basic_text_classification

April 24, 2019

Copyright 2018 The TensorFlow Authors.

1 Text classification with movie reviews

```
<a target="_blank" href="https://www.tensorflow.org/tutorials/keras/basic_text_classification">

<a target="_blank" href="https://colab.research.google.com/github/tensorflow/docs/blob/master/s

<a target="_blank" href="https://github.com/tensorflow/docs/blob/master/site/en/tutorials/keras
```

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.

```
In [0]: 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__)
```

1.1 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):

```
In [0]: imdb = keras.datasets.imdb

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

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.

1.2 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.

```
In [0]: print("Training entries: {}, labels: {}".format(len(train_data), len(train_labels)))
```

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:

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

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.

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

1.2.1 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:

```
In [0]: # 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])
```

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

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

1.3 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:

```
In [0]: 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:

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

And inspect the (now padded) first review:

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

1.4 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:

```
In [0]: # 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()
```

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.

1.4.1 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.

1.4.2 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:

```
In [0]: model.compile(optimizer='adam',
                      loss='binary_crossentropy',
                      metrics=['acc'])
```

1.5 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).

```
In [0]: x_val = train_data[:10000]
        partial_x_train = train_data[10000:]

        y_val = train_labels[:10000]
        partial_y_train = train_labels[10000:]
```

1.6 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:

```
In [0]: history = model.fit(partial_x_train,
                            partial_y_train,
                            epochs=40,
                            batch_size=512,
                            validation_data=(x_val, y_val),
                            verbose=1)
```

1.7 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.

```
In [0]: results = model.evaluate(test_data, test_labels)

        print(results)
```

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

1.8 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:

```
In [0]: history_dict = history.history
        history_dict.keys()
```

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:

```
In [0]: 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()

In [0]: 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.