Lecture_4_IMDB_binary classification

October 27, 2022

1 Classifying movie reviews: A binary classification example

1.1 The IMDB dataset

We'll work with the IMDB dataset: a set of 50,000 highly polarized reviews from the Internet Movie Database. They're split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting of 50% negative and 50% positive reviews.

Just like the MNIST dataset, the IMDB dataset comes packaged with Keras. It has already been preprocessed: the reviews (sequences of words) have been turned into sequences of integers, where each integer stands for a specific word in a dictionary.

Loading the IMDB dataset

[1]:
```
from tensorflow.keras.datasets import imdb
(train_data, train_labels), (test_data, test_labels) = imdb.
 ↪load_data(num_words=10000)
```

The argument `num_words=10000` means we'll only keep the top 10,000 most frequently occurring words in the training data, i.e. words are indexed by overall frequency in the dataset. For instance, the integer "3" encodes the 3rd most frequent word in the data. Rare words will be discarded. This allows us to work with vector data of manageable size. If we didn't set this limit, we'd be working with 88,585 unique words in the training data, which is unnecessarily large. Many of these words only occur in a single sample, and thus can't be meaningfully used for classification.

The variables `train_data` and `test_data` are lists of reviews; each review is a list of word indices (encoding a sequence of words). `train_labels` and `test_labels` are lists of 0s and 1s, where 0 stands for negative and 1 stands for positive review.

[2]:
```
#train_data[0]
```

[3]:
```
train_labels[0]
```

[3]:
```
1
```

Because we're restricting ourselves to the top 10,000 most frequent words, no word index will exceed 10,000.

[4]:
```
max([max(sequence) for sequence in train_data])
```

[4]:
```
9999
```

Decoding reviews back to text

Let's decode one of the reviews back to English words.

[5]:
```
word_index = imdb.get_word_index()    # Dictionary of words {word:index}
reverse_word_index = dict(            # Dictionary of words {index:word}
    [(value, key) for (key, value) in word_index.items()])
index_from = 3                        # Actual words are indexed with index_from␣
 ↪and higher
decoded_review = " ".join(
    [reverse_word_index.get(i - index_from, "?") for i in train_data[0]])
#print(decoded_review)
```

1.2 Preparing the data

We can't directly feed lists of integers into a neural network. They all have different lengths, but a neural network expects to process contiguous batches of data. You have to turn your lists into tensors. There are two ways to do that:

- Pad your lists so that they all have the same length, turn them into an integer tensor of shape `(samples, max_length)`, and start your model with a layer capable of handling such integer tensors.
- **Multi-hot encode** your lists to turn them into vectors of 0s and 1s. This would mean, for instance, turning the sequence [8, 5] into a 10,000-dimensional vector that would be all 0s except for indices 8 and 5, which would be 1s. Then you could use a Dense layer, capable of handling floating-point vector data, as the first layer in your model.

We'll use the latter solution to vectorize the data.

Encoding the integer sequences via multi-hot encoding

[6]:
```
import numpy as np
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        for j in sequence:
            results[i, j] = 1.
    return results
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

[7]:
```
x_train[0]
```

[7]:
```
array([0., 1., 1., …, 0., 0., 0.])
```

Let's convert our labels to float32 numbers

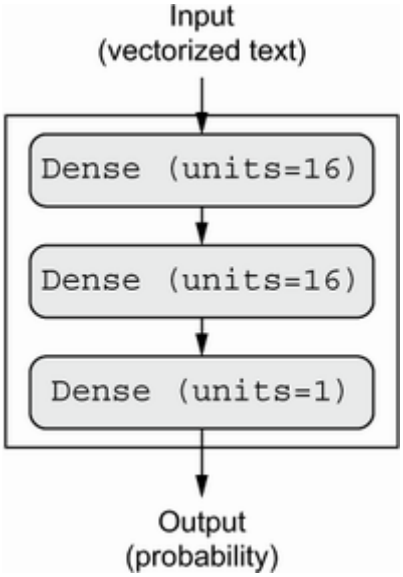
[8]:
```
y_train = train_labels.astype('float32')
y_test = test_labels.astype('float32')
```

1.3 Building the model

The input data are vectors, and the labels are scalars (1s and 0s): this is one of the simplest problem setups we'll ever encounter. A type of model that performs well on such a problem is a plain stack of densely connected (Dense) layers with relu activations.

There are two key architecture decisions to be made about such a stack of Dense layers: * How many layers to use * How many neurons to choose for each layer

We'll go with the following architecture choices: * Two intermediate layers with 16 units each * A third layer that will output the scalar prediction regarding the sentiment of the current review



Model definition

[9]:
```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```

