# Text classification with movie reviews

View on TensorFlow.org Run in Google Colab View source on GitHub

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 that contains the text of 50,000 movie reviews from the Internet Movie Database. 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, a high-level API to build and train models in TensorFlow. For a more advanced text classification tutorial using `tf.keras`, see the MLCC Text Classification Guide.

```
# keras.datasets.imdb is broken in 1.13 and 1.14, by np 1.16.3
!pip install tf_nightly
```

```
Collecting tf_nightly
  Downloading https://files.pythonhosted.org/packages/35/98/8017ea1b83e4552d858ab62
     |████████████████████████████████| 109.8MB 1.2MB/s
Requirement already satisfied: wheel>=0.26 in /usr/local/lib/python3.6/dist-package
Requirement already satisfied: numpy<2.0,>=1.14.5 in /usr/local/lib/python3.6/dist-
Requirement already satisfied: gast>=0.2.0 in /usr/local/lib/python3.6/dist-package
Requirement already satisfied: protobuf>=3.6.1 in /usr/local/lib/python3.6/dist-pac
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.6/dist-pa
Collecting wrapt>=1.11.1 (from tf_nightly)
  Downloading https://files.pythonhosted.org/packages/67/b2/0f71ca90b0ade7fad27e3d2
Requirement already satisfied: keras-applications>=1.0.6 in /usr/local/lib/python3.
Requirement already satisfied: absl-py>=0.7.0 in /usr/local/lib/python3.6/dist-pack
Collecting tb-nightly<1.15.0a0,>=1.14.0a0 (from tf_nightly)
```

```python
from __future__ import absolute_import, division, print_function, unicode_literals

import tensorflow as tf
from tensorflow import keras

import numpy as np

print(tf.__version__)
```

```
1.14.1-dev20190604
```

## 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):

```python
imdb = keras.datasets.imdb

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/i
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.

```python
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, 256,
```

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])
```

```
(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-datasets/i
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])
```

```
"<START> this film was just brilliant casting location scenery story direction ever
```

# 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:

- Convert the arrays into vectors of 0s and 1s indicating word occurrence, similar to a one-hot encoding. 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] 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   38 1334   88   12   16  283    5   16 4472  113
  103   32   15   16 5345   19  178   32    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0]
```

## 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

```
# 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()
```

```
WARNING: Logging before flag parsing goes to stderr.
W0604 19:39:10.412949 140560623351680 deprecation.py:506] From /usr/local/lib/pytho
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the cons
Model: "sequential"
_________________________________________________________________
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='adam',
              loss='binary_crossentropy',
              metrics=['acc'])
```

```
W0604 19:39:14.968133 140560623351680 deprecation.py:323] From /usr/local/lib/pytho
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
```

## 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 71us/sample - loss: 0.6921 - acc:
Epoch 2/40
15000/15000 [==============================] - 1s 59us/sample - loss: 0.6878 - acc:
Epoch 3/40
15000/15000 [==============================] - 1s 62us/sample - loss: 0.6785 - acc:
Epoch 4/40
15000/15000 [==============================] - 1s 62us/sample - loss: 0.6615 - acc:
Epoch 5/40
15000/15000 [==============================] - 1s 61us/sample - loss: 0.6347 - acc:
Epoch 6/40
15000/15000 [==============================] - 1s 60us/sample - loss: 0.5990 - acc:
Epoch 7/40
15000/15000 [==============================] - 1s 59us/sample - loss: 0.5563 - acc:
Epoch 8/40
15000/15000 [==============================] - 1s 60us/sample - loss: 0.5105 - acc:
Epoch 9/40
15000/15000 [==============================] - 1s 61us/sample - loss: 0.4658 - acc:
Epoch 10/40
15000/15000 [==============================] - 1s 59us/sample - loss: 0.4248 - acc:
Epoch 11/40
15000/15000 [==============================] - 1s 59us/sample - loss: 0.3894 - acc:
Epoch 12/40
15000/15000 [==============================] - 1s 60us/sample - loss: 0.3591 - acc:
Epoch 13/40
15000/15000 [==============================] - 1s 61us/sample - loss: 0.3344 - acc:
Epoch 14/40
15000/15000 [==============================] - 1s 61us/sample - loss: 0.3123 - acc:
Epoch 15/40
15000/15000 [==============================] - 1s 62us/sample - loss: 0.2940 - acc:
Epoch 16/40
15000/15000 [==============================] - 1s 61us/sample - loss: 0.2777 - acc:
Epoch 17/40
15000/15000 [==============================] - 1s 62us/sample - loss: 0.2629 - acc:
Epoch 18/40
15000/15000 [==============================] - 1s 61us/sample - loss: 0.2498 - acc:
Epoch 19/40
15000/15000 [==============================] - 1s 61us/sample - loss: 0.2380 - acc:
Epoch 20/40
15000/15000 [==============================] - 1s 60us/sample - loss: 0.2274 - acc:
Epoch 21/40
15000/15000 [==============================] - 1s 61us/sample - loss: 0.2168 - acc:
Epoch 22/40
15000/15000 [==============================] - 1s 61us/sample - loss: 0.2078 - acc:
Epoch 23/40
15000/15000 [==============================] - 1s 60us/sample - loss: 0.1988 - acc:
Epoch 24/40
15000/15000 [==============================] - 1s 59us/sample - loss: 0.1909 - acc:
Epoch 25/40
15000/15000 [==============================] - 1s 60us/sample - loss: 0.1829 - acc:
Epoch 26/40
15000/15000 [==============================] - 1s 61us/sample - loss: 0.1758 - acc:
Epoch 27/40
15000/15000 [==============================] - 1s 60us/sample - loss: 0.1690 - acc:
Epoch 28/40
15000/15000 [==============================] - 1s 60us/sample - loss: 0.1625 - acc:
Epoch 29/40
15000/15000 [==============================] - 1s 60us/sample - loss: 0.1569 - acc:
Epoch 30/40
15000/15000 [==============================] - 1s 61us/sample - loss: 0.1510 - acc:
```

```
Epoch 31/40
15000/15000 [==============================] - 1s 61us/sample - loss: 0.1450 - acc:
Epoch 32/40
15000/15000 [==============================] - 1s 61us/sample - loss: 0.1395 - acc:
Epoch 33/40
15000/15000 [==============================] - 1s 60us/sample - loss: 0.1343 - acc:
Epoch 34/40
15000/15000 [==============================] - 1s 60us/sample - loss: 0.1296 - acc:
Epoch 35/40
15000/15000 [==============================] - 1s 60us/sample - loss: 0.1254 - acc:
Epoch 36/40
15000/15000 [==============================] - 1s 61us/sample - loss: 0.1206 - acc:
Epoch 37/40
15000/15000 [==============================] - 1s 62us/sample - loss: 0.1161 - acc:
Epoch 38/40
15000/15000 [==============================] - 1s 60us/sample - loss: 0.1119 - acc:
Epoch 39/40
15000/15000 [==============================] - 1s 61us/sample - loss: 0.1086 - acc:
Epoch 40/40
15000/15000 [==============================] - 1s 61us/sample - loss: 0.1044 - acc:
```

