence data we've covered has been text data, such as the IMDB dataset and the Reuters dataset. But sequence data is found in many more problems than just language processing. In all the examples in this section, you'll be playing with a weather timeseries dataset recorded at the Weather Station at the Max-Planck-Institute for Biogeochemistry in Jena, Germany.[13]

Footnote 13 Olaf Kolle, www.bgc-jena.mpg.de/wetter.

In this dataset, 14 different quantities (such air temperature, atmospheric pressure, humidity, wind direction, and so on) were recorded every 10 minutes, over several years. The original data goes back to 2003, but this example is limited to data from 2009–2016.

This dataset is perfect for learning to work with numerical timeseries. You'll use it to build a model that takes as input some data from the recent past (a few days' worth of data points) and predicts the air temperature 24 hours in the future.

Download and uncompress the data as follows:

```
cd ~/Downloads
mkdir jena_climate
cd jena_climate
wget https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
unzip jena_climate_2009_2016.csv.zip
```

Let's look at the data.

Listing 6.28 Inspecting the data of the Jena weather dataset

```
library(tibble)
library(readr)

data_dir <- "~/Downloads/jena_climate"
fname <- file.path(data_dir, "jena_climate_2009_2016.csv")
data <- read_csv(fname)

> glimpse(data)

Observations: 420,551
Variables: 15
$ `Date Time`       <chr> "01.01.2009 00:10:00", "01.01.2009 00:20:00", "...
$ `p (mbar)`        <dbl> 996.52, 996.57, 996.53, 996.51, 996.51, 996.50,...
$ `T (degC)`        <dbl> -8.02, -8.41, -8.51, -8.31, -8.27, -8.05, -7.62...
$ `Tpot (K)`        <dbl> 265.40, 265.01, 264.91, 265.12, 265.15, 265.38,...
$ `Tdew (degC)`     <dbl> -8.90, -9.28, -9.31, -9.07, -9.04, -8.78, -8.30...
$ `rh (%)`          <dbl> 93.3, 93.4, 93.9, 94.2, 94.1, 94.4, 94.8, 94.4,...
$ `VPmax (mbar)`    <dbl> 3.33, 3.23, 3.21, 3.26, 3.27, 3.33, 3.44, 3.44,...
$ `VPact (mbar)`    <dbl> 3.11, 3.02, 3.01, 3.07, 3.08, 3.14, 3.26, 3.25,...
$ `VPdef (mbar)`    <dbl> 0.22, 0.21, 0.20, 0.19, 0.19, 0.19, 0.18, 0.19,...
$ `sh (g/kg)`       <dbl> 1.94, 1.89, 1.88, 1.92, 1.92, 1.96, 2.04, 2.03,...
$ `H2OC (mmol/mol)` <dbl> 3.12, 3.03, 3.02, 3.08, 3.09, 3.15, 3.27, 3.26,...
$ `rho (g/m**3)`    <dbl> 1307.75, 1309.80, 1310.24, 1309.19, 1309.00, 13...
$ `wv (m/s)`        <dbl> 1.03, 0.72, 0.19, 0.34, 0.32, 0.21, 0.18, 0.19,...
$ `max. wv (m/s)`   <dbl> 1.75, 1.50, 0.63, 0.50, 0.63, 0.63, 0.63, 0.50,...
$ `wd (deg)`        <dbl> 152.3, 136.1, 171.6, 198.0, 214.3, 192.7, 166.5...
```

Here is the plot of temperature (in degrees Celsius) over time (see figure 6.14). On this plot, you can clearly see the yearly periodicity of temperature.

Listing 6.29 Plotting the temperature timeseries

```
library(ggplot2)
ggplot(data, aes(x = 1:nrow(data), y = `T (degC)`)) + geom_line()
```