The first argument being passed to each `Dense` layer is the number of units in the layer: the dimensionality of representation space of the layer. Each such `Dense` layer with a `relu` activation implements the following chain of tensor operations:

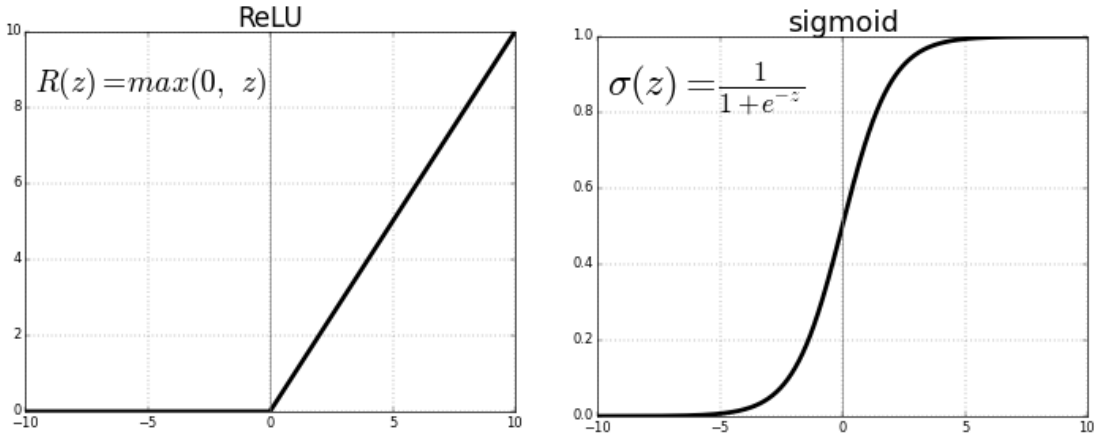
```
output = relu(dot(input, W) + b)
```

16 units in the layer means the weight matrix `W` will have shape `(input_dimension, 16)`: the dot product with `W` will project the input data onto a 16-dimensional representation space. Having more units (a higher-dimensional representation space) allows our model to learn more-complex representations, but it makes the model more computationally expensive and may lead to learning unwanted patterns present only in the training data.

The intermediate layers use `relu` (rectified linear unit) as their activation function, and the final layer uses a `sigmoid` activation.

`relu` is a function which zeros out negative values.

`sigmoid` is always used in binary classification problems. It outputs values within [0, 1] and therefore can be interpreted as a probability. In our case, it indicates how likely the review is positive (review is declared positive if probability > 0.5).



What are activation functions, and why are they necessary?

Without an activation function like relu (also called a non-linearity), the Dense layer would consist of two linear operations — a dot product and an addition:

output = dot(input, W) + b

The layer could only learn linear transformations (affine transformations) of the input data: the hypothesis space of the layer would be the set of all possible linear transformations of the input data into a 16-dimensional space. Such a hypothesis space is too restricted and wouldn't benefit from multiple layers of representations, because a **deep stack of linear layers would still implement a linear operation: adding more layers wouldn't extend the hypothesis space**. In order to get access to a much richer hypothesis space that will benefit from deep representations, we need a non-linearity, or activation function. relu is the most popular activation function in deep learning.

Finally, we need to choose a **loss function** and an **optimizer**. Because we're facing a binary classification problem and the output of our model is a probability, it's best to use the `binary_crossentropy` loss. We could also use other losses, such as `mean_squared_error`, but crossentropy is usually the best choice with models that output probabilities.

Crossentropy is a quantity from the field of information theory that measures the distance between probability distributions or, in this case, between the ground-truth distribution and your predictions.

As for the choice of the optimizer, we'll go with `rmsprop`, which is a usually a good default choice for virtually any problem.

Let's compile our model!

[10]:
```
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
```

1.4 Validating our approach

A deep learning model should **never be evaluated on its training data** — it's standard practice to use a **validation set** to monitor the accuracy of the model during training.

Here, we'll create a validation set by setting apart 10,000 samples from the original training data.

