Text classification with an RNN

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

This text classification tutorial trains a recurrent neural network on the IMDB large movie review dataset for sentiment analysis.

Setup

```
import numpy as np

import tensorflow_datasets as tfds
import tensorflow as tf

tfds.disable_progress_bar()
```

Import `matplotlib` and create a helper function to plot graphs:

```
import matplotlib.pyplot as plt


def plot_graphs(history, metric):
  plt.plot(history.history[metric])
  plt.plot(history.history['val_'+metric], '')
  plt.xlabel("Epochs")
  plt.ylabel(metric)
  plt.legend([metric, 'val_'+metric])
```

Setup input pipeline

The IMDB large movie review dataset is a *binary classification* dataset—all the reviews have either a *positive* or *negative* sentiment.

Download the dataset using TFDS. See the loading text tutorial for details on how to load this sort of data manually.

```
dataset, info = tfds.load('imdb_reviews', with_info=True,
                          as_supervised=True)
train_dataset, test_dataset = dataset['train'], dataset['test']

train_dataset.element_spec
```

```
Downloading and preparing dataset Unknown size (download: Unknown size, generated: Unknown size, total: Unknown size) to /ro
Dataset imdb_reviews downloaded and prepared to /root/tensorflow_datasets/imdb_reviews/plain_text/1.0.0. Subsequent calls wi
(TensorSpec(shape=(), dtype=tf.string, name=None),
 TensorSpec(shape=(), dtype=tf.int64, name=None))
```

Initially this returns a dataset of (text, label pairs):

```
for example, label in train_dataset.take(1):
  print('text: ', example.numpy())
  print('label: ', label.numpy())
```

```
text:  b"This was an absolutely terrible movie. Don't be lured in by Christopher Walken or Michael Ironside. Both are great
label:  0
```

Next shuffle the data for training and create batches of these `(text, label)` pairs:

```
BUFFER_SIZE = 10000
BATCH_SIZE = 64
```

```
train_dataset = train_dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
test_dataset = test_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
```

```
for example, label in train_dataset.take(1):
  print('texts: ', example.numpy()[:3])
  print()
  print('labels: ', label.numpy()[:3])
```

```
texts:  [b'This movie is just not worth your time. Its reliance upon New-Age mysticism serves as its only semi-interesting d
 b'Dieter Bohlen, Germany\'s notorious composer and producer of slightly trashy pop hits like "You\'re my heart, you\'re my
 b'Working-class romantic drama from director Martin Ritt is as unbelievable as they come, yet there are moments of pleasure

labels:  [0 0 1]
```

Create the text encoder

The raw text loaded by `tfds` needs to be processed before it can be used in a model. The simplest way to process text for training is using the `TextVectorization` layer. This layer has many capabilities, but this tutorial sticks to the default behavior.

Create the layer, and pass the dataset's text to the layer's `.adapt` method:

```
VOCAB_SIZE = 1000
encoder = tf.keras.layers.TextVectorization(
    max_tokens=VOCAB_SIZE)
encoder.adapt(train_dataset.map(lambda text, label: text))
```

```
WARNING:tensorflow:From /usr/local/lib/python3.9/dist-packages/tensorflow/python/autograph/pyct/static_analysis/liveness.py:
Instructions for updating:
Lambda fuctions will be no more assumed to be used in the statement where they are used, or at least in the same block. http
```

The `.adapt` method sets the layer's vocabulary. Here are the first 20 tokens. After the padding and unknown tokens they're sorted by frequency:

```
vocab = np.array(encoder.get_vocabulary())
vocab[:20]
```

```
array(['', '[UNK]', 'the', 'and', 'a', 'of', 'to', 'is', 'in', 'it', 'i',
       'this', 'that', 'br', 'was', 'as', 'for', 'with', 'movie', 'but'],
      dtype='<U14')
```

Once the vocabulary is set, the layer can encode text into indices. The tensors of indices are 0-padded to the longest sequence in the batch (unless you set a fixed `output_sequence_length`):

```
encoded_example = encoder(example)[:3].numpy()
encoded_example
```

```
array([[ 11,  18,   7, ...,   0,   0,   0],
       [  1,   1,   1, ...,   0,   0,   0],
       [  1, 718, 477, ...,   0,   0,   0]])
```

With the default settings, the process is not completely reversible. There are three main reasons for that:

1. The default value for `preprocessing.TextVectorization`'s `standardize` argument is `"lower_and_strip_punctuation"`.
2. The limited vocabulary size and lack of character-based fallback results in some unknown tokens.

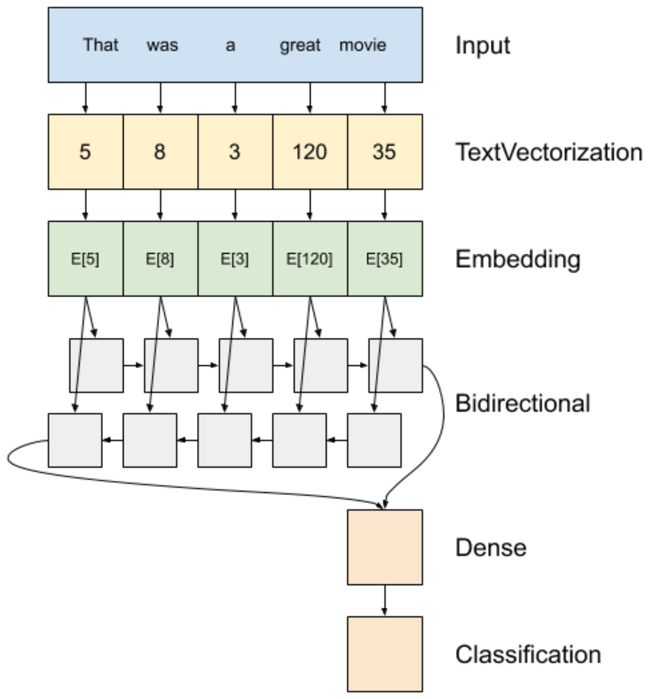
```
for n in range(3):
  print("Original: ", example[n].numpy())
  print("Round-trip: ", " ".join(vocab[encoded_example[n]]))
  print()
```

```
Original:  b'This movie is just not worth your time. Its reliance upon New-Age mysticism serves as its only semi-interesting
Round-trip:  this movie is just not worth your time its [UNK] upon [UNK] [UNK] [UNK] as its only [UNK] [UNK] the plot is one

Original:  b'Dieter Bohlen, Germany\'s notorious composer and producer of slightly trashy pop hits like "You\'re my heart, y
Round-trip:  [UNK] [UNK] [UNK] [UNK] [UNK] and [UNK] of [UNK] [UNK] [UNK] [UNK] like youre my heart youre my [UNK] felt the

Original:  b'Working-class romantic drama from director Martin Ritt is as unbelievable as they come, yet there are moments o
Round-trip:  [UNK] romantic drama from director [UNK] [UNK] is as [UNK] as they come yet there are moments of [UNK] due most
```

Create the model



Above is a diagram of the model.

1. This model can be build as a `tf.keras.Sequential`.
2. The first layer is the `encoder`, which converts the text to a sequence of token indices.
3. After the encoder is an embedding layer. An embedding layer stores one vector per word. When called, it converts the sequences of word indices to sequences of vectors. These vectors are trainable. After training (on enough data), words with similar meanings often have similar vectors.

   This index-lookup is much more efficient than the equivalent operation of passing a one-hot encoded vector through a `tf.keras.layers.Dense` layer.
4. A recurrent neural network (RNN) processes sequence input by iterating through the elements. RNNs pass the outputs from one timestep to their input on the next timestep.

   The `tf.keras.layers.Bidirectional` wrapper can also be used with an RNN layer. This propagates the input forward and backwards through the RNN layer and then concatenates the final output.

   - The main advantage of a bidirectional RNN is that the signal from the beginning of the input doesn't need to be processed all the way through every timestep to affect the output.
   - The main disadvantage of a bidirectional RNN is that you can't efficiently stream predictions as words are being added to the end.
5. After the RNN has converted the sequence to a single vector the two `layers.Dense` do some final processing, and convert from this vector representation to a single logit as the classification output.

The code to implement this is below:

```
model = tf.keras.Sequential([
    encoder,
    tf.keras.layers.Embedding(
        input_dim=len(encoder.get_vocabulary()),
        output_dim=64,
        # Use masking to handle the variable sequence lengths
        mask_zero=True),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1)
])
```

Please note that Keras sequential model is used here since all the layers in the model only have single input and produce single output. In case you want to use stateful RNN layer, you might want to build your model with Keras functional API or model subclassing so that you can retrieve and reuse the RNN layer states. Please check Keras RNN guide for more details.

The embedding layer uses masking to handle the varying sequence-lengths. All the layers after the `Embedding` support masking:

```
print([layer.supports_masking for layer in model.layers])

    [False, True, True, True, True]
```

To confirm that this works as expected, evaluate a sentence twice. First, alone so there's no padding to mask:

```
# predict on a sample text without padding.

sample_text = ('The movie was cool. The animation and the graphics '
               'were out of this world. I would recommend this movie.')
predictions = model.predict(np.array([sample_text]))
print(predictions[0])

    1/1 [==============================] - 6s 6s/step
    [0.00053373]
```

Now, evaluate it again in a batch with a longer sentence. The result should be identical:

```
# predict on a sample text with padding

padding = "the " * 2000
predictions = model.predict(np.array([sample_text, padding]))
print(predictions[0])

    1/1 [==============================] - 0s 66ms/step
    [0.00053373]
```

Compile the Keras model to configure the training process:

```
model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(1e-4),
              metrics=['accuracy'])
```

Train the model

```
history = model.fit(train_dataset, epochs=10,
                    validation_data=test_dataset,
                    validation_steps=30)

    Epoch 1/10
    391/391 [==============================] - 47s 95ms/step - loss: 0.6316 - accuracy: 0.5838 - val_loss: 0.4550 - val_accuracy
    Epoch 2/10
    391/391 [==============================] - 24s 62ms/step - loss: 0.3753 - accuracy: 0.8276 - val_loss: 0.3501 - val_accuracy
    Epoch 3/10
    391/391 [==============================] - 23s 59ms/step - loss: 0.3312 - accuracy: 0.8522 - val_loss: 0.3332 - val_accuracy
    Epoch 4/10
    391/391 [==============================] - 23s 58ms/step - loss: 0.3190 - accuracy: 0.8598 - val_loss: 0.3278 - val_accuracy
    Epoch 5/10
    391/391 [==============================] - 22s 56ms/step - loss: 0.3090 - accuracy: 0.8669 - val_loss: 0.3224 - val_accuracy
    Epoch 6/10
```

```
391/391 [==============================] - 23s 57ms/step - loss: 0.3055 - accuracy: 0.8682 - val_loss: 0.3475 - val_accuracy
Epoch 7/10
391/391 [==============================] - 22s 55ms/step - loss: 0.3025 - accuracy: 0.8677 - val_loss: 0.3183 - val_accuracy
Epoch 8/10
391/391 [==============================] - 22s 57ms/step - loss: 0.3017 - accuracy: 0.8692 - val_loss: 0.3159 - val_accuracy
Epoch 9/10
391/391 [==============================] - 23s 58ms/step - loss: 0.2995 - accuracy: 0.8704 - val_loss: 0.3279 - val_accuracy
Epoch 10/10
391/391 [==============================] - 23s 59ms/step - loss: 0.2984 - accuracy: 0.8720 - val_loss: 0.3214 - val_accuracy
```

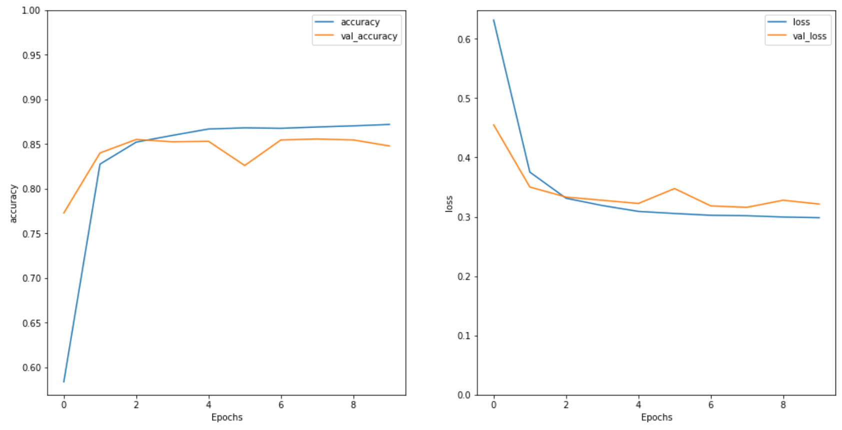
```
test_loss, test_acc = model.evaluate(test_dataset)

print('Test Loss:', test_loss)
print('Test Accuracy:', test_acc)
```

```
391/391 [==============================] - 10s 26ms/step - loss: 0.3172 - accuracy: 0.8518
Test Loss: 0.317198783159256
Test Accuracy: 0.8517600297927856
```

```
plt.figure(figsize=(16, 8))
plt.subplot(1, 2, 1)
plot_graphs(history, 'accuracy')
plt.ylim(None, 1)
plt.subplot(1, 2, 2)
plot_graphs(history, 'loss')
plt.ylim(0, None)
```

```
(0.0, 0.6482444569468498)
```



Run a prediction on a new sentence:

If the prediction is >= 0.0, it is positive else it is negative.

```
sample_text = ('The movie was cool. The animation and the graphics '
               'were out of this world. I would recommend this movie.')
predictions = model.predict(np.array([sample_text]))
```

```
1/1 [==============================] - 3s 3s/step
```

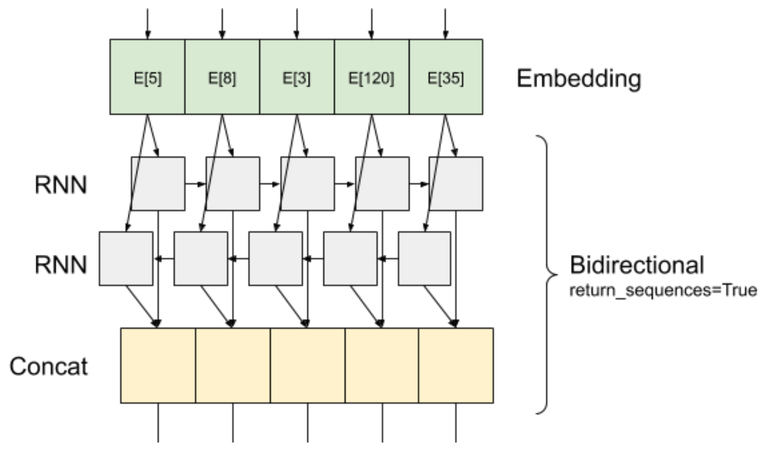
Stack two or more LSTM layers

Keras recurrent layers have two available modes that are controlled by the `return_sequences` constructor argument:

- If `False` it returns only the last output for each input sequence (a 2D tensor of shape (batch_size, output_features)). This is the default, used in the previous model.

- If `True` the full sequences of successive outputs for each timestep is returned (a 3D tensor of shape `(batch_size, timesteps, output_features)`).

