Basic text classification

Run in
Google (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/keras
Colab

This tutorial demonstrates text classification starting from plain text files stored on disk. You'll train a binary classifier to perform sentiment analysis on an IMDB dataset. At the end of the notebook, there is an exercise for you to try, in which you'll train a multi-class classifier to predict the tag for a programming question on Stack Overflow.

```
import matplotlib.pyplot as plt
import os
import re
import shutil
import string
import tensorflow as tf

from tensorflow.keras import layers
from tensorflow.keras import losses
```

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

```
2.16.1
```

Sentiment analysis

This notebook trains a sentiment analysis model to classify movie reviews as *positive* or *negative*, based on 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.

You'll use the Large Movie Review Dataset (https://ai.stanford.edu/%7Eamaas/data/sentiment/) that contains the text of 50,000 movie reviews from the Internet Movie Database (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.

Download and explore the IMDB dataset

Let's download and extract the dataset, then explore the directory structure.

```
url = "https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz"

dataset = tf.keras.utils.get_file("aclImdb_v1", url,
                                    untar=True, cache_dir='.',
                                    cache_subdir='')

dataset_dir = os.path.join(os.path.dirname(dataset), 'aclImdb')
```

```
Downloading data from https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar
84125825/84125825 ━━━━━━━━━━━━━━━━━━━━ 5s 0us/step
```

```
os.listdir(dataset_dir)
```

```
['README', 'test', 'imdb.vocab', 'train', 'imdbEr.txt']
```

```
train_dir = os.path.join(dataset_dir, 'train')
os.listdir(train_dir)
```

```
['unsup',
 'urls_neg.txt',
 'labeledBow.feat',
 'unsupBow.feat',
 'neg',
 'urls_unsup.txt',
 'urls_pos.txt',
 'pos']
```

The **`aclImdb/train/pos`** and **`aclImdb/train/neg`** directories contain many text files, each of which is a single movie review. Let's take a look at one of them.

```
sample_file = os.path.join(train_dir, 'pos/1181_9.txt')
with open(sample_file) as f:
  print(f.read())
```

```
Rachel Griffiths writes and directs this award winning short film. A heartwarming
```

Load the dataset

Next, you will load the data off disk and prepare it into a format suitable for training. To do so, you will use the helpful text_dataset_from_directory (https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text_dataset_from_directory) utility, which expects a directory structure as follows.

```
main_directory/
...class_a/
......a_text_1.txt
......a_text_2.txt
...class_b/
......b_text_1.txt
......b_text_2.txt
```

To prepare a dataset for binary classification, you will need two folders on disk, corresponding to **`class_a`** and **`class_b`**. These will be the positive and negative movie reviews, which can be found in **`aclImdb/train/pos`** and **`aclImdb/train/neg`**. As the IMDB dataset contains additional folders, you will remove them before using this utility.

```
remove_dir = os.path.join(train_dir, 'unsup')
shutil.rmtree(remove_dir)
```

Next, you will use the **`text_dataset_from_directory`** utility to create a labeled **`tf.data.Dataset`** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset). tf.data (https://www.tensorflow.org/guide/data) is a powerful collection of tools for working with data.

When running a machine learning experiment, it is a best practice to divide your dataset into three splits: train (https://developers.google.com/machine-learning/glossary#training_set), validation (https://developers.google.com/machine-learning/glossary#validation_set), and test (https://developers.google.com/machine-learning/glossary#test-set).

The IMDB dataset has already been divided into train and test, but it lacks a validation set. Let's create a validation set using an 80:20 split of the training data by using the **`validation_split`** argument below.

```
batch_size = 32
seed = 42

raw_train_ds = tf.keras.utils.text_dataset_from_directory(
    'aclImdb/train',
    batch_size=batch_size,
    validation_split=0.2,
    subset='training',
    seed=seed)
```

```
Found 25000 files belonging to 2 classes.
Using 20000 files for training.
```

As you can see above, there are 25,000 examples in the training folder, of which you will use 80% (or 20,000) for training. As you will see in a moment, you can train a model by passing a dataset directly to **`model.fit`**. If you're new to **`tf.data`** (https://www.tensorflow.org/api_docs/python/tf/data), you can also iterate over the dataset and print out a few examples as follows.

```
for text_batch, label_batch in raw_train_ds.take(1):
  for i in range(3):
    print("Review", text_batch.numpy()[i])
    print("Label", label_batch.numpy()[i])
```

```
Review b'"Pandemonium" is a horror movie spoof that comes off more stupid than fun
Label 0
Review b"David Mamet is a very interesting and a very un-equal director. His first
Label 0
Review b'Great documentary about the lives of NY firefighters during the worst ter
Label 1
2024-03-13 01:21:51.005004: W tensorflow/core/framework/local_rendezvous.cc:404] L
```

Notice the reviews contain raw text (with punctuation and occasional HTML tags like **`<br/>`**). You will show how to handle these in the following section.

The labels are 0 or 1. To see which of these correspond to positive and negative movie reviews, you can check the **`class_names`** property on the dataset.

```
print("Label 0 corresponds to", raw_train_ds.class_names[0])
print("Label 1 corresponds to", raw_train_ds.class_names[1])
```

```
Label 0 corresponds to neg
Label 1 corresponds to pos
```

Next, you will create a validation and test dataset. You will use the remaining 5,000 reviews from the training set for validation.

Note: When using the **`validation_split`** and **`subset`** arguments, make sure to either specify a random seed, or to pass **`shuffle=False`**, so that the validation and training splits have no overlap.

```
raw_val_ds = tf.keras.utils.text_dataset_from_directory(
    'aclImdb/train',
    batch_size=batch_size,
    validation_split=0.2,
    subset='validation',
    seed=seed)
```

```
Found 25000 files belonging to 2 classes.
Using 5000 files for validation.
```

```
raw_test_ds = tf.keras.utils.text_dataset_from_directory(
    'aclImdb/test',
    batch_size=batch_size)
```

```
Found 25000 files belonging to 2 classes.
```

Prepare the dataset for training

Next, you will standardize, tokenize, and vectorize the data using the helpful **`tf.keras.layers.TextVectorization`** (https://www.tensorflow.org/api_docs/python/tf/keras/layers/TextVectorization) layer.

Standardization refers to preprocessing the text, typically to remove punctuation or HTML elements to simplify the dataset. Tokenization refers to splitting strings into tokens (for example, splitting a sentence into individual words, by splitting on whitespace). Vectorization refers to converting tokens into numbers so they can be fed into a neural network. All of these tasks can be accomplished with this layer.

As you saw above, the reviews contain various HTML tags like `<br />`. These tags will not be removed by the default standardizer in the `TextVectorization` layer (which converts text to lowercase and strips punctuation by default, but doesn't strip HTML). You will write a custom standardization function to remove the HTML.

Note: To prevent training-testing skew (https://developers.google.com/machine-learning/guides/rules-of-ml#training-serving_skew) (also known as training-serving skew), it is important to preprocess the data identically at train and test time. To facilitate this, the `TextVectorization` layer can be included directly inside your model, as shown later in this tutorial.

```
def custom_standardization(input_data):
  lowercase = tf.strings.lower(input_data)
  stripped_html = tf.strings.regex_replace(lowercase, '<br />', ' ')
  return tf.strings.regex_replace(stripped_html,
                                  '[%s]' % re.escape(string.punctuation),
                                  '')
```

Next, you will create a `TextVectorization` layer. You will use this layer to standardize, tokenize, and vectorize our data. You set the `output_mode` to `int` to create unique integer indices for each token.

Note that you're using the default split function, and the custom standardization function you defined above. You'll also define some constants for the model, like an explicit maximum `sequence_length`, which will cause the layer to pad or truncate sequences to exactly `sequence_length` values.

```
max_features = 10000
sequence_length = 250

vectorize_layer = layers.TextVectorization(
    standardize=custom_standardization,
    max_tokens=max_features,
    output_mode='int',
    output_sequence_length=sequence_length)
```

Next, you will call `adapt` to fit the state of the preprocessing layer to the dataset. This will cause the model to build an index of strings to integers.

Note: It's important to only use your training data when calling adapt (using the test set would leak information).

```
# Make a text-only dataset (without labels), then call adapt
train_text = raw_train_ds.map(lambda x, y: x)
vectorize_layer.adapt(train_text)
```

```
2024-03-13 01:21:58.431096: W tensorflow/core/framework/local_rendezvous.cc:404] L
```

Let's create a function to see the result of using this layer to preprocess some data.

```
def vectorize_text(text, label):
  text = tf.expand_dims(text, -1)
  return vectorize_layer(text), label
```

```
# retrieve a batch (of 32 reviews and labels) from the dataset
text_batch, label_batch = next(iter(raw_train_ds))
first_review, first_label = text_batch[0], label_batch[0]
print("Review", first_review)
print("Label", raw_train_ds.class_names[first_label])
print("Vectorized review", vectorize_text(first_review, first_label))
```

```
Review tf.Tensor(b'Silent Night, Deadly Night 5 is the very last of the series, a
Label neg
Vectorized review (<tf.Tensor: shape=(1, 250), dtype=int64, numpy=
array([[1287,  313, 2380,  313,  661,    7,    2,   52,  229,    5,    2,
         200,    3,   38,  170,  669,   29, 5492,    6,    2,   83,  297,
         549,   32,  410,    3,    2,  186,   12,   29,    4,    1,  191,
         510,  549,    6,    2, 8229,  212,   46,  576,  175,  168,   20,
           1, 5361,  290,    4,    1,  761,  969,    1,    3,   24,  935,
        2271,  393,    7,    1, 1675,    4, 3747,  250,  148,    4,  112,
         436,  761, 3529,  548,    4, 3633,   31,    2, 1331,   28, 2096,
           3, 2912,    9,    6,  163,    4, 1006,   20,    2,    1,   15,
          85,   53,  147,    9,  292,   89,  959, 2314,  984,   27,  762,
           6,  959,    9,  564,   18,    7, 2140,   32,   24, 1254,   36,
```

As you can see above, each token has been replaced by an integer. You can lookup the token (string) that each integer corresponds to by calling `.get_vocabulary()` on the layer.

```
print("1287 ---> ",vectorize_layer.get_vocabulary()[1287])
print(" 313 ---> ",vectorize_layer.get_vocabulary()[313])
print('Vocabulary size: {}'.format(len(vectorize_layer.get_vocabulary())))
```

```
1287 --->  silent
 313 --->  night
Vocabulary size: 10000
```

You are nearly ready to train your model. As a final preprocessing step, you will apply the TextVectorization layer you created earlier to the train, validation, and test dataset.

```
train_ds = raw_train_ds.map(vectorize_text)
val_ds = raw_val_ds.map(vectorize_text)
test_ds = raw_test_ds.map(vectorize_text)
```

Configure the dataset for performance

These are two important methods you should use when loading data to make sure that I/O does not become blocking.

`.cache()` keeps data in memory after it's loaded off disk. This will ensure the dataset does not become a bottleneck while training your model. If your dataset is too large to fit into memory, you can also use this method to create a performant on-disk cache, which is more efficient to read than many small files.

`.prefetch()` overlaps data preprocessing and model execution while training.

You can learn more about both methods, as well as how to cache data to disk in the data performance guide (https://www.tensorflow.org/guide/data_performance).

```
AUTOTUNE = tf.data.AUTOTUNE

train_ds = train_ds.cache().prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
test_ds = test_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

Create the model

It's time to create your neural network:

```
embedding_dim = 16
```

```
model = tf.keras.Sequential([
  layers.Embedding(max_features, embedding_dim),
  layers.Dropout(0.2),
  layers.GlobalAveragePooling1D(),
  layers.Dropout(0.2),
  layers.Dense(1, activation='sigmoid')])

model.summary()
```

```
del: "sequential"
```

```
Layer (type)                      ┃ Output Shape           ┃      Param # ┃
embedding (Embedding)             │ ?                      │  0 (unbuilt) │
dropout (Dropout)                 │ ?                      │            0 │
global_average_pooling1d          │ ?                      │  0 (unbuilt) │
(GlobalAveragePooling1D)          │                        │              │
dropout_1 (Dropout)               │ ?                      │            0 │
dense (Dense)                     │ ?                      │  0 (unbuilt) │
```

```
otal params: 0 (0.00 B)
```

```
rainable params: 0 (0.00 B)
```

```
on-trainable params: 0 (0.00 B)
```

The layers are stacked sequentially to build the classifier:

1. The first layer is an **`Embedding`** layer. This layer takes the integer-encoded reviews and looks up an 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)`**. To learn more about embeddings, check out the Word embeddings (https://www.tensorflow.org/text/guide/word_embeddings) tutorial.
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. The last layer is densely connected with a single output node.

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), you'll use **`losses.BinaryCrossentropy`** (https://www.tensorflow.org/api_docs/python/tf/keras/losses/BinaryCrossentropy) loss function.

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

```
model.compile(loss=losses.BinaryCrossentropy(),
              optimizer='adam',
              metrics=[tf.metrics.BinaryAccuracy(threshold=0.5)])
```

Train the model

You will train the model by passing the `dataset` object to the fit method.

```
epochs = 10
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=epochs)
```

```
Epoch 1/10
WARNING: All log messages before absl::InitializeLog() is called are written to S
I0000 00:00:1710292919.947515   10027 service.cc:145] XLA service 0xc8f5ec0 initi
I0000 00:00:1710292919.947561   10027 service.cc:153]   StreamExecutor device (0)
I0000 00:00:1710292919.947565   10027 service.cc:153]   StreamExecutor device (1)
I0000 00:00:1710292919.947568   10027 service.cc:153]   StreamExecutor device (2)
I0000 00:00:1710292919.947571   10027 service.cc:153]   StreamExecutor device (3)
84/625 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - binary_accuracy: 0.5238 - loss: 0.6921
I0000 00:00:1710292922.503775   10027 device_compiler.h:188] Compiled cluster usi
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - binary_accuracy: 0.5831 - loss: 0.6809 -
```

```
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - binary_accuracy: 0.7595 - loss: 0.5778 -
```

Evaluate the model

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.

```
loss, accuracy = model.evaluate(test_ds)

print("Loss: ", loss)
print("Accuracy: ", accuracy)
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - binary_accuracy: 0.8571 - loss: 0.3293
Loss:  0.3322346806526184
Accuracy:  0.8547599911689758
```

This fairly naive approach achieves an accuracy of about 86%.

Create a plot of accuracy and loss over time

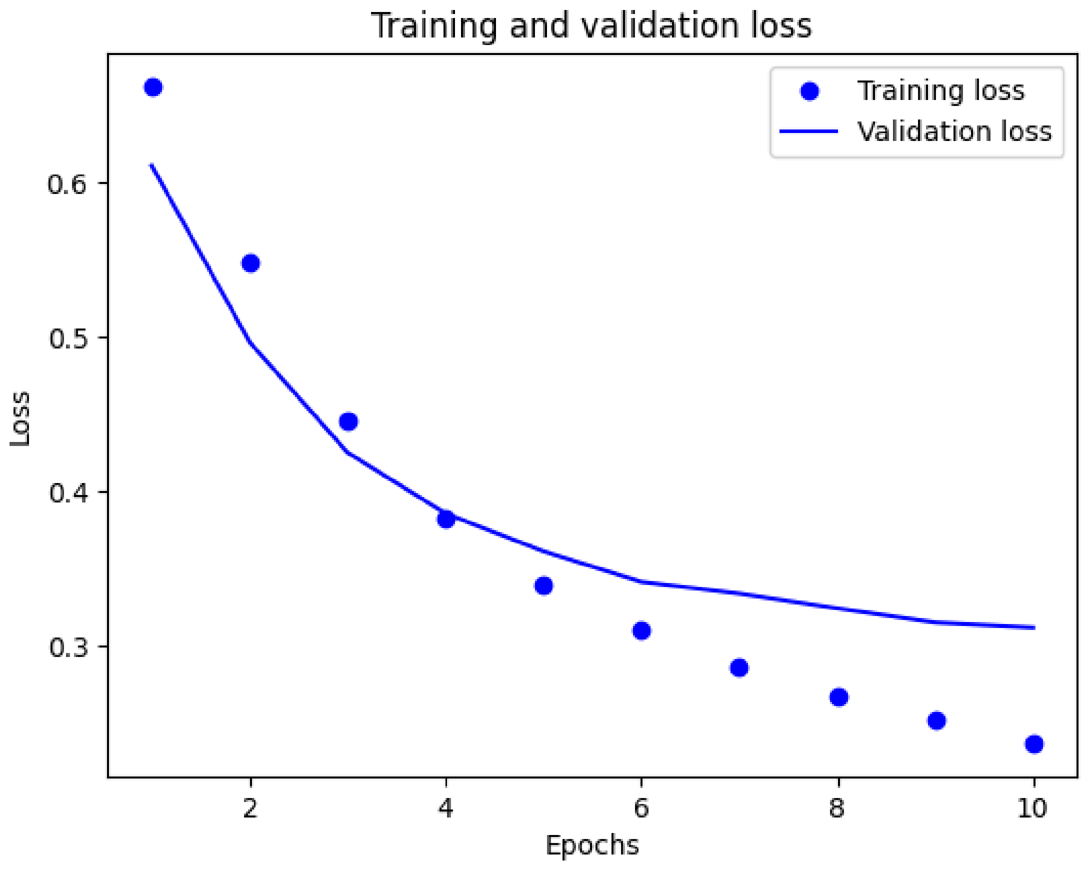
`model.fit()` returns a **`History`** object that contains a dictionary with everything that happened during training:

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

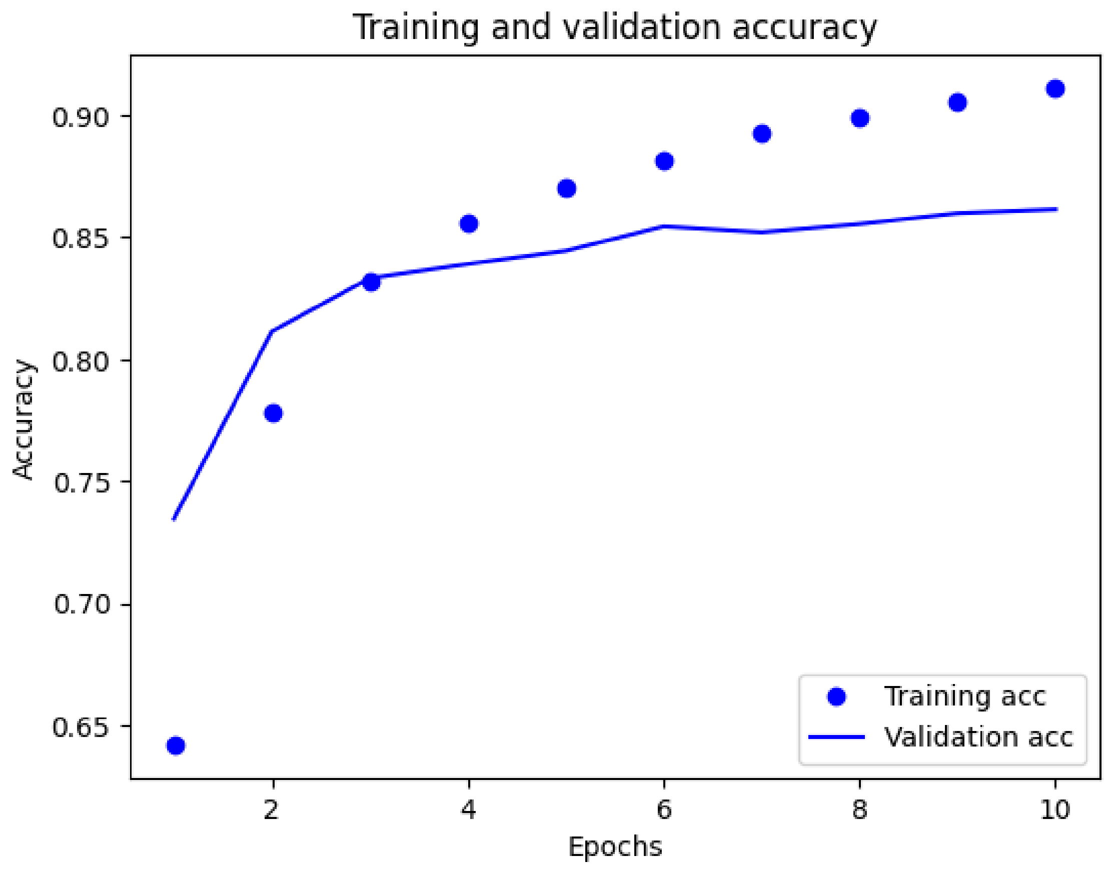
```
dict_keys(['binary_accuracy', 'loss', 'val_binary_accuracy', 'val_loss'])
```

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

```
acc = history_dict['binary_accuracy']
val_acc = history_dict['val_binary_accuracy']
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.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(loc='lower right')

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 before the training accuracy. 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, you could prevent overfitting by simply stopping the training when the validation accuracy is no longer increasing. One way to do so is to use the **`tf.keras.callbacks.EarlyStopping`** (https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping) callback.

Export the model

In the code above, you applied the `TextVectorization` layer to the dataset before feeding text to the model. If you want to make your model capable of processing raw strings (for example, to simplify deploying it), you can include the `TextVectorization` layer inside your model. To do so, you can create a new model using the weights you just trained.

```
export_model = tf.keras.Sequential([
  vectorize_layer,
  model,
  layers.Activation('sigmoid')
])

export_model.compile(
    loss=losses.BinaryCrossentropy(from_logits=False), optimizer="adam", metrics=[
)

# Test it with `raw_test_ds`, which yields raw strings
loss, accuracy = export_model.evaluate(raw_test_ds)
print(accuracy)
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - accuracy: 0.5000 - loss: 0.5876
0.5000399947166443
```

Inference on new data

To get predictions for new examples, you can simply call `model.predict()`.

```
examples = tf.constant([
  "The movie was great!",
  "The movie was okay.",
  "The movie was terrible..."
])

export_model.predict(examples)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 159ms/step
array([[0.58198076],
       [0.54727507],
       [0.5339556 ]], dtype=float32)
```

Including the text preprocessing logic inside your model enables you to export a model for production that simplifies deployment, and reduces the potential for train/test skew (https://developers.google.com/machine-learning/guides/rules-of-ml#training-serving_skew).

There is a performance difference to keep in mind when choosing where to apply your TextVectorization layer. Using it outside of your model enables you to do asynchronous CPU processing and buffering of your data when training on GPU. So, if you're training your model on the GPU, you probably want to go with this option to get the best performance while developing your model, then switch to including the TextVectorization layer inside your model when you're ready to prepare for deployment.

Visit this tutorial (https://www.tensorflow.org/tutorials/keras/save_and_load) to learn more about saving models.

Exercise: multi-class classification on Stack Overflow questions

This tutorial showed how to train a binary classifier from scratch on the IMDB dataset. As an exercise, you can modify this notebook to train a multi-class classifier to predict the tag of a programming question on Stack Overflow (http://stackoverflow.com/).

A dataset (https://storage.googleapis.com/download.tensorflow.org/data/stack_overflow_16k.tar.gz) has been prepared for you to use containing the body of several thousand programming questions (for example, "How can I sort a dictionary by value in Python?") posted to Stack Overflow. Each of these is labeled with exactly one tag (either Python, CSharp, JavaScript, or Java). Your task is to take a question as input, and predict the appropriate tag, in this case, Python.

The dataset you will work with contains several thousand questions extracted from the much larger public Stack Overflow dataset on BigQuery (https://console.cloud.google.com/marketplace/details/stack-exchange/stack-overflow), which contains more than 17 million posts.

After downloading the dataset, you will find it has a similar directory structure to the IMDB dataset you worked with previously:

```
train/
...python/
......0.txt
......1.txt
...javascript/
......0.txt
......1.txt
...csharp/
......0.txt
......1.txt
...java/
......0.txt
......1.txt
```

Note: To increase the difficulty of the classification problem, occurrences of the words Python, CSharp, JavaScript, or Java in the programming questions have been replaced with the word *blank* (as many questions contain the language they're about).

To complete this exercise, you should modify this notebook to work with the Stack Overflow dataset by making the following modifications:

1. At the top of your notebook, update the code that downloads the IMDB dataset with code to download the Stack Overflow dataset (https://storage.googleapis.com/download.tensorflow.org/data/stack_overflow_16k.tar.gz) that has already been prepared. As the Stack Overflow dataset has a similar directory structure, you will not need to make many modifications.
2. Modify the last layer of your model to `Dense(4)`, as there are now four output classes.

3. When compiling the model, change the loss to `tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)` (https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy). This is the correct loss function to use for a multi-class classification problem, when the labels for each class are integers (in this case, they can be 0, *1*, *2*, or *3*). In addition, change the metrics to `metrics=['accuracy']`, since this is a multi-class classification problem (`tf.metrics.BinaryAccuracy` is only used for binary classifiers).
4. When plotting accuracy over time, change `binary_accuracy` and `val_binary_accuracy` to `accuracy` and `val_accuracy`, respectively.
5. Once these changes are complete, you will be able to train a multi-class classifier.

Learning more

This tutorial introduced text classification from scratch. To learn more about the text classification workflow in general, check out the Text classification guide (https://developers.google.com/machine-learning/guides/text-classification/) from Google Developers.

Last updated 2024-03-23 UTC.