Deep Learning for text data (1)

Contents

Working with text data **1**
Text vectorization (Encoding) 1

Word embeddings **2**
Learn word embeddings 3
Example. Using an embedding layer and classifier on the IMDB data 4
Pretrained word embeddings 6
From raw text to word embeddings 7

Working with text data

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.

We explore 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 previously. We'll discuss both of these approaches in this unit.

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
- Timeseries comparisons, such as estimating how closely related two documents or two stock tickers are
- Sequence-to-sequence learning, such as decoding an English sentence into French
- Sentiment analysis, such as classifying the sentiment of tweets or movie reviews as positive or negative
- Timeseries forecasting, such as predicting the future weather at a certain location, given recent weather data

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 unit 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.

Text vectorization (Encoding)

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. The two major ones are: *one-hot encoding* of tokens, and *token embedding* (typically used exclusively for words, and called *word embedding*).

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). 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.

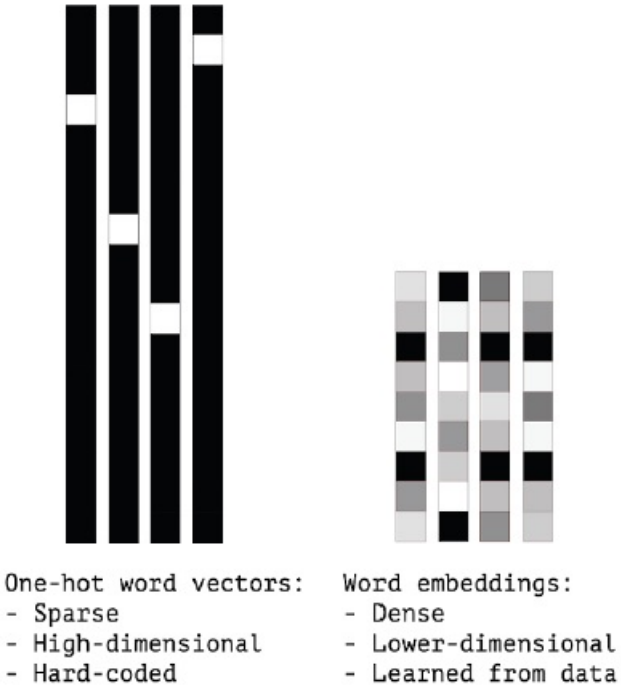


Figure 1: Word representations

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.

- Pretrained word embeddings. Load into your model word embeddings that were precomputed using a different machine-learning task than the one you're trying to solve.

Learn word embeddings

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 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()`.

The embedding layer takes at least two arguments:

- the number of possible tokens, and
- the dimensionality of the embeddings.

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.

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.

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.