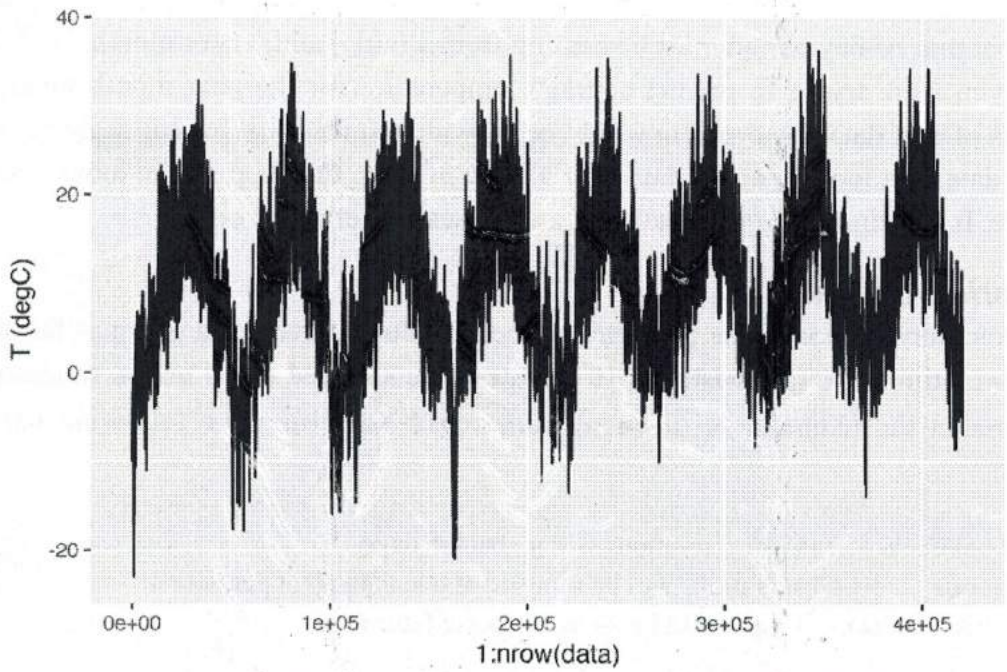

Figure 6.14 Temperature over the full temporal range of the dataset (°C)

Here is a more narrow plot of the first 10 days of temperature data (see figure 6.15). Because the data is recorded every 10 minutes, you get 144 data points per day.

Listing 6.30 Plotting the first 10 days of the temperature timeseries

```
ggplot(data[1:1440,], aes(x = 1:1440, y = `T (degC)`)) + geom_line()
```

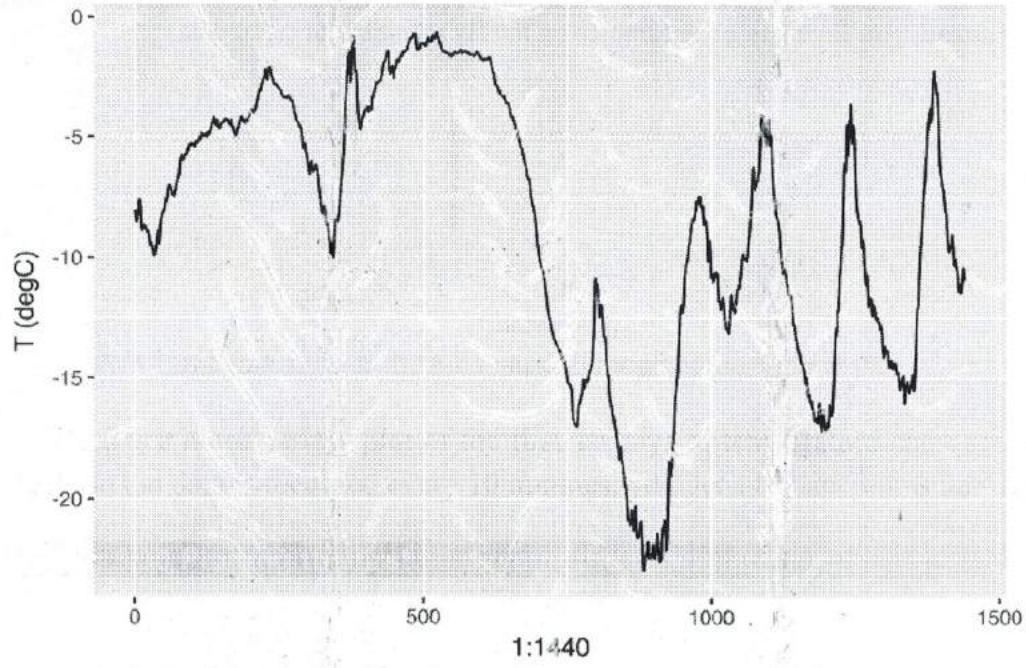

Figure 6.15 Temperature over the first 10 days of the dataset (°C)

On this plot, you can see daily periodicity, especially evident for the last 4 days. Also

note that this 10-day period must be coming from a fairly cold winter month.

