Assignment 5.2

April 18, 2021

0.1 Assignment 5.2

Implement the news classifier found in section 3.5 of Deep Learning with Python.

[1]:
```
import keras
keras.__version__
```

[1]:
```
'2.4.3'
```

Classifying newswires: a multi-class classification example
The Reuters dataset: We will be working with the Reuters dataset, a set of short newswires and their topics, published by Reuters in 1986. It's a very simple, widely used toy dataset for text classification. There are 46 different topics; some topics are more represented than others, but each topic has at least 10 examples in the training set.
Like IMDB and MNIST, the Reuters dataset comes packaged as part of Keras. Let's take a look right away:

[2]:
```
from keras.datasets import reuters
(train_data, train_labels), (test_data, test_labels) = reuters.
 ↪load_data(num_words=10000)
```

```
/opt/conda/lib/python3.8/site-
packages/tensorflow/python/keras/datasets/reuters.py:148:
VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences
(which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths
or shapes) is deprecated. If you meant to do this, you must specify
'dtype=object' when creating the ndarray
  x_train, y_train = np.array(xs[:idx]), np.array(labels[:idx])
/opt/conda/lib/python3.8/site-
packages/tensorflow/python/keras/datasets/reuters.py:149:
VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences
(which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths
or shapes) is deprecated. If you meant to do this, you must specify
'dtype=object' when creating the ndarray
  x_test, y_test = np.array(xs[idx:]), np.array(labels[idx:])
```

Like with the IMDB dataset, the argument num_words=10000 restricts the data to the 10,000 most frequently occurring words found in the data.
We have 8,982 training examples and 2,246 test examples:

[3]:
```
len(train_data)
```

[3]: 8982

[4]:
```
len(test_data)
```

[4]: 2246

As with the IMDB reviews, each example is a list of integers (word indices):

[5]:
```
train_data[10]
```

[5]:
```
[1,
 245,
 273,
 207,
 156,
 53,
 74,
 160,
 26,
 14,
 46,
 296,
 26,
 39,
 74,
 2979,
 3554,
 14,
 46,
 4689,
 4329,
 86,
 61,
 3499,
 4795,
 14,
 61,
 451,
 4329,
 17,
 12]
```

Here's how you can decode it back to words:

[6]:
```
word_index = reuters.get_word_index()
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
# Note that our indices were offset by 3
# because 0, 1 and 2 are reserved indices for "padding", "start of sequence",␣
↪and "unknown".
decoded_newswire = ' '.join([reverse_word_index.get(i - 3, '?') for i in␣
↪train_data[0]])
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/reuters_word_index.json
557056/550378 [==============================] - 0s 0us/step
```

[7]:
```
decoded_newswire
```

[7]:
```
'? ? ? said as a result of its december acquisition of space co it expects
earnings per share in 1987 of 1 15 to 1 30 dlrs per share up from 70 cts in 1986
the company said pretax net should rise to nine to 10 mln dlrs from six mln dlrs
in 1986 and rental operation revenues to 19 to 22 mln dlrs from 12 5 mln dlrs it
said cash flow per share this year should be 2 50 to three dlrs reuter 3'
```

[8]:
```
train_labels[10]
```

[8]:
```
3
```

Preparing the data: We can vectorize the data with the exact same code as in our previous example:

[9]:
```
import numpy as np
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results
# Our vectorized training data
x_train = vectorize_sequences(train_data)
# Our vectorized test data
x_test = vectorize_sequences(test_data)
```

One-hot encoding of our labels consists in embedding each label as an all-zero vector with a 1 in the place of the label index, e.g.:

[10]:
```
def to_one_hot(labels, dimension=46):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results
# Our vectorized training labels
one_hot_train_labels = to_one_hot(train_labels)
```

```
# Our vectorized test labels
one_hot_test_labels = to_one_hot(test_labels)
```

[11]:
```
from keras.utils.np_utils import to_categorical

one_hot_train_labels = to_categorical(train_labels)
one_hot_test_labels = to_categorical(test_labels)
```

Building our network: This topic classification problem looks very similar to our previous movie review classification problem: in both cases, we are trying to classify short snippets of text. There is however a new constraint here: the number of output classes has gone from 2 to 46, i.e. the dimensionality of the output space is much larger.
In a stack of Dense layers like what we were using, each layer can only access information present in the output of the previous layer. If one layer drops some information relevant to the classification problem, this information can never be recovered by later layers: each layer can potentially become an "information bottleneck". In our previous example, we were using 16-dimensional intermediate layers, but a 16-dimensional space may be too limited to learn to separate 46 different classes: such small layers may act as information bottlenecks, permanently dropping relevant information.
For this reason we will use larger layers. Let's go with 64 units:

[12]:
```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
```

The best loss function to use in this case is categorical_crossentropy. It measures the distance between two probability distributions: in our case, between the probability distribution output by our network, and the true distribution of the labels. By minimizing the distance between these two distributions, we train our network to output something as close as possible to the true labels

[13]:
```
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Validating our approach: Let's set apart 1,000 samples in our training data to use as a validation set:

[14]:
```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]

y_val = one_hot_train_labels[:1000]
partial_y_train = one_hot_train_labels[1000:]
```

Now let's train our network for 20 epochs:

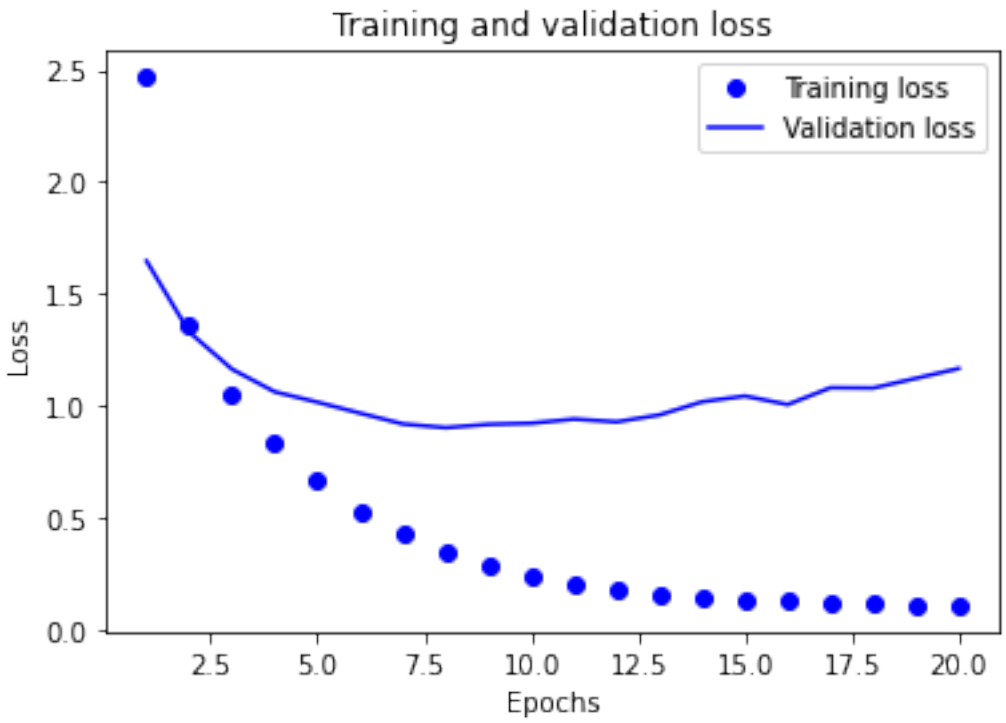
[15]:
```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

```
Epoch 1/20
16/16 [==============================] - 1s 46ms/step - loss: 2.9907 - accuracy:
0.4446 - val_loss: 1.6475 - val_accuracy: 0.6410
Epoch 2/20
16/16 [==============================] - 0s 18ms/step - loss: 1.4348 - accuracy:
0.6887 - val_loss: 1.3301 - val_accuracy: 0.6980
Epoch 3/20
16/16 [==============================] - 0s 16ms/step - loss: 1.0664 - accuracy:
0.7710 - val_loss: 1.1648 - val_accuracy: 0.7540
Epoch 4/20
16/16 [==============================] - 0s 14ms/step - loss: 0.8750 - accuracy:
0.8138 - val_loss: 1.0637 - val_accuracy: 0.7800
Epoch 5/20
16/16 [==============================] - 0s 17ms/step - loss: 0.6730 - accuracy:
0.8591 - val_loss: 1.0175 - val_accuracy: 0.7890
Epoch 6/20
16/16 [==============================] - 0s 21ms/step - loss: 0.5371 - accuracy:
0.8834 - val_loss: 0.9683 - val_accuracy: 0.7980
Epoch 7/20
16/16 [==============================] - 0s 20ms/step - loss: 0.4307 - accuracy:
0.9128 - val_loss: 0.9202 - val_accuracy: 0.8110
Epoch 8/20
16/16 [==============================] - 0s 19ms/step - loss: 0.3340 - accuracy:
0.9312 - val_loss: 0.9038 - val_accuracy: 0.8140
Epoch 9/20
16/16 [==============================] - 0s 15ms/step - loss: 0.2751 - accuracy:
0.9441 - val_loss: 0.9187 - val_accuracy: 0.8100
Epoch 10/20
16/16 [==============================] - 0s 15ms/step - loss: 0.2312 - accuracy:
0.9482 - val_loss: 0.9235 - val_accuracy: 0.8060
Epoch 11/20
16/16 [==============================] - 0s 16ms/step - loss: 0.1943 - accuracy:
0.9546 - val_loss: 0.9421 - val_accuracy: 0.8080
Epoch 12/20
16/16 [==============================] - 0s 18ms/step - loss: 0.1689 - accuracy:
0.9562 - val_loss: 0.9299 - val_accuracy: 0.8120
Epoch 13/20
16/16 [==============================] - 0s 17ms/step - loss: 0.1507 - accuracy:
0.9592 - val_loss: 0.9607 - val_accuracy: 0.8160
Epoch 14/20
16/16 [==============================] - 0s 17ms/step - loss: 0.1457 - accuracy:
```

