

# Text classification with movie reviews

Run in



Google (<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/ker>  
Colab

This notebook classifies movie reviews as *positive* or *negative* using the text of the review. This is an example of *binary*—or two-class—classification, an important and widely applicable kind of machine learning problem.

We'll use the [IMDB dataset](https://www.tensorflow.org/api_docs/python/tf/keras/datasets/imdb) ([https://www.tensorflow.org/api\\_docs/python/tf/keras/datasets/imdb](https://www.tensorflow.org/api_docs/python/tf/keras/datasets/imdb)) that contains the text of 50,000 movie reviews from the [Internet Movie Database](https://www.imdb.com/) (<https://www.imdb.com/>). These are split into 25,000 reviews for training and 25,000 reviews for testing. The training and testing sets are *balanced*, meaning they contain an equal number of positive and negative reviews.

This notebook uses [tf.keras](https://www.tensorflow.org/guide/keras) (<https://www.tensorflow.org/guide/keras>), a high-level API to build and train models in TensorFlow. For a more advanced text classification tutorial using [tf.keras](https://www.tensorflow.org/api_docs/python/tf/keras) ([https://www.tensorflow.org/api\\_docs/python/tf/keras](https://www.tensorflow.org/api_docs/python/tf/keras)), see the [MLCC Text Classification Guide](https://developers.google.com/machine-learning/guides/text-classification/) (<https://developers.google.com/machine-learning/guides/text-classification/>).

```
import tensorflow as tf
from tensorflow import keras
```



```
import numpy as np
```

```
print(tf.__version__)
```

```
1.12.0
```



## Download the IMDB dataset

The IMDB dataset comes packaged with TensorFlow. It has already been preprocessed such that the reviews (sequences of words) have been converted to sequences of integers, where

each integer represents a specific word in a dictionary.

The following code downloads the IMDB dataset to your machine (or uses a cached copy if you've already downloaded it):

```
imdb = keras.datasets.imdb

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
17465344/17464789 [=====] - 0s 0us/step
```

The argument `num_words=10000` keeps the top 10,000 most frequently occurring words in the training data. The rare words are discarded to keep the size of the data manageable.

## Explore the data

Let's take a moment to understand the format of the data. The dataset comes preprocessed: each example is an array of integers representing the words of the movie review. Each label is an integer value of either 0 or 1, where 0 is a negative review, and 1 is a positive review.

```
print("Training entries: {}, labels: {}".format(len(train_data), len(train_labels)))

Training entries: 25000, labels: 25000
```

The text of reviews have been converted to integers, where each integer represents a specific word in a dictionary. Here's what the first review looks like:

```
print(train_data[0])

[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, ...]
```

Movie reviews may be different lengths. The below code shows the number of words in the first and second reviews. Since inputs to a neural network must be the same length, we'll need to resolve this later.

```
len(train_data[0]) - len(train_data[1])
```

```
len(train_data[0]), len(train_data[1])
```

```
(218, 189)
```

## Convert the integers back to words

It may be useful to know how to convert integers back to text. Here, we'll create a helper function to query a dictionary object that contains the integer to string mapping:

```
# A dictionary mapping words to an integer index
word_index = imdb.get_word_index()

# The first indices are reserved
word_index = {k:(v+3) for k,v in word_index.items()}
word_index["<PAD>"] = 0
word_index["<START>"] = 1
word_index["<UNK>"] = 2 # unknown
word_index["<UNUSED>"] = 3

reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])

def decode_review(text):
    return ' '.join([reverse_word_index.get(i, '?') for i in text])
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
1646592/1641221 [=====] - 0s 0us/step
```

Now we can use the `decode_review` function to display the text for the first review:

```
decode_review(train_data[0])
```

```
" this film was just brilliant casting location scenery story direction every
```

## Prepare the data

The reviews—the arrays of integers—must be converted to tensors before fed into the neural network. This conversion can be done a couple of ways:

- One-hot-encode the arrays to convert them into vectors of 0s and 1s. For example, the sequence [3, 5] would become a 10,000-dimensional vector that is all zeros except for indices 3 and 5, which are ones. Then, make this the first layer in our network—a Dense layer—that can handle floating point vector data. This approach is memory intensive, though, requiring a `num_words * num_reviews` size matrix.
- Alternatively, we can pad the arrays so they all have the same length, then create an integer tensor of shape `max_length * num_reviews`. We can use an embedding layer capable of handling this shape as the first layer in our network.

In this tutorial, we will use the second approach.