If you were trying to predict average temperature for the next month given a few months of past data, the problem would be easy, due to the reliable year-scale periodicity of the data. But looking at the data over a scale of days, the temperature looks a lot more chaotic. Is this timeseries predictable at a daily scale? Let's find out.

6.3.2 Preparing the data

The exact formulation of the problem will be as follows: given data going as far back as `lookback` timesteps (a timestep is 10 minutes) and sampled every `steps` timesteps, can you predict the temperature in `delay` timesteps? You'll use the following parameter values:

- `lookback = 720`—Observations will go back 5 days.
- `steps = 6`—Observations will be sampled at one data point per hour.
- `delay = 144`—Targets will be 24 hours in the future.

To get started, you need to do two things:

- Preprocess the data to a format a neural network can ingest. This is easy: the data is already numerical, so you don't need to do any vectorization. But each timeseries in the data is on a different scale (for example, temperature is typically between -20 and +30, but pressure, measured in mbar, is around 1,000). You'll normalize each timeseries independently so that they all take small values on a similar scale.
- Write a generator function that takes the current array of float data and yields batches of data from the recent past, along with a target temperature in the future. Because the samples in the dataset are highly redundant (sample N and sample $N + 1$ will have most of their timesteps in common), it would be wasteful to explicitly allocate every sample. Instead, you'll generate the samples on the fly using the original data.

NOTE **Understanding generator functions**

A generator function is a special type of function that you call repeatedly to obtain a sequence of values from. Often generators need to maintain internal state, so they are typically constructed by calling another yet another function which returns the generator function (the environment of the function which returns the generator is then used to track state).

For example, the `sequence_generator()` function below returns a generator function that yields an infinite sequence of numbers:

```
sequence_generator <- function(start) {
  value <- start - 1
  function() {
    value <<- value + 1
    value
  }
}

> gen <- sequence_generator(10)
> gen()
[1] 10
> gen()
[1] 11
```

The current state of the generator is the `value` variable that is defined outside of the function. Note that superassignment (`<<-`) is used to update this state from within the function.

Generator functions can signal completion by returning the value `NULL`. However, generator functions passed to Keras training methods (e.g. `fit_generator()`) should always return values infinitely (the number of calls to the generator function is controlled by the `epochs` and `steps_per_epoch` parameters).

First, you'll convert the R data frame which we read earlier into a matrix of floating point values (we'll discard the first column which included a text timestamp):

Listing 6.31 Converting the data to a floating point matrix

```
data <- data.matrix(data[,-1])
```

You'll then preprocess the data by subtracting the mean of each timeseries and dividing by the standard deviation. You're going to use the first 200,000 timesteps as training data, so compute the mean and standard deviation for normalization only on this fraction of the data.

Listing 6.32 Normalizing the data

```
train_data <- data[1:200000,]
mean <- apply(train_data, 2, mean)
std <- apply(train_data, 2, sd)
```

```
data <- scale(data, center = mean, scale = std)
```

Listing 6.33 shows the data generator you'll use. It yields a list `(samples, targets)`, where `samples` is one batch of input data and `targets` is the corresponding array of target temperatures. It takes the following arguments:

- `data`—The original array of floating-point data, which you normalized in listing 6.32.
- `lookback`—How many timesteps back the input data should go.
- `delay`—How many timesteps in the future the target should be.
- `min_index` and `max_index`—Indices in the `data` array that delimit which timesteps to draw from. This is useful for keeping a segment of the data for validation and another for testing.
- `shuffle`—Whether to shuffle the samples or draw them in chronological order.
- `batch_size`—The number of samples per batch.
- `step`—The period, in timesteps, at which you sample data. You'll set it 6 in order to draw one data point every hour.