[11]:
```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

Training the model

We will now train the model for 20 epochs in mini-batches of 512 samples. At the same time, we will monitor loss and accuracy on the 10,000 samples that we set apart. We do so by passing the validation data as the `validation_data` argument.

[12]:
```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val),
                    verbose=0)      # verbose=2 = one line per epoch, no␣
 ↪progress bar
```

The call to `model.fit()` returns a **`History`** object, which has a member history - a dictionary containing data about everything that happened during training. Let's look at it:

[13]:
```
history_dict = history.history
history_dict.keys()
```

[13]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

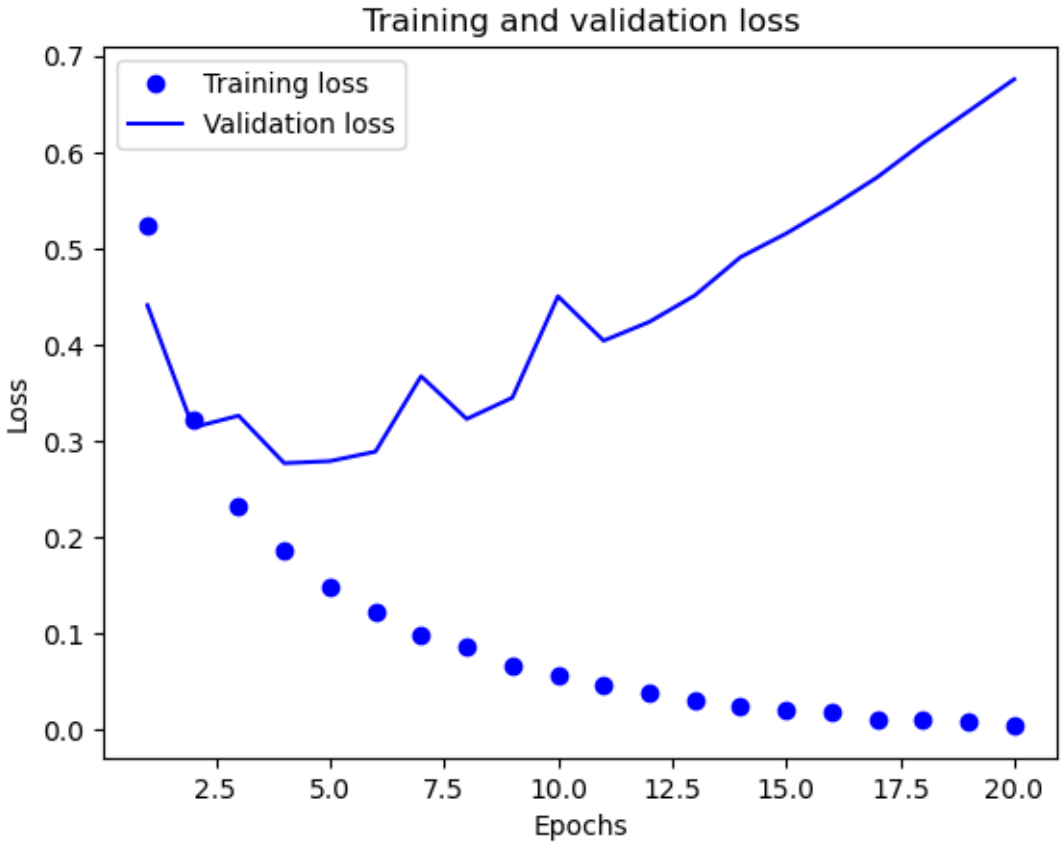
The `History` dictionary contains four entries: one per metric that was being monitored during training and during validation.

Plotting the training and validation loss

[14]:
```
import matplotlib.pyplot as plt
history_dict = history.history
loss = history_dict["loss"]
val_loss = history_dict["val_loss"]
epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, "bo", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
```

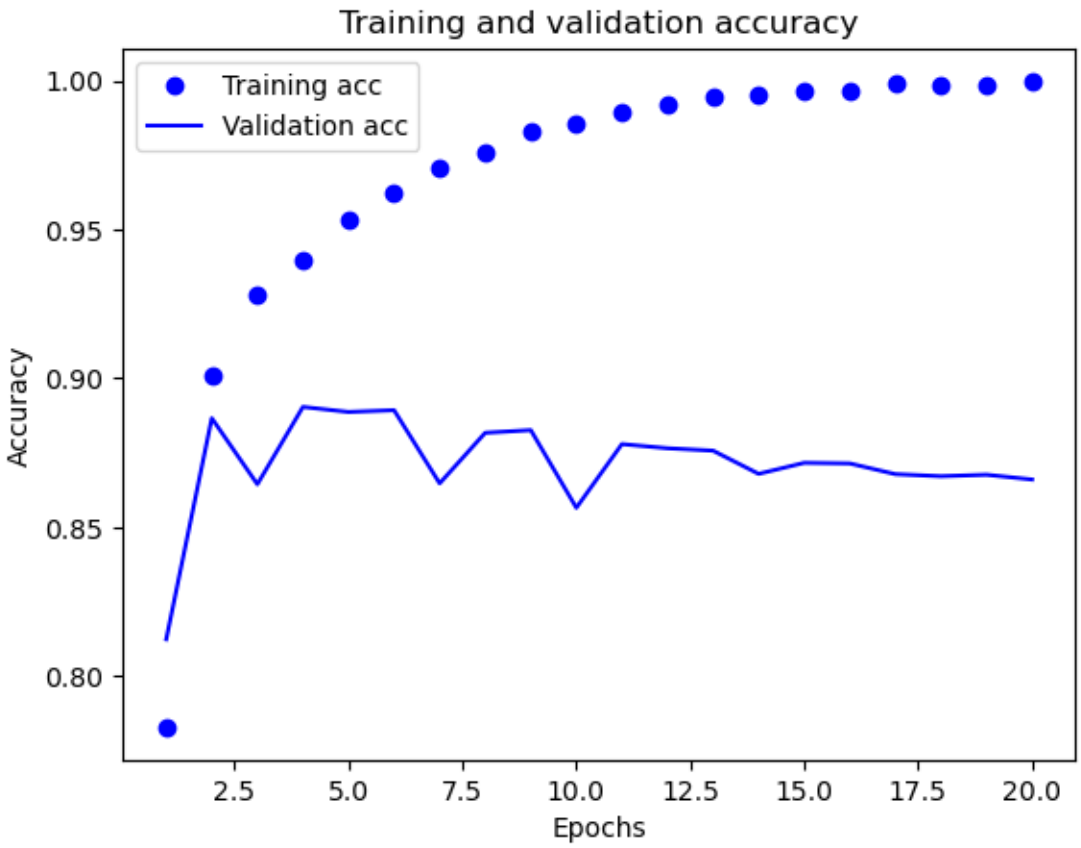
```
plt.title("Training and validation loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