Since the movie reviews must be the same length, we will use the `pad_sequences` ([https://www.tensorflow.org/api\\_docs/python/tf/keras/preprocessing/sequence/pad\\_sequences](https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/sequence/pad_sequences)) function to standardize the lengths:

```
train_data = keras.preprocessing.sequence.pad_sequences(train_data,
                                                         value=word_index["<PAD>"]
                                                         padding='post',
                                                         maxlen=256)

test_data = keras.preprocessing.sequence.pad_sequences(test_data,
                                                         value=word_index["<PAD>"]
                                                         padding='post',
                                                         maxlen=256)
```

Let's look at the length of the examples now:

```
len(train_data[0]), len(train_data[1])
```

```
(256, 256)
```

And inspect the (now padded) first review:

```
print(train_data[0])
```

```
[ 1  14  22  16  43 530 973 1622 1385  65 458 4468  66 3941
  4 173  36 256  5  25 100  43 838 112  50 670  2  9
 35 480 284  5 150  4 172 112 167  2 336 385 39  4
172 4536 1111 17 546 38 13 447  4 192  50 16  6 147
2025 19 14 22  4 1920 4613 469  4 22 71 87 12 16
```

```

43 530 38 76 15 13 1247 4 22 17 515 17 12 16
626 18 2 5 62 386 12 8 316 8 106 5 4 2223
5244 16 480 66 3785 33 4 130 12 16 38 619 5 25
124 51 36 135 48 25 1415 33 6 22 12 215 28 77
52 5 14 407 16 82 2 8 4 107 117 5952 15 256
4 2 7 3766 5 723 36 71 43 530 476 26 400 317
46 7 4 2 1029 13 104 88 4 381 15 297 98 32
2071 56 26 141 6 194 7486 18 4 226 22 21 134 476
26 480 5 144 30 5535 18 51 36 28 224 92 25 104
4 226 65 16 20 1224 88 12 16 202 5 16 1172 112

```

## Build the model

The neural network is created by stacking layers—this requires two main architectural decisions:

- How many layers to use in the model?
- How many *hidden units* to use for each layer?

In this example, the input data consists of an array of word-indices. The labels to predict are either 0 or 1. Let's build a model for this problem:

```
# input shape is the vocabulary count used for the movie reviews (10,000 words)
vocab_size = 10000
```

```
model = keras.Sequential()
model.add(keras.layers.Embedding(vocab_size, 16))
model.add(keras.layers.GlobalAveragePooling1D())
model.add(keras.layers.Dense(16, activation=tf.nn.relu))
model.add(keras.layers.Dense(1, activation=tf.nn.sigmoid))
```

```
model.summary()
```

```

-----
Layer (type)                 Output Shape              Param #
-----
embedding (Embedding)        (None, None, 16)         160000
-----
global_average_pooling1d (Gl (None, 16)                0
-----
dense (Dense)                (None, 16)               272
-----

```

```
dense_1 (Dense)                (None, 1)                17
=====
Total params: 160,289
Trainable params: 160,289
Non-trainable params: 0
```

The layers are stacked sequentially to build the classifier:

1. The first layer is an **Embedding** layer. This layer takes the integer-encoded vocabulary and looks up the embedding vector for each word-index. These vectors are learned as the model trains. The vectors add a dimension to the output array. The resulting dimensions are: **(batch, sequence, embedding)**.
2. Next, a **GlobalAveragePooling1D** layer returns a fixed-length output vector for each example by averaging over the sequence dimension. This allows the model to handle input of variable length, in the simplest way possible.
3. This fixed-length output vector is piped through a fully-connected (**Dense**) layer with 16 hidden units.
4. The last layer is densely connected with a single output node. Using the **sigmoid** activation function, this value is a float between 0 and 1, representing a probability, or confidence level.

## Hidden units

The above model has two intermediate or "hidden" layers, between the input and output. The number of outputs (units, nodes, or neurons) is the dimension of the representational space for the layer. In other words, the amount of freedom the network is allowed when learning an internal representation.

If a model has more hidden units (a higher-dimensional representation space), and/or more layers, then the network can learn more complex representations. However, it makes the network more computationally expensive and may lead to learning unwanted patterns—patterns that improve performance on training data but not on the test data. This is called *overfitting*, and we'll explore it later.

## Loss function and optimizer

A model needs a loss function and an optimizer for training. Since this is a binary classification problem and the model outputs a probability (a single-unit layer with a sigmoid activation), we'll use the `binary_crossentropy` loss function.

This isn't the only choice for a loss function, you could, for instance, choose `mean_squared_error`. But, generally, `binary_crossentropy` is better for dealing with probabilities—it measures the "distance" between probability distributions, or in our case, between the ground-truth distribution and the predictions.

Later, when we are exploring regression problems (say, to predict the price of a house), we will see how to use another loss function called mean squared error.

Now, configure the model to use an optimizer and a loss function:

```
model.compile(optimizer=tf.train.AdamOptimizer(),  
              loss='binary_crossentropy',  
              metrics=['accuracy'])
```



## Create a validation set

---

When training, we want to check the accuracy of the model on data it hasn't seen before. Create a *validation* set by setting apart 10,000 examples from the original training data. (Why not use the testing set now? Our goal is to develop and tune our model using only the training data, then use the test data just once to evaluate our accuracy).

```
x_val = train_data[:10000]  
partial_x_train = train_data[10000:]  
  
y_val = train_labels[:10000]  
partial_y_train = train_labels[10000:]
```



## Train the model

---

Train the model for 40 epochs in mini-batches of 512 samples. This is 40 iterations over all samples in the `x_train` and `y_train` tensors. While training, monitor the model's loss and accuracy on the 10,000 samples from the validation set:



```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=40,
                    batch_size=512,
                    validation_data=(x_val, y_val),
                    verbose=1)
```



Train on 15000 samples, validate on 10000 samples

Epoch 1/40

15000/15000 [=====] - 1s 54us/step - loss: 0.6923 - acc

Epoch 2/40

15000/15000 [=====] - 1s 39us/step - loss: 0.6879 - acc

Epoch 3/40

15000/15000 [=====] - 1s 41us/step - loss: 0.6781 - acc

Epoch 4/40

15000/15000 [=====] - 1s 42us/step - loss: 0.6599 - acc

Epoch 5/40

15000/15000 [=====] - 1s 39us/step - loss: 0.6311 - acc

Epoch 6/40

15000/15000 [=====] - 1s 41us/step - loss: 0.5928 - acc

Epoch 7/40

## Evaluate the model

And let's see how the model performs. Two values will be returned. Loss (a number which represents our error, lower values are better), and accuracy.



```
results = model.evaluate(test_data, test_labels)
```

```
print(results)
```



```
25000/25000 [=====] - 1s 35us/step
[0.32051958542823794, 0.87336]
```

This fairly naive approach achieves an accuracy of about 87%. With more advanced approaches, the model should get closer to 95%.

## Create a graph of accuracy and loss over time



`model.fit()` returns a `History` object that contains a dictionary with everything that happened during training:

```
history_dict = history.history
history_dict.keys()
```



```
dict_keys(['val_acc', 'acc', 'val_loss', 'loss'])
```



There are four entries: one for each monitored metric during training and validation. We can use these to plot the training and validation loss for comparison, as well as the training and validation accuracy:

```
import matplotlib.pyplot as plt
```



```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
```

```
epochs = range(1, len(acc) + 1)
```

```
# "bo" is for "blue dot"
plt.plot(epochs, loss, 'bo', label='Training loss')
# b is for "solid blue line"
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```

```
plt.show()
```



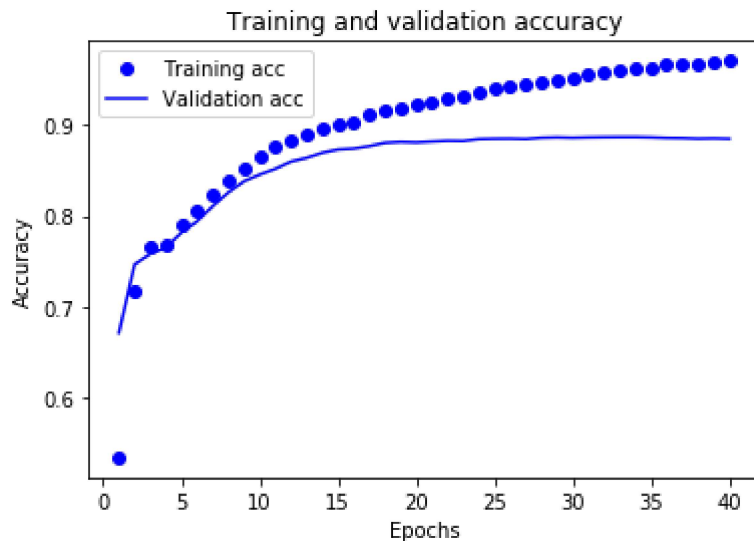
```
plt.clf() # clear figure
acc_values = history_dict['acc']
val_acc_values = history_dict['val_acc']
```



```
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
```

```
plt.legend()
```

```
plt.show()
```



In this plot, the dots represent the training loss and accuracy, and the solid lines are the validation loss and accuracy.

Notice the training loss *decreases* with each epoch and the training accuracy *increases* with each epoch. This is expected when using a gradient descent optimization—it should minimize the desired quantity on every iteration.

This isn't the case for the validation loss and accuracy—they seem to peak after about twenty epochs. This is an example of overfitting: the model performs better on the training data than it does on data it has never seen before. After this point, the model over-optimizes and learns representations *specific* to the training data that do not *generalize* to test data.

For this particular case, we could prevent overfitting by simply stopping the training after twenty or so epochs. Later, you'll see how to do this automatically with a callback.

```
#@title MIT License
#
# Copyright (c) 2017 François Chollet
#
# Permission is hereby granted, free of charge, to any person obtaining a
# copy of this software and associated documentation files (the "Software"),
# to deal in the Software without restriction, including without limitation
# the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the
```



```
# Software is furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
# THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
# FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
# DEALINGS IN THE SOFTWARE.
```

---

*Except as otherwise noted, the content of this page is licensed under the [Creative Commons Attribution 3.0 License](https://creativecommons.org/licenses/by/3.0/) (https://creativecommons.org/licenses/by/3.0/), and code samples are licensed under the [Apache 2.0 License](https://www.apache.org/licenses/LICENSE-2.0) (https://www.apache.org/licenses/LICENSE-2.0). For details, see our [Site Policies](https://developers.google.com/terms/site-policies) (https://developers.google.com/terms/site-policies). Java is a registered trademark of Oracle and/or its affiliates.*

*Last updated November 20, 2018.*