Listing 6.33 Generator yielding timeseries samples and their targets

```
generator <- function(data, lookback, delay, min_index, max_index,
                      shuffle = FALSE, batch_size = 128, step = 6) {
  if (is.null(max_index))
    max_index <- nrow(data) - delay - 1
  i <- min_index + lookback
  function() {
    if (shuffle) {
      rows <- sample(c((min_index+lookback):max_index), size = batch_size)
    } else {
      if (i + batch_size >= max_index)
        i <<- min_index + lookback
      rows <- c(i:min(i+batch_size, max_index))
      i <<- i + length(rows)
    }

    samples <- array(0, dim = c(length(rows),
                                lookback / step,
                                dim(data)[[-1]]))
    targets <- array(0, dim = c(length(rows)))

    for (j in 1:length(rows)) {
      indices <- seq(rows[[j]] - lookback, rows[[j]],
                     length.out = dim(samples)[[2]])
      samples[j,,] <- data[indices,]
      targets[[j]] <- data[rows[[j]] + delay,2]
    }

    list(samples, targets)
  }
}
```

The `i` variable contains the state that tracks next window of data to return, so it is updated using superassignment (e.g. `i < i + length(rows)`).

Now, let's use the abstract `generator` function to instantiate three generators: one for training, one for validation, and one for testing. Each will look at different temporal segments of the original data: the training generator looks at the first 200,000 timesteps, the validation generator looks at the following 100,000, and the test generator looks at the remainder.

Listing 6.34 Preparing the training, validation, and test generators

```
lookback <- 1440
step <- 6
delay <- 144
batch_size <- 128

train_gen <- generator(
  data,
  lookback = lookback,
  delay = delay,
  min_index = 1,
  max_index = 200000,
  shuffle = TRUE,
  step = step,
  batch_size = batch_size
)

val_gen = generator(
  data,
  lookback = lookback,
  delay = delay,
  min_index = 200001,
  max_index = 300000,
  step = step,
  batch_size = batch_size
)

test_gen <- generator(
  data,
  lookback = lookback,
  delay = delay,
  min_index = 300001,
  max_index = NULL,
  step = step,
  batch_size = batch_size
)

val_steps <- (300000 - 200001 - lookback) / batch_size    # 1

test_steps <- (nrow(data) - 300001 - lookback) / batch_size    # 2
```

1. How many steps to draw from val_gen in order to see the entire validation set
2. How many steps to draw from test_gen in order to see the entire test set

6.3.3 A common-sense, non-machine-learning baseline

Before you start using black-box deep-learning models to solve the temperature-prediction problem, let's try a simple, common-sense approach. It will serve as a sanity check, and it will establish a baseline that you'll have to beat in order to demonstrate the usefulness of more advanced machine-learning models. Such common-sense baselines can be useful when you're approaching a new problem for which there is no known solution (yet). A classic example is that of unbalanced classification tasks, where some classes are much more common than others. If your dataset contains 90% instances of class A and 10% instances of class B, then a common-sense approach to the classification task is to always predict "A" when presented with a new sample. Such a classifier is 90% accurate overall, and any learning-based approach should therefore beat this 90% score in order to demonstrate usefulness. Sometimes, such elementary baselines can prove surprisingly hard to beat.

In this case, the temperature timeseries can safely be assumed to be continuous (the temperatures tomorrow are likely to be close to the temperatures today) as well as periodical with a daily period. Thus a common-sense approach is to always predict that the temperature 24 hours from now will be equal to the temperature right now. Let's evaluate this approach, using the mean absolute error (MAE) metric:

```
mean(abs(preds - targets))
```

Here's the evaluation loop.

Listing 6.35 Computing the common-sense baseline MAE

```
evaluate_naive_method <- function() {
  batch_maes <- c()
  for (step in 1:val_steps) {
    c(samples, targets) %<-% val_gen()
    preds <- samples[,dim(samples)[[2]],2]
    mae <- mean(abs(preds - targets))
    batch_maes <- c(batch_maes, mae)
  }
  print(mean(batch_maes))
}

evaluate_naive_method()
```

This yields an MAE of 0.29. Because the temperature data has been normalized to be centered on 0 and have a standard deviation of 1, this number isn't immediately interpretable. It translates to an average absolute error of 0.29 * `temperature_std` degrees Celsius: 2.57C.

Listing 6.36 Converting the MAE back to a Celsius error

```
celsius_mae <- 0.29 * std[[2]]
```

That's a fairly large average absolute error. now the game is to use your knowledge of deep learning to do better.

6.3.4 A basic machine-learning approach

In the same way that it's useful to establish a common-sense baseline before trying machine-learning approaches, it's useful to try simple, cheap machine-learning models (such as small, densely connected networks) before looking into complicated and computationally expensive models such as RNNs. This is the best way to make sure any further complexity you throw at the problem is legitimate and delivers real benefits.