Here is what the flow of information looks like with `return_sequences=True`:



The interesting thing about using an `RNN` with `return_sequences=True` is that the output still has 3-axes, like the input, so it can be passed to another RNN layer, like this:

```
model = tf.keras.Sequential([
    encoder,
    tf.keras.layers.Embedding(len(encoder.get_vocabulary()), 64, mask_zero=True),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64,  return_sequences=True)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1)
])
```

```
model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(1e-4),
              metrics=['accuracy'])
```

```
history = model.fit(train_dataset, epochs=10,
                    validation_data=test_dataset,
                    validation_steps=30)
```

```
Epoch 1/10
391/391 [==============================] - 72s 142ms/step - loss: 0.6148 - accuracy: 0.6011 - val_loss: 0.4169 - val_accurac
Epoch 2/10
391/391 [==============================] - 43s 109ms/step - loss: 0.3808 - accuracy: 0.8314 - val_loss: 0.3597 - val_accurac
Epoch 3/10
391/391 [==============================] - 42s 108ms/step - loss: 0.3347 - accuracy: 0.8557 - val_loss: 0.3401 - val_accurac
Epoch 4/10
391/391 [==============================] - 42s 108ms/step - loss: 0.3190 - accuracy: 0.8631 - val_loss: 0.3241 - val_accurac
Epoch 5/10
391/391 [==============================] - 41s 105ms/step - loss: 0.3125 - accuracy: 0.8666 - val_loss: 0.3337 - val_accurac
Epoch 6/10
391/391 [==============================] - 42s 108ms/step - loss: 0.3088 - accuracy: 0.8670 - val_loss: 0.3183 - val_accurac
Epoch 7/10
391/391 [==============================] - 42s 107ms/step - loss: 0.3037 - accuracy: 0.8694 - val_loss: 0.3250 - val_accurac
Epoch 8/10
391/391 [==============================] - 42s 107ms/step - loss: 0.2989 - accuracy: 0.8698 - val_loss: 0.3168 - val_accurac
Epoch 9/10
391/391 [==============================] - 42s 107ms/step - loss: 0.3006 - accuracy: 0.8692 - val_loss: 0.3182 - val_accurac
Epoch 10/10
391/391 [==============================] - 44s 111ms/step - loss: 0.2957 - accuracy: 0.8709 - val_loss: 0.3170 - val_accurac
```

```
test_loss, test_acc = model.evaluate(test_dataset)

print('Test Loss:', test_loss)
print('Test Accuracy:', test_acc)
```

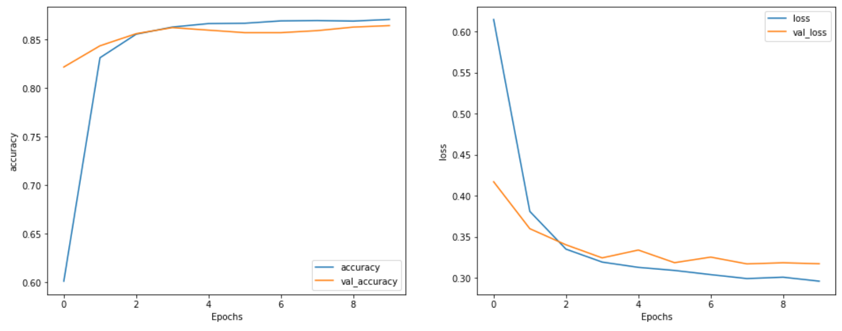
```
391/391 [==============================] - 17s 43ms/step - loss: 0.3183 - accuracy: 0.8639
Test Loss: 0.3183119595050812
Test Accuracy: 0.8638799786567688
```

```
# predict on a sample text without padding.

sample_text = ('The movie was not good. The animation and the graphics '
               'were terrible. I would not recommend this movie.')
predictions = model.predict(np.array([sample_text]))
print(predictions)
```

```
    1/1 [==============================] - 5s 5s/step
    [[-1.53878]]
```

```
plt.figure(figsize=(16, 6))
plt.subplot(1, 2, 1)
plot_graphs(history, 'accuracy')
plt.subplot(1, 2, 2)
plot_graphs(history, 'loss')
```



Check out other existing recurrent layers such as [GRU layers](https://www.tensorflow.org/api_docs/python/tf/keras/layers/GRU).

If you're interested in building custom RNNs, see the [Keras RNN Guide](https://www.tensorflow.org/guide/keras/rnn).

✓ 0s completed at 12:00 PM