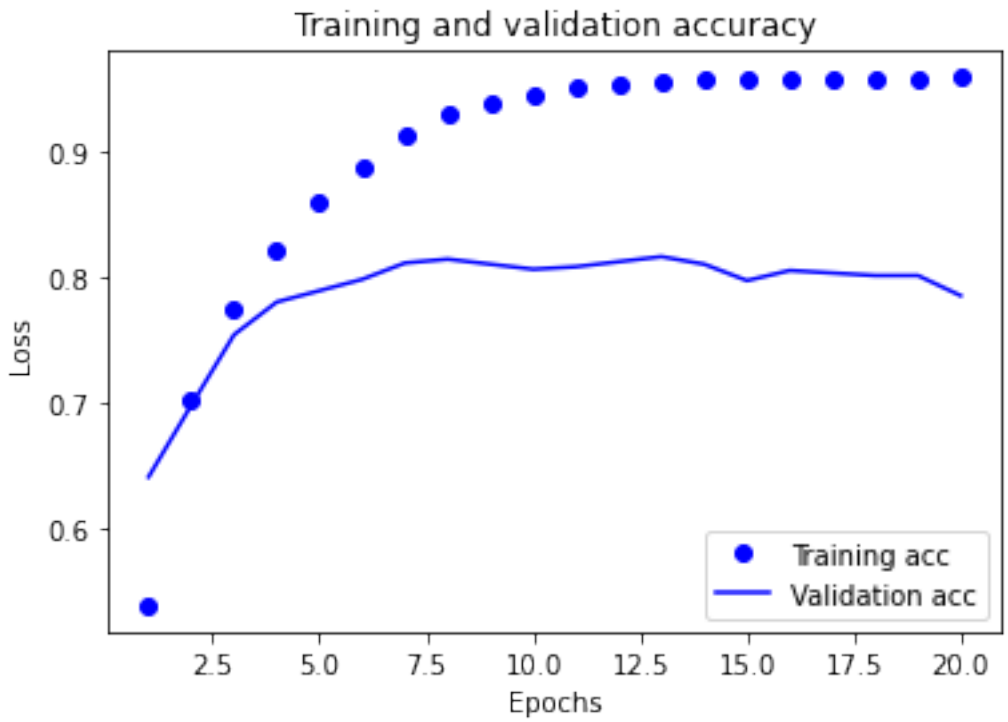
```
0.9606 - val_loss: 1.0196 - val_accuracy: 0.8100
Epoch 15/20
16/16 [==============================] - 0s 20ms/step - loss: 0.1245 - accuracy:
0.9626 - val_loss: 1.0450 - val_accuracy: 0.7970
Epoch 16/20
16/16 [==============================] - 0s 18ms/step - loss: 0.1171 - accuracy:
0.9638 - val_loss: 1.0070 - val_accuracy: 0.8050
Epoch 17/20
16/16 [==============================] - 0s 18ms/step - loss: 0.1143 - accuracy:
0.9613 - val_loss: 1.0820 - val_accuracy: 0.8030
Epoch 18/20
16/16 [==============================] - 0s 17ms/step - loss: 0.1081 - accuracy:
0.9624 - val_loss: 1.0808 - val_accuracy: 0.8010
Epoch 19/20
16/16 [==============================] - 0s 18ms/step - loss: 0.1060 - accuracy:
0.9587 - val_loss: 1.1239 - val_accuracy: 0.8010
Epoch 20/20
16/16 [==============================] - 0s 16ms/step - loss: 0.1036 - accuracy:
0.9638 - val_loss: 1.1680 - val_accuracy: 0.7850
```