The following listing shows a fully connected model that starts by flattening the data and then runs it through two dense layers. Note the lack of activation function on the last dense layer, which is typical for a regression problem. You use MAE as the loss. Because you're evaluating on the exact same data and with the exact same metric you did with the common-sense approach, the results will be directly comparable.

Listing 6.37 Training and evaluating a densely connected model

```
library(keras)

model <- keras_model_sequential() %>%
  layer_flatten(input_shape = c(lookback / step, dim(data)[-1])) %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1)

model %>% compile(
  optimizer = optimizer_rmsprop(),
  loss = "mae"
)

history <- model %>% fit_generator(
  train_gen,
  steps_per_epoch = 500,
  epochs = 20,
  validation_data = val_gen,
  validation_steps = val_steps
)
```

Let's display the loss curves for validation and training (see figure 6.16).

Listing 6.38 Plotting results

```
plot(history)
```

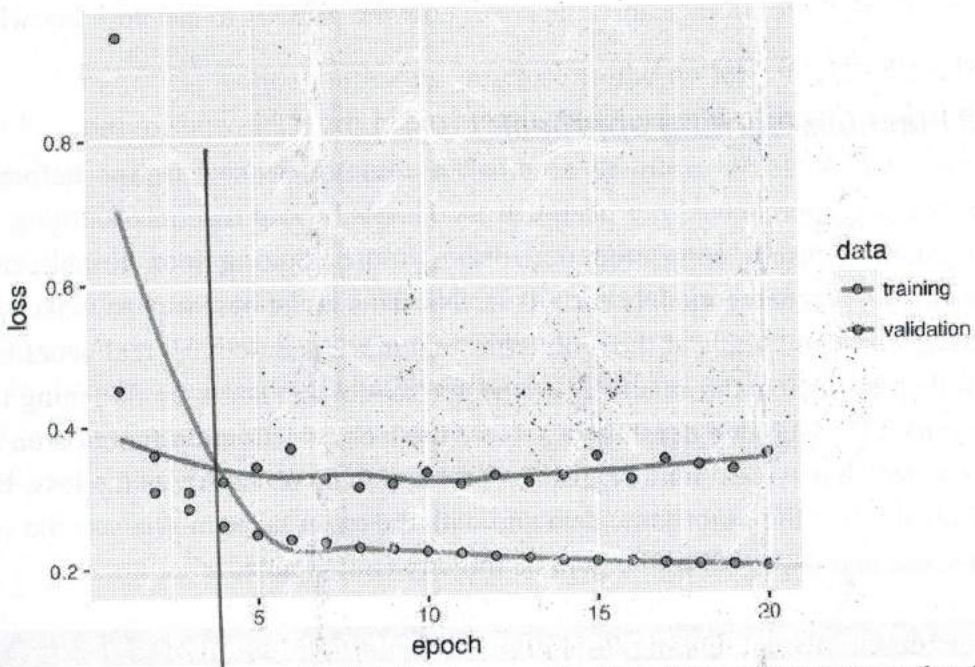

Figure 6.16 Training and validation loss on the Jena temperature forecasting task with a simple, densely connected network

Some of the validation losses are close to the no-learning baseline, but not reliably. This goes to show the merit of having this baseline in the first place: it turns out to be not easy to outperform. Your common sense contains a lot of valuable information that a machine-learning model doesn't have access to.

You may wonder, if a simple, well-performing model exists to go from the data to the targets (the common-sense baseline), why doesn't the model you're training find it and improve on it? Because this simple solution isn't what your training setup is looking for. The space of models in which you're searching for a solution—that is, your hypothesis space—is the space of all possible two-layer networks with the configuration you defined. These networks are already fairly complicated. When you're looking for a solution with a space of complicated models, the simple, well-performing baseline may be unlearnable, even if it's technically part of the hypothesis space. That is a pretty significant limitation of machine learning in general: unless the learning algorithm is hard-coded to look for a specific kind of simple model, parameter learning can sometimes fail to find a simple solution to a simple problem.

6.3.5 A first recurrent baseline

The first fully connected approach didn't do well, but that doesn't mean machine learning isn't applicable to this problem. The previous approach consisted first flattened the timeseries, which removed the notion of time from the input data. Let's instead look at the data as what it is: a sequence, where causality and order matter. You'll try a recurrent-sequence processing model—it should be the perfect fit for such sequence data, precisely because it exploits the temporal ordering of data points, unlike the first approach.