Plotting the training and validation accuracy

[15]:
```
plt.clf()
acc = history_dict["accuracy"]
val_acc = history_dict["val_accuracy"]

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()
```



As we can see, the training loss decreases with every epoch, and the training accuracy increases with every epoch. That's what we would expect when running gradient-descent optimization — the quantity we're trying to minimize should be less with every iteration. But that isn't the case for the validation loss and accuracy: they seem to peak at the fourth epoch.

A model which performs well on the training data isn't necessarily a model that will perform well on data not seen before. What we see here is called overfitting: after the fourth epoch, we're *overoptimizing* on the training data, and we end up learning representations that are specific to the training data and don't generalize to data outside of the training set.

In this case, to prevent overfitting, you could stop training after four epochs. In general, you can use a range of techniques to mitigate overfitting, which we'll cover later. Be sure to always monitor performance on data that is outside of the training set!

Retraining a model from scratch

Let's train a new model from scratch for four epochs and then evaluate it on the test data.

[16]:
```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
```

```
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)
```

```
Epoch 1/4
49/49 [==============================] - 1s 7ms/step - loss: 0.4530 - accuracy:
0.8259
Epoch 2/4
49/49 [==============================] - 0s 7ms/step - loss: 0.2551 - accuracy:
0.9094
Epoch 3/4
49/49 [==============================] - 0s 7ms/step - loss: 0.1954 - accuracy:
0.9305
Epoch 4/4
49/49 [==============================] - 0s 6ms/step - loss: 0.1647 - accuracy:
0.9406
782/782 [==============================] - 1s 1ms/step - loss: 0.2988 -
accuracy: 0.8821
```

[17]:
```
results
```

[17]:
```
[0.298782080411911, 0.8821200132369995]
```

1.4.1 Using a trained model to generate predictions on new data

[18]:
```
model.predict(x_test)
```

```
782/782 [==============================] - 1s 874us/step
```

[18]:
```
array([[0.18081577],
       [0.9997718 ],
       [0.6410961 ],
       …,
       [0.09428697],
       [0.05419033],
       [0.4454011 ]], dtype=float32)
```

As we can see, the model is confident for some samples (close to 1 or 0) but less confident for others (within [0.4-0.6]).

1.4.2 Further experiments

The architecture choices we've made are fairly reasonable, although there's still room for improvement: * We used two representation layers before the final classification layer. Try using one or three representation layers, and see how doing so affects validation and test accuracy. * Try using layers with more units or fewer units: 32 units, 64 units, and so on. * Try using the `mse` loss

function instead of `binary_crossentropy`. * Try using the `tanh` activation (an activation that was popular in the early days of neural networks) instead of `relu`.