Let's display its loss and accuracy curves:

[16]:
```python
import matplotlib.pyplot as plt

loss = history.history['loss']
val_loss = history.history['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()
```



[17]:
```
plt.clf() # clear figure

acc = history.history['accuracy']
val_acc = history.history['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('Loss')
plt.legend()
plt.show()
```



It seems that the network starts overfitting after 8 epochs. Let's train a new network from scratch for 8 epochs, then let's evaluate it on the test set:

[18]:
```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(partial_x_train,
          partial_y_train,
          epochs=8,
          batch_size=512,
          validation_data=(x_val, y_val))
results = model.evaluate(x_test, one_hot_test_labels)
```

```
Epoch 1/8
16/16 [==============================] - 1s 31ms/step - loss: 3.1405 - accuracy:
0.4146 - val_loss: 1.7481 - val_accuracy: 0.6390
Epoch 2/8
16/16 [==============================] - 0s 19ms/step - loss: 1.4844 - accuracy:
0.7017 - val_loss: 1.2834 - val_accuracy: 0.7280
```

```
Epoch 3/8
16/16 [==============================] - 0s 15ms/step - loss: 1.0594 - accuracy:
0.7765 - val_loss: 1.1051 - val_accuracy: 0.7680
Epoch 4/8
16/16 [==============================] - 0s 15ms/step - loss: 0.8244 - accuracy:
0.8266 - val_loss: 1.0296 - val_accuracy: 0.7800
Epoch 5/8
16/16 [==============================] - 0s 16ms/step - loss: 0.6273 - accuracy:
0.8671 - val_loss: 0.9526 - val_accuracy: 0.8060
Epoch 6/8
16/16 [==============================] - 0s 16ms/step - loss: 0.5129 - accuracy:
0.8945 - val_loss: 0.9017 - val_accuracy: 0.8100
Epoch 7/8
16/16 [==============================] - 0s 15ms/step - loss: 0.4104 - accuracy:
0.9184 - val_loss: 0.8931 - val_accuracy: 0.8110
Epoch 8/8
16/16 [==============================] - 0s 18ms/step - loss: 0.3260 - accuracy:
0.9355 - val_loss: 0.9248 - val_accuracy: 0.8100
71/71 [==============================] - 0s 2ms/step - loss: 1.0042 - accuracy:
0.7747
```

[19]:
```
results
```

[19]:
```
[1.004197359085083, 0.7747105956077576]
```

[20]:
```
import copy
test_labels_copy = copy.copy(test_labels)
np.random.shuffle(test_labels_copy)
float(np.sum(np.array(test_labels) == np.array(test_labels_copy))) /
 ↪len(test_labels)
```

[20]:
```
0.1918967052537845
```

Our approach reaches an accuracy of ~77%. With a balanced binary classification problem, the accuracy reached by a purely random classifier would be 50%, but in our case it is closer to 19%, so our results seem pretty good, at least when compared to a random baseline:

Generating predictions on new data: We can verify that the predict method of our model instance returns a probability distribution over all 46 topics. Let's generate topic predictions for all of the test data:

[21]:
```
predictions = model.predict(x_test)
```

Each entry in predictions is a vector of length 46:

[22]:
```
predictions[0].shape
```

[22]:
```
(46,)
```

The coefficients in this vector sum to 1:

[23]:
```
np.sum(predictions[0])
```

[23]:
```
1.0000001
```

The largest entry is the predicted class, i.e. the class with the highest probability:

[24]:
```
np.argmax(predictions[0])
```

[24]:
```
4
```

A different way to handle the labels and the loss: We mentioned earlier that another way to encode the labels would be to cast them as an integer tensor, like such:

[25]:
```
y_train = np.array(train_labels)
y_test = np.array(test_labels)
```

[26]:
```
model.compile(optimizer='rmsprop',␣
↪loss='sparse_categorical_crossentropy',metrics=['acc'])
```

This new loss function is still mathematically the same as categorical_crossentropy; it just has a different interface.

On the importance of having sufficiently large intermediate layers: We mentioned earlier that since our final outputs were 46-dimensional, we should avoid intermediate layers with much less than 46 hidden units. Now let's try to see what happens when we introduce an information bottleneck by having intermediate layers significantly less than 46-dimensional, e.g. 4-dimensional.