Instead of the LSTM layer introduced in the previous section, you'll use the GRU

layer, developed by Chung et al. in 2014.[14] Gated recurrent unit (GRU) layers work using the same principle as LSTM, but they're somewhat streamlined and thus cheaper to run (although they may not have as much representational power as LSTM). This trade-off between computational expensiveness and representational power is seen everywhere in machine learning.

Footnote 14 Junyoung Chung et al., "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling," Conference on Neural Information Processing Systems (2014), arxiv.org/abs/1412.3555.

Listing 6.39 Training and evaluating a model with `layer_gru`

```
model <- keras_model_sequential() %>%
  layer_gru(units = 32, input_shape = list(NULL, dim(data)[[-1]])) %>%
  layer_dense(units = 1)

model %>% compile(
  optimizer = optimizer_rmsprop(),
  loss = "mae"
)

history <- model %>% fit_generator(
  train_gen,
  steps_per_epoch = 500,
  epochs = 20,
  validation_data = val_gen,
  validation_steps = val_steps
)
```

Figure 6.17 shows the results. Much better! You can significantly beat the common-sense baseline, demonstrating the value of machine learning as well as the superiority of recurrent networks compared to sequence-flattening dense networks on this type of task.

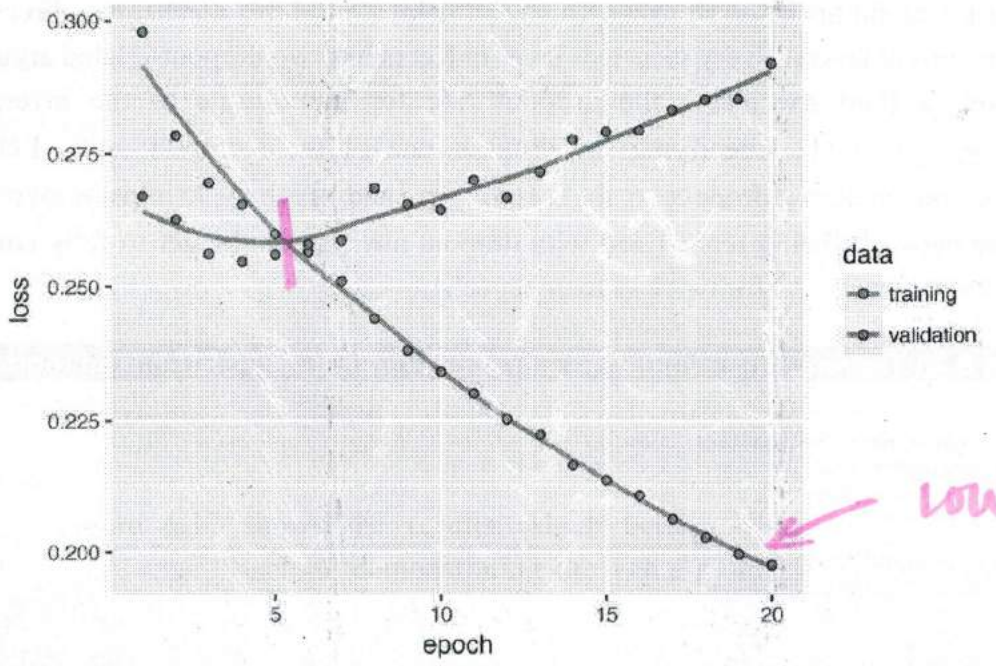

Figure 6.17 Training and validation loss on the Jena temperature-forecasting task with layer_gru

The new validation MAE of ~0.265 (before you start significantly overfitting) translates to a mean absolute error of 2.35C after denormalization. That's a solid gain on the initial error of 2.57C, but you probably still have a bit of a margin for improvement.