Example. Using an embedding layer and classifier on the IMDB data

IMDB Movie reviews sentiment classification

Description: Dataset of 25,000 movies reviews from IMDB, labeled by sentiment (positive/negative). Reviews have been **preprocessed**, and each review is encoded as a sequence of word indexes (integers). For convenience, words are indexed by overall frequency in the dataset, so that for instance the integer "3" encodes the 3rd most frequent word in the data.

```
max_features<-10000  # Max number of words to include
maxlen<-20 # sequences longer than this will be filtered out
```

```
imdb<-dataset_imdb(num_words = max_features)
```

```
c(c(x_train,y_train),c(x_test,y_test))%<-%imdb
class(x_train)
```

```
## [1] "list"
```

```
class(y_train)
```

```
## [1] "integer"
```

```
x_train[[1]]
```

```
##   [1]    1   14   22   16   43  530  973 1622 1385   65  458 4468   66 3941    4
##  [16]  173   36  256    5   25  100   43  838  112   50  670    2    9   35  480
##  [31]  284    5  150    4  172  112  167    2  336  385   39    4  172 4536 1111
##  [46]   17  546   38   13  447    4  192   50   16    6  147 2025   19   14   22
##  [61]    4 1920 4613  469    4   22   71   87   12   16   43  530   38   76   15
##  [76]   13 1247    4   22   17  515   17   12   16  626   18    2    5   62  386
##  [91]   12    8  316    8  106    5    4 2223 5244   16  480   66 3785   33    4
## [106]  130   12   16   38  619    5   25  124   51   36  135   48   25 1415   33
## [121]    6   22   12  215   28   77   52    5   14  407   16   82    2    8    4
## [136]  107  117 5952   15  256    4    2    7 3766    5  723   36   71   43  530
## [151]  476   26  400  317   46    7    4    2 1029   13  104   88    4  381   15
## [166]  297   98   32 2071   56   26  141    6  194 7486   18    4  226   22   21
## [181]  134  476   26  480    5  144   30 5535   18   51   36   28  224   92   25
## [196]  104    4  226   65   16   38 1334   88   12   16  283    5   16 4472  113
## [211]  103   32   15   16 5345   19  178   32
```

```
x_train<-pad_sequences(x_train,maxlen=maxlen)  # 2D integer tensor of shape (samples, maxlen)
x_test<-pad_sequences(x_test,maxlen=maxlen) # 2D integer tensor of shape (samples, maxlen)
dim(x_train)
```

```
## [1] 25000    20
```

```
x_train[1,]  # the last twenty (maxlen)
```

```
##  [1]   65   16   38 1334   88   12   16  283    5   16 4472  113  103   32   15
## [16]   16 5345   19  178   32
```

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.

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


summary(model)
```

```
## Model: "sequential"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## embedding (Embedding)               (None, 20, 8)                   80000
## ________________________________________________________________________________
## flatten (Flatten)                   (None, 160)                     0
## ________________________________________________________________________________
## dense (Dense)                       (None, 1)                       161
## ================================================================================
## Total params: 80,161
## Trainable params: 80,161
## Non-trainable params: 0
## ________________________________________________________________________________
```

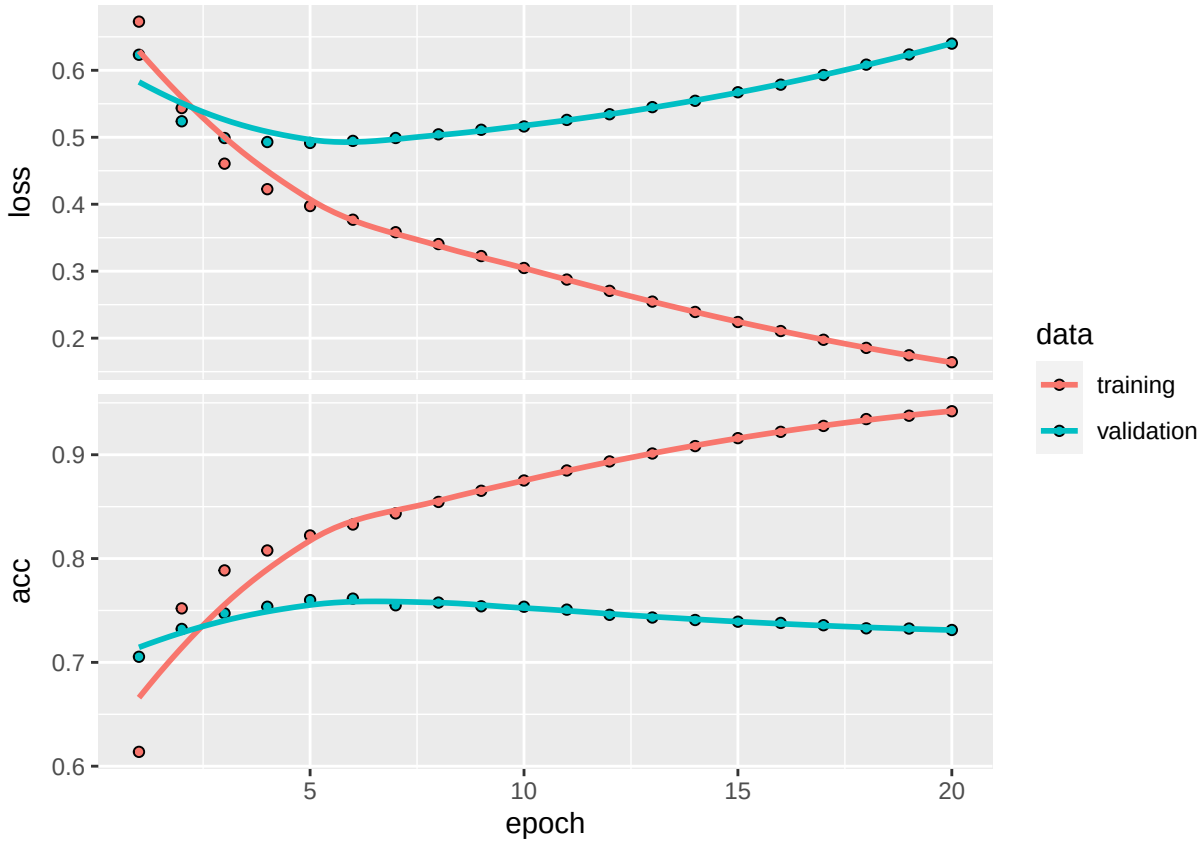
Notice that the embedding layer have 80000=10000x8 weights. Thus, network learns 8-dimensional embeddings for each of the 10000 words.

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

```
history <- model %>% fit(
  x_train,y_train,
  epochs = 20,
  batch_size = 32,
  validation_split = 0.2
)
```

```
plot(history)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



You get to a validation accuracy of ~76%, which is pretty good considering that you're only looking at 20 words from each 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 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.

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, 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, developed by Tomas Mikolov at Google in 2013.

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), 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.

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.

From raw text to word embeddings

```
imdb_dir<-"~/Docencia/curs21_22/UB/MESIO/DL/unitat5_RNN/imdb/aclImdb"
train_dir<-file.path(imdb_dir,"train")

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

# Takes a couple of minutes
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,full.names = TRUE)) {
    texts <- c(texts, readChar(fname, file.info(fname)$size))
    labels <- c(labels, label)
  }
}
```

```
texts[1]
```

```
## [1] "Story of a man who has unnatural feelings for a pig. Starts out with a opening scene that is a
```

```
labels[1]
```

```
## [1] 0
```

```
maxlen <- 50
max_words <- 10000
```

Tokenizing the data

```
tokenizer <- text_tokenizer(num_words = max_words) %>%
  fit_text_tokenizer(texts)
sequences <- texts_to_sequences(tokenizer, texts)
word_index = tokenizer$word_index  # les paraules que tenim en la nostra base de dades
#names(word_index)

sequences[1]
```

```
## [[1]]
##   [1]   62    4    3  129   34   44 7576 1414   15    3 4252  514   43   16    3
##  [16]  633  133   12    6    3 1301  459    4 1751  209    3 7693  308    6  676
```

```
##  [31]   80   32 2137 1110 3008   31    1  929    4   42 5120  469    9 2665 1751
##  [46]    1  223   55   16   54  828 1318  847  228    9   40   96  122 1484   57
##  [61]  145   36    1  996  141   27  676  122    1  411   59   94 2278  303  772
##  [76]    5    3  837   20    3 1755  646   42  125   71   22  235  101   16   46
##  [91]   49  624   31  702   84  702  378 3493    2 8422   67   27  107 3348
```

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

```
## Found 88584 unique tokens.
```

```
data <- pad_sequences(sequences, maxlen = maxlen)
```

```
dim(data)
```

```
## [1] 25000    50
```

```
data[1,]
```

```
##  [1]    9   40   96  122 1484   57  145   36    1  996  141   27  676  122    1
## [16]  411   59   94 2278  303  772    5    3  837   20    3 1755  646   42  125
## [31]   71   22  235  101   16   46   49  624   31  702   84  702  378 3493    2
## [46] 8422   67   27  107 3348
```

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

```
## Shape of data tensor: 25000 50
```

```
cat('Shape of label tensor:', dim(labels), "\n")
```

```
## Shape of label tensor: 25000
```

```
training_samples <- 200     # small subset of examples
validation_samples <- 10000

indices <- sample(1:nrow(data))
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]
```

Word embedding

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.

```
glove_dir<-"~/Docencia/curs21_22/UB/MESIO/DL/unitat5_RNN/imdb/glove6B"
lines <- readLines(file.path(glove_dir, "glove.6B.100d.txt"))

# el 100-vector de coordenades de 400000 paraules en angles
#lines[50] # les coordenades del after


embeddings_index <- new.env(hash = TRUE, parent = emptyenv())

# processat per obtenir un format llista del word embedding
```

```
for (i in 1:length(lines)) {
  line <- lines[[i]]
  values <- strsplit(line, " ")[[1]]
  word <- values[[1]]  # la paraula
  embeddings_index[[word]] <- as.double(values[-1]) # les coordenades de la paraula
}
cat("Found", length(embeddings_index), "word vectors.\n")
```

```
## Found 400000 word vectors.
```

Examples

```
embeddings_index[["after"]]
```

```
##   [1]  0.3771100 -0.3447100  0.1340500 -0.0117100 -0.1942700  0.4146400
##   [7]  0.4060800  0.4306300 -0.0570600 -0.1992100  0.4326700 -0.0162690
##  [13]  0.2171000 -0.0026149  0.3942400 -0.4280300 -0.0174950 -0.5665800
##  [19] -0.4455800 -0.1852900  0.2673200 -0.1571200  0.2165700  0.7971400
##  [25]  0.6962300  0.2040500 -0.4990700 -0.4551900  0.3821000  0.2060300
##  [31] -0.2160600  0.1009300 -0.5014800 -0.1105800 -0.4345500 -0.2678500
##  [37] -0.2023400  0.0038320 -0.4910800 -0.1764200 -0.8897100 -0.2790000
##  [43]  0.8638700 -0.0173560  0.3121000  0.4100400  0.2319900 -0.6081200
##  [49]  0.4476300 -0.8957900 -0.0384910 -0.2577200  0.3946800  1.6186000
##  [55] -0.5488200 -3.0291000 -0.7784500 -0.3246300  1.7658000  0.9730300
##  [61] -0.3934200  0.5481100  0.0131640  0.3785000  0.2453800  0.0310790
##  [67]  0.2362800  0.2890100  0.0270470  0.2898500 -0.7452300  0.0115170
##  [73] -0.3945600 -0.5770600 -0.6360400  0.3102200 -0.3831700 -0.0776630
##  [79] -1.3539000  0.0180090  0.8564600  0.0382590 -0.3943700  0.4433100
##  [85] -1.0802000 -0.4315900  0.1439100  0.1185400 -0.5645900 -0.4796600
##  [91]  0.2286000 -0.2436900 -0.4282300  1.0366000 -0.8307100  0.1246000
##  [97]  0.2063000  0.5423200  0.1142500 -0.6692700
```

```
embeddings_index[["is"]]
```

```