## 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 38us/sample - loss: 0.3225 - acc:
[0.32245946591377256, 0.87288]
```

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:

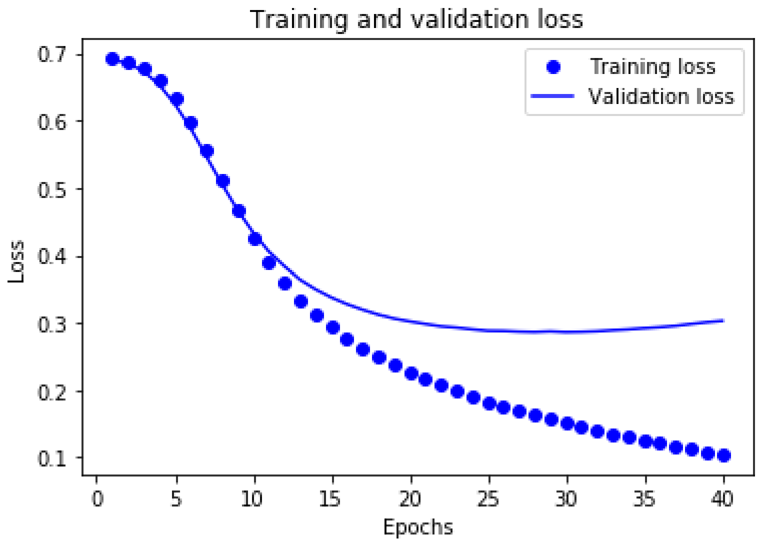
```
history_dict = history.history
history_dict.keys()
```

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

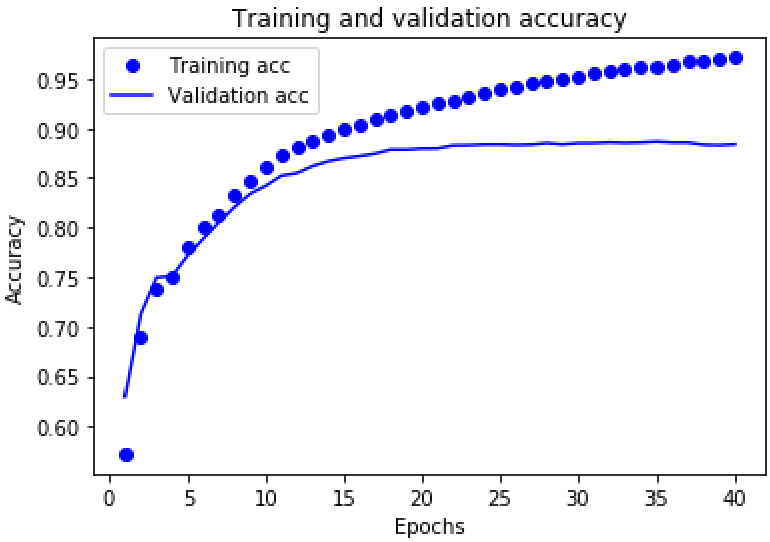
There are four entries: one for each monitored metric during training and validation. We can use these to plot the training and validation loss for comparison, as well as the training and validation accuracy:

```
import matplotlib.pyplot as plt

acc = history_dict['acc']
val_acc = history_dict['val_acc']
loss = history_dict['loss']
val_loss = history_dict['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

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.