6.3.6 Using recurrent dropout to fight overfitting

It's evident from the training and validation curves that the model is overfitting: the training and validation losses start to diverge considerably after a few epochs. You're already familiar with a classic technique for fighting this phenomenon: dropout, which randomly zeros out input units of a layer in order to break happenstance correlations in the training data that the layer is exposed to. But how to correctly apply dropout in recurrent networks isn't a trivial question. It has long been known that applying dropout before a recurrent layer hinders learning rather than helping with regularization. In 2015, Yarin Gal, as part of his Ph.D. thesis on Bayesian deep learning,[15] determined the proper way to use dropout with a recurrent network: the same dropout mask (the same pattern of dropped units) should be applied at every timestep, instead of a dropout mask that varies randomly from timestep to timestep. What's more, in order to regularize the representations formed by the recurrent gates of layers such as `layer_gru` and `layer_lstm`, a temporally constant dropout mask should be applied to the inner recurrent activations of the layer (a *recurrent* dropout mask). Using the same dropout mask at every timestep allows the network to properly propagate its learning error through time; a temporally random dropout mask would disrupt this error signal and be harmful to the learning process.

Footnote 15 See Yarin Gal, "Uncertainty in Deep Learning (PhD Thesis)," October 13, 2016, mlg.eng.cam.ac.uk/yarin/blog_2248.html.

Yarin Gal did his research using Keras and helped build this mechanism directly into Keras recurrent layers. Every recurrent layer in Keras has two dropout-related arguments: `dropout`, a float specifying the dropout rate for input units of the layer, and `recurrent_dropout`, specifying the dropout rate of the recurrent units. Let's add dropout and recurrent dropout to the `layer_gru` and see how it impacts overfitting. Because networks being regularized with dropout always take longer to fully converge, you'll train the network for twice as many epochs.

Listing 6.40 Training and evaluating a dropout-regularized GRU-based model

```
model <- keras_model_sequential() %>%
  layer_gru(units = 32, dropout = 0.2, recurrent_dropout = 0.2,
            input_shape = list(NULL, dim(data)[[-1]])) %>%
  layer_dense(units = 1)

model %>% compile(
  optimizer = optimizer_rmsprop(),
  loss = "mae"
)

history <- model %>% fit_generator(
  train_gen,
```

```
  steps_per_epoch = 500,
  epochs = 40,
  validation_data = val_gen,
  validation_steps = val_steps
)
```

Figure 6.18 shows the results. Success! You're no longer overfitting during the first 20 epochs. But although you have more stable evaluation scores, your best scores aren't much lower than they were previously.

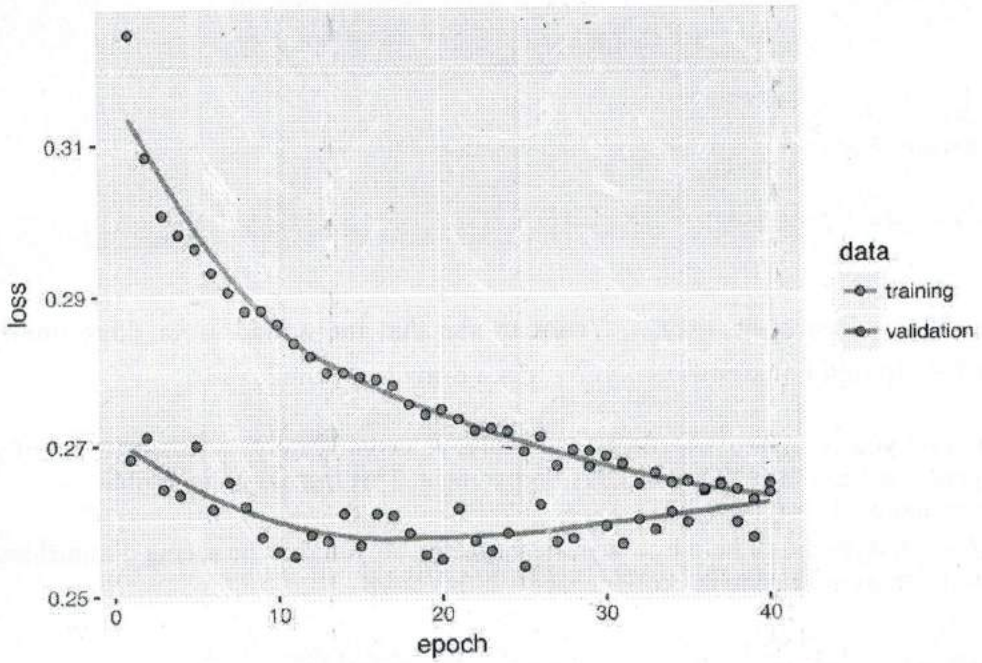

Figure 6.18 Training and validation loss on the Jena temperature forecasting task with a dropout-regularized GRU

6.3.7 Stacking recurrent layers

Because you're no longer overfitting but seem to have hit a performance bottleneck, you should consider increasing the capacity of the network. Recall our description of the universal machine-learning workflow: it's generally a good idea to increase the capacity of your network until overfitting becomes the primary obstacle (assuming you're already taking basic steps to mitigate overfitting, such as using dropout). As long as you aren't overfitting too badly, you're likely under capacity.

Increasing network capacity is typically done by increasing the number of units in the layers or adding more layers. Recurrent layer stacking is a classic way to build more powerful recurrent networks: for instance, what currently powers the Google Translate algorithm is a stack of seven large LSTM layers—that's huge.

To stack recurrent layers on top of each other in Keras, all intermediate layers should return their full sequence of outputs (a 3D tensor) rather than their output at the last timestep. This is done by specifying `return_sequences = TRUE`.

Listing 6.41 Training and evaluating a dropout-regularized, stacked GRU model

```
model <- keras_model_sequential() %>%
  layer_gru(units = 32,
            dropout = 0.1,
            recurrent_dropout = 0.5,
            return_sequences = TRUE,
            input_shape = list(NULL, dim(data)[[-1]])) %>%
  layer_gru(units = 64, activation = "relu",
            dropout = 0.1,
            recurrent_dropout = 0.5) %>%
  layer_dense(units = 1)

model %>% compile(
  optimizer = optimizer_rmsprop(),
  loss = "mae"
)

history <- model %>% fit_generator(
  train_gen,
  steps_per_epoch = 500,
  epochs = 40,
  validation_data = val_gen,
  validation_steps = val_steps
)
```

Figure 6.19 shows the results. You can see that the added layer does improve the results a bit, though not significantly. You can draw two conclusions:

- Because you're still not overfitting too badly, you could safely increase the size of your layers in a quest for validation loss improvement. This has a non-negligible computational cost, though.
- Adding a layer didn't help by a significant factor, so you may be seeing diminishing returns from increasing network capacity at this point.

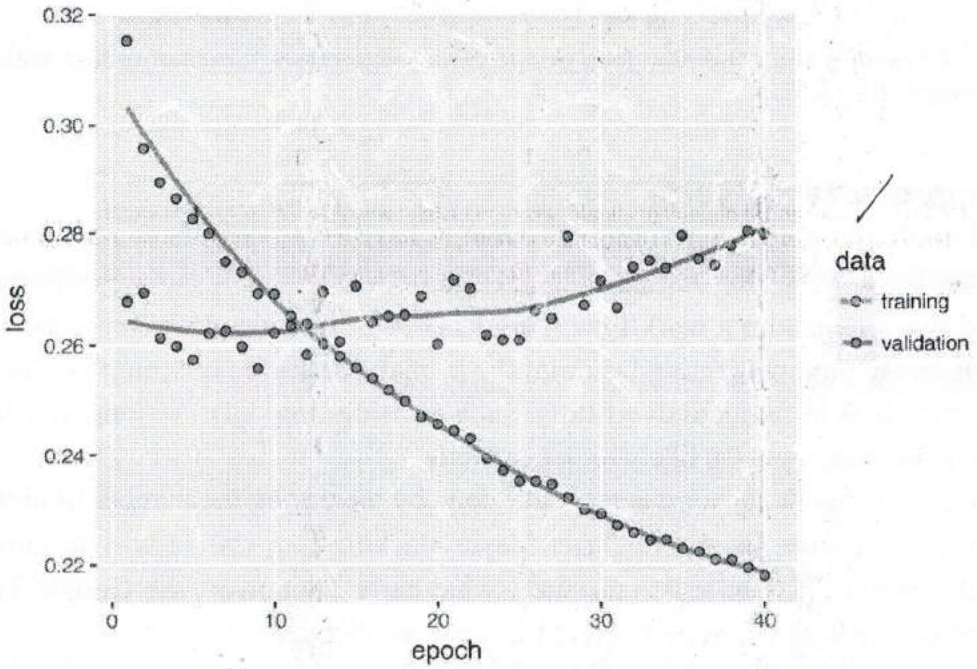

Figure 6.19 Training and validation loss on the Jena temperature forecasting task with a stacked GRU network