[27]:
```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(partial_x_train,
          partial_y_train,
          epochs=20,
          batch_size=128,
          validation_data=(x_val, y_val))
```

```
Epoch 1/20
63/63 [==============================] - 1s 14ms/step - loss: 3.5368 - accuracy:
0.0289 - val_loss: 2.7304 - val_accuracy: 0.3040
Epoch 2/20
63/63 [==============================] - 1s 9ms/step - loss: 2.3752 - accuracy:
0.3380 - val_loss: 1.7233 - val_accuracy: 0.6270
Epoch 3/20
63/63 [==============================] - 1s 9ms/step - loss: 1.4869 - accuracy:
```

```
0.6595 - val_loss: 1.4367 - val_accuracy: 0.6560
Epoch 4/20
63/63 [==============================] - 0s 7ms/step - loss: 1.2369 - accuracy:
0.6870 - val_loss: 1.3566 - val_accuracy: 0.6610
Epoch 5/20
63/63 [==============================] - 0s 8ms/step - loss: 1.0590 - accuracy:
0.7179 - val_loss: 1.3221 - val_accuracy: 0.6740
Epoch 6/20
63/63 [==============================] - 0s 8ms/step - loss: 0.9597 - accuracy:
0.7434 - val_loss: 1.3038 - val_accuracy: 0.6890
Epoch 7/20
63/63 [==============================] - 0s 7ms/step - loss: 0.8673 - accuracy:
0.7708 - val_loss: 1.3030 - val_accuracy: 0.7050
Epoch 8/20
63/63 [==============================] - 0s 7ms/step - loss: 0.8036 - accuracy:
0.7885 - val_loss: 1.3094 - val_accuracy: 0.7060
Epoch 9/20
63/63 [==============================] - 1s 8ms/step - loss: 0.7397 - accuracy:
0.7991 - val_loss: 1.3303 - val_accuracy: 0.7120
Epoch 10/20
63/63 [==============================] - 0s 8ms/step - loss: 0.6742 - accuracy:
0.8185 - val_loss: 1.3589 - val_accuracy: 0.7080
Epoch 11/20
63/63 [==============================] - 0s 7ms/step - loss: 0.6247 - accuracy:
0.8338 - val_loss: 1.3880 - val_accuracy: 0.7180
Epoch 12/20
63/63 [==============================] - 0s 7ms/step - loss: 0.5710 - accuracy:
0.8494 - val_loss: 1.4050 - val_accuracy: 0.7230
Epoch 13/20
63/63 [==============================] - 0s 6ms/step - loss: 0.5355 - accuracy:
0.8546 - val_loss: 1.4809 - val_accuracy: 0.7150
Epoch 14/20
63/63 [==============================] - 0s 6ms/step - loss: 0.5173 - accuracy:
0.8560 - val_loss: 1.5168 - val_accuracy: 0.7180
Epoch 15/20
63/63 [==============================] - 0s 7ms/step - loss: 0.4763 - accuracy:
0.8671 - val_loss: 1.5549 - val_accuracy: 0.7190
Epoch 16/20
63/63 [==============================] - 0s 6ms/step - loss: 0.4752 - accuracy:
0.8674 - val_loss: 1.6433 - val_accuracy: 0.7200
Epoch 17/20
63/63 [==============================] - 0s 6ms/step - loss: 0.4598 - accuracy:
0.8701 - val_loss: 1.7026 - val_accuracy: 0.7080
Epoch 18/20
63/63 [==============================] - 0s 6ms/step - loss: 0.4265 - accuracy:
0.8811 - val_loss: 1.7439 - val_accuracy: 0.7200
Epoch 19/20
63/63 [==============================] - 0s 7ms/step - loss: 0.4136 - accuracy:
```

```
0.8828 - val_loss: 1.7817 - val_accuracy: 0.7160
Epoch 20/20
63/63 [==============================] - 0s 6ms/step - loss: 0.3959 - accuracy:
0.8914 - val_loss: 1.8233 - val_accuracy: 0.7020
```

[27]:
```
<tensorflow.python.keras.callbacks.History at 0x7fa6583e0e50>
```

Our network now seems to peak at ~70% test accuracy, a 8% absolute drop. This drop is mostly due to the fact that we are now trying to compress a lot of information (enough information to recover the separation hyperplanes of 46 classes) into an intermediate space that is too low-dimensional. The network is able to cram most of the necessary information into these 8-dimensional representations, but not all of it.

Reference: https://github.com/fchollet/deep-learning-with-python-notebooks

[ ]: