Text

Description automatically generated

# Classifying movie reviews

* Two-class classification, or binary classification, is one of the most common kinds of machine learning problems. In this example, you’ll learn to classify movie reviews as positive or negative, based on the text content of the reviews

## The IMDB dataset

* The IMDB dataset contains 50,000 highly polarized reviews from the Internet Movie Database
  + They’re split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting of 50% negative and 50% positive reviews
  + IMDB dataset comes packaged with Keras. It has already been preprocessed: the reviews (sequences of words) have been turned into sequences of integers, where each integer stands for a specific word in a dictionary.
    - This enables us to focus on model building, training, and evaluation.
    - In chapter 11, you’ll learn how to process raw text input from scratch
* Load the data set (when first run it, about 80MB of data will be downloaded to the machine)

Text

Description automatically generated with low confidence

* + The argument num\_words = 10000 means you’ll only keep the top 10,000 most frequently occurring words in the training data. Rare words will be discarded
    - This allows us to work with vector data of manageable size.
    - If we didn’t set this limit, we’d be working with 88,585 unique words in the training data, which is unnecessarily large.
    - Many of these words only occur in a single sample, and thus can’t be meaningfully used for classification
  + The variables train\_data and test\_data are lists of reviews; each review is a list of word indices (encoding a sequence of words). train\_labels and test\_labels are lists of 0s and 1s, where 0 stands for negative and 1 stands for positive



* + Since we’re restricting ourselves to the top 10,000 most frequent words, no word index will exceed 10,000:



* + Here’s how you can quickly decode one of these reviews back to English words

Text

Description automatically generated

## Preparing the data

* You can’t directly feed lists of integers into a neural network. They all have different lengths, but a neural network expects to process continuous batches of data. You have to turn your lists into tensors. There are two ways to do that
  + Pad your lists so that they all have the same length, turn them into an integer tensor of shape (samples, max\_length), and start your model with a layer capable of handling such integer tensors (the *Embedding* layer, which we’ll cover in detail later in this book)
  + *Multi-hot encode* your lists to turn them into vectors of 0s and 1s. This would mean, for instance, turning the sequence [8, 5] into a 10,000-dimensional vector that would be all 0s except for indices 9 and 5, which would be 1s. Then you could use a *Dense* layer, capable of handling floating-point vector data, as the first layer in the model.
* Let’s go with the second option

Graphical user interface, text, application

Description automatically generated

* + Here’s what the samples look like now



* We should also vectorize the labels



* Now the data is ready to be fed into a neural network

## Building your model

* The input data is vectors, and the labels are scalars (1s and 0s). A type of model that performs well on such a problem is a plain stack of densely connected layers with relu activations
* There are two key architecture decisions to be made about such a stack of dense layers
  + How many layers to use
  + How many units to choose for each layer
* Diagram, schematic

  Description automatically generatedIn chapter 5, you’ll learn formal principles to guide you in making these choices. For the time being, you’ll have to trust me with the following architecture choices
  + Two intermediate layers with 16 units each
  + A third layer that will output the scalar prediction regarding the sentiment of the current review
* The following lists shows the Keras implementation, similar to the MNIST example you saw previously

Graphical user interface, text

Description automatically generated

* + Recall that, dense layer with a relu activation implements the following chain of tensor operations



* + Having 16 units means the weight matrix W will have shape (input\_dimension, 16): the dot product with W will project the input data onto a 16-disneisonal representation space (and then you’ll add the bias vector b and apply the relu operations).
    - You can intuitively understand the dimensionality of your representation space as “how much freedom you’re allowing the model to have when learning internal representations.”
    - Having more units allows your model to learn more-complex representations, but it makes the model more computationally expensive and may lead to learning unwanted patterns (patterns that will improve performance on the training data but not on the test data).
  + The intermediate layers use relu as their activation function, and the final layer uses a sigmoid activation so as to output a probability (how likely the sample is to have the target “1”)
    - A rectified linear unit is a function meant to zero out negative values, whereas a sigmoid “squashes” arbitrary values into the [0, 1] interval, outputting something that can be interpreted as a probability

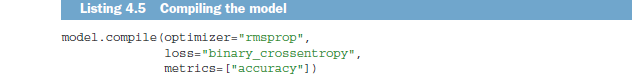
Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

* Finally, you need to choose a loss function and an optimizer. Because you’re facing a binary classification problem and the output of your model is probability, it’s best to use the binary\_crossentropy loss.
  + It isn’t the only viable choice: for instance, you could use mean\_squared\_error.
  + But crossentropy is usually the best choice when you’re dealing with models that output probabilities.
  + Crossentropy is a quantity from the field of information theory that measures the distance between probability distribution or, in this case, between the ground-truth distribution and your prediction.
* As for the choice of optimizer, we’ll go with RMSprop, which is a usually good default choice for virtually any problem.
* Here is how we would compile for the program



## Validating your approach

* As learned in chapter 3, a deep learning model should never be evaluated on its training data – it’s standard practice to use a validation set to monitor the accuracy of the model during training.
  + Here, we’ll create a validation set by setting apart 10,000 samples from the original training data.

Graphical user interface

Description automatically generated with medium confidence

* We will now train the model for 20 epochs (2o iterations over all samples in the training data) in mini-batches of 512 samples. At the same time, we will monitor loss and accuracy on the 10,000 samples that we set apart. We do so by passing the validation data as the validation\_data argument.

Text

Description automatically generated with medium confidence

* + On CPU, this will take less than 2 seconds per epoch – training is over in 20 seconds.
    - At the end of every epoch, there is a slight pause as the model computes its loss and accuracy on the 10,000 samples of the validation data
  + Note that the call to *model.fit( )* returns a *History* object.
    - This object has a member *history*, which is a dictionary containing data about everything that happened during training.