##   [1] -0.5426400  0.4147600  1.0322000 -0.4024400  0.4669100  0.2181600
##   [7] -0.0748640  0.4733200  0.0809960 -0.2207900 -0.1280800 -0.1144000
##  [13]  0.5089100  0.1156800  0.0282110 -0.3628000  0.4382300  0.0475110
##  [19]  0.2028200  0.4985700 -0.1006800  0.1326900  0.1697200  0.1165300
##  [25]  0.3135500  0.2571300  0.0927830 -0.5682600 -0.5297500 -0.0514560
##  [31] -0.6732600  0.9253300  0.2693000  0.2273400  0.6636500  0.2622100
##  [37]  0.1971900  0.2609000  0.1877400 -0.3454000 -0.4263500  0.1397500
##  [43]  0.5633800 -0.5690700  0.1239800 -0.1289400  0.7248400 -0.2610500
##  [49] -0.2631400 -0.4360500  0.0789080 -0.8414600  0.5159500  1.3997000
##  [55] -0.7646000 -3.1453000 -0.2920200 -0.3124700  1.5129000  0.5243500
##  [61]  0.2145600  0.4245200 -0.0884110 -0.1780500  1.1876000  0.1057900
##  [67]  0.7657100  0.2191400  0.3582400 -0.1163600  0.0932610 -0.6248300
##  [73] -0.2189800  0.2179600  0.7405600 -0.4373500  0.1434300  0.1471900
##  [79] -1.1605000 -0.0505080  0.1267700 -0.0143950 -0.9867600 -0.0912970
##  [85] -1.2054000 -0.1197400  0.0478470 -0.5400100  0.5245700 -0.7096300
##  [91] -0.3252800 -0.1346000 -0.4131400  0.3343500 -0.0072412  0.3225300
##  [97] -0.0442190 -1.2969000  0.7621700  0.4634900
```

```
embeddings_index[["sky"]]
```

```
##   [1] -0.1972800 -0.1933200  0.4700300 -0.2034900  0.3532000 -0.0019670
##   [7] -0.1606100  0.3437600 -0.4712300 -0.1380500  0.4084700 -0.2754000
```

```
##  [13]  0.2601900 -0.2364600  0.3346400 -1.0288000  0.2665800  0.2653400
##  [19]  0.3347400  0.0356790 -0.4613300 -0.1142700 -0.3871700 -0.3867900
##  [25]  1.2251000  0.5477800  0.4922000  0.5493100 -0.4144200 -0.4091900
##  [31] -0.0037017 -0.1153000 -0.3067600  0.2577000  0.5285500  0.1668900
##  [37] -0.0057968  0.4064500  0.5322900 -0.3034000 -0.6427200 -0.0968670
##  [43] -1.0288000  0.5903300  0.2534700  0.4018800 -0.3599100  0.6805600
##  [49]  0.3915600 -0.6984500 -0.1170300  0.3048400  0.4127800  1.1828000
##  [55]  0.0278790 -2.5020000  0.5065100  0.9917900  1.3479000  0.2901800
##  [61]  0.2911700  0.8186800 -0.7085700 -0.4168900 -0.1447400  0.7120200
##  [67]  0.3346600 -0.3939300 -0.4236900  0.0064274  0.1589900 -0.4306600
##  [73]  0.8409400 -0.1740900 -0.0906750 -0.2506900  0.9342800  0.1916400
##  [79] -0.4131600 -0.0074316  0.5892800 -0.1279000  0.2514400 -0.2211400
##  [85]  0.3306500 -1.2090000 -0.0240410  0.0639580  0.8044700  0.7981900
##  [91]  0.3666700 -0.1914300 -0.2636900  0.2301300 -0.7819900  0.2088100
##  [97] -0.8535100  0.5592400 -0.1934500  0.0092169
```

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.

```
embedding_dim <- 100  # dimension compatible with pretrained embedding
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
  }
}
```

```
dim(embedding_matrix)
```

```
## [1] 10000   100
```

Defining the model

```
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)
```

```
## Model: "sequential_1"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## embedding_1 (Embedding)             (None, 50, 100)                 1000000
## ________________________________________________________________________________
## flatten_1 (Flatten)                 (None, 5000)                    0
## ________________________________________________________________________________
## dense_2 (Dense)                     (None, 32)                      160032
## ________________________________________________________________________________
## dense_1 (Dense)                     (None, 1)                       33
```

```
## ================================================================================
## Total params: 1,160,065
## Trainable params: 1,160,065
## Non-trainable params: 0
## ________________________________________________________________________________
```

Param of layer embedding are 1000000 equal to max_words=10000 times embedding_dim=100.

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.

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.

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

```
summary(model)
```

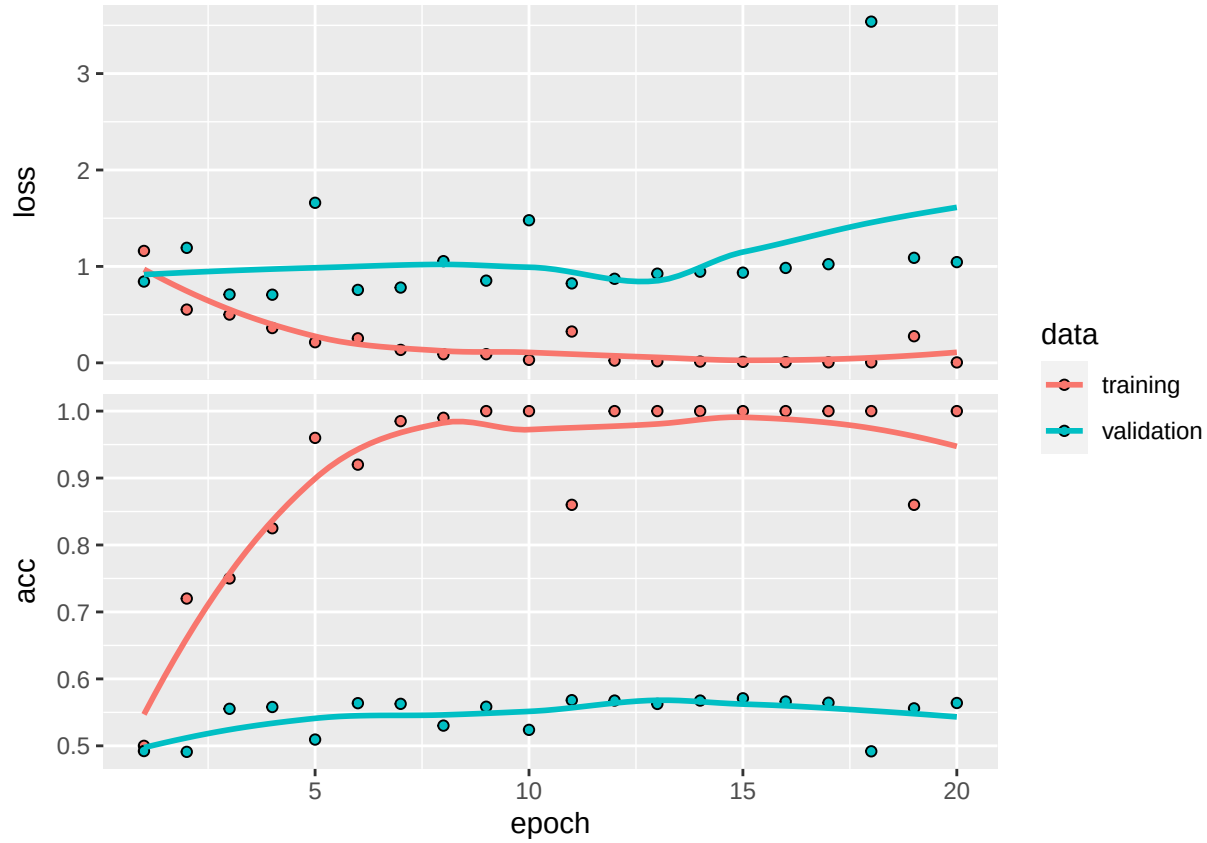
```
## Model: "sequential_1"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## embedding_1 (Embedding)             (None, 50, 100)                 1000000
## ________________________________________________________________________________
## flatten_1 (Flatten)                 (None, 5000)                    0
## ________________________________________________________________________________
## dense_2 (Dense)                     (None, 32)                      160032
## ________________________________________________________________________________
## dense_1 (Dense)                     (None, 1)                       33
## ================================================================================
## Total params: 1,160,065
## Trainable params: 160,065
## Non-trainable params: 1,000,000
## ________________________________________________________________________________
```

```
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)
)
```

```
plot(history)
```

```
## `geom_smooth()` using formula 'y ~ x'
```




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

Evaluating the model on the test set

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


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

```
##     loss      acc
## 1.046331 0.563400
```