* + - The dictionary contains four entries: one er metric that was being monitored during training and during validation.
    - In the following two listings, let’s use Matplotlib to plot the training and validation loss side by side, and training and validation accuracy.
      * Note that your own results may vary slightly due to a different random initialization of your model

Text

Description automatically generated

Chart, line chart

Description automatically generated

Text

Description automatically generated

Chart

Description automatically generated

* + As you can see, the training loss decreases with every epoch, and the training accuracy increases with every epoch.
    - That’s what you would expect when running gradient descent optimization – the quantity you’re trying to minimize should be less with every iteration
  + But validation loss and accuracy seem to peak at the fourth epoch.
    - This is an example of what we warned against earlier: a model that performs better on the training data isn’t necessarily a model that will do better on data it has never seen before.
    - In precise terms, what you’re seeing is *overfitting*: overoptimizing on the training data, end up learning representations that are specific to the training data and don’t generalize to data outside of the training set.
  + In this case, to prevent overfitting, you could stop training after four epochs.
    - In general, you can use a range of techniques to mitigate overfitting, which will be covered in chapter 5
* Let’s train a new model from scratch for four epochs and then evaluate it on the test data

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

* + This fairly naïve approach achieves an accuracy of 88%. With state-of-the-art approaches, you should be able to get close to 95%

## Using a trained model to generate predictions on new data

* After having trained a model, you’ll want to use it in a practical setting.
  + You can generate the likelihood of reviews being positive by using the *predict* method, as you’ve learned in chapter 3

Graphical user interface

Description automatically generated with medium confidence

* + As you can see, the model is confident for some samples (0.99 or more, or 0.01 or less) but less confident for others (0.6, 0.4)

## Further experiments

* The following experiments will help convince you that the architecture choices you’ve made are all fairly reasonable, although there’s still room for improvement
  + You used two representation layers before the final classification layer. Try using one or three representation layers, and see how doing so affects validation and test accuracy
  + Try using layers with more units or fewer units: 32 units, 64 units, and so on
  + Try using the mse loss function instead of binary\_crossentropy.
  + Try using the tanh activation (an activation that was popular in the early days of neural networks) instead of relu.

## Wrapping up

* Here’s what you should take away from this example
  + You usually need to do quite a bit of preprocessing on your raw data in order to be able to feed it – as tensors – into a neural network
    - Sequences of words can be encoded as binary vectors, but there are other encoding options too.
  + Stacks of dense layers with relu activations can solve a wide range of problems (including sentiment classification), and you’ll likely use them frequently
  + In a binary classification problem, your model should end with a dense layer with one unit and a sigmoid activation: the output of your model should be a scalar between 0 and 1, encoding a probability
  + With such a scalar sigmoid output on a binary classification problem, the loss function you should use is binary\_crossentropy
  + The rmsprop optimizer is generally a good enough choice, whatever your problem. That’s one less thing for you to worry about.
  + As they get better on their training data, neural networks eventually start overfitting and end up obtaining increasingly worse results on data they’ve never seen before.
    - Be sure to always monitor performance on data that is outside of the training set.

# Multiclass classification example

* In the previous section, you saw how to classify vector inputs into mutually exclusive classes using a densely connected neural network. What happens when you have more than two classes?
  + In this section, we’ll build a model to classify Reuters newswires into 46 mutually exclusive topics.
  + Because we have many classes, this problem is an instance of multiclass classification, and because each data point should be classified into only one category, the problem is more specifically an instance of *single-label multiclass classification*.
    - If each data point could belong to multiple categories (in this case, topics), we’d be facing a *multilabel multiclass classification*.

## The Reuters dataset

* The *Reuters dataset* is a set of short newswires and their topics, published by Reuters in 1986.
  + It’s a simple, widely used toy dataset for text classification.
  + There are 46 different topics; some topics are more represented than others, but each topic has at least 10 examples in the training set.
* Like IMDB and MNIST, the Reuters dataset comes packaged as part of Keras.

Text

Description automatically generated

* + The argument num\_words = 10000 restricts the data to the 10,000 most frequently occurring words found in the data
  + You have 8,982 training examples and 2,246 test examples



* + As with the IMDB reviews, each example is a list of integers (word indices)



* + Here’s how you can decode it back to words

Text

Description automatically generated

* + The label associated with an example is an integer between 0 and 45 – a topic index:



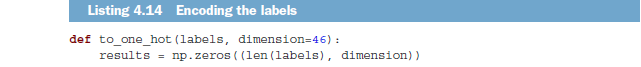
## Preparing the data

* Vectorize the data with the exact same code in the previous example

Graphical user interface, text, application

Description automatically generated

* + To vectorize the labels, there are two possibilities
    - Cast the label list as an integer tensor, or
    - Use *one-hot­ encoding*.
      * In this case, one-hot encoding of the labels consists of embedding each label as an all-zero vector with a 1 in the place of the label index.



A picture containing graphical user interface

Description automatically generated

* + We could also use the built-in way to do this in Keras



## Building your model

* This topic-classification problem looks similar to the previous movie-review classification problem: in both cases, we’re trying to classify short snippets of text. But there is a new constraint here: the number of output classes has gone from 2 to 46. The dimensionality of the output space is much larger
  + In a stack of dense layers, each layer can only access information present in the output of the previous layer.
    - If one layer drops some information relevant to the classification problem, this information can never be recovered by later layers: each layer can potentially become an information bottleneck.
    - In the previous example, we used 16-dimensional intermediate layers, but a 16-dimensiaonl space may be too limited to learn to separate 46 different classes: such small layers may act as information bottleneck, permanently dropping relevant information.
  + For this reason, we use larger layers, let’s use 64 units.

Text

Description automatically generated

* There are two other things you should note about this architecture