DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

Assignment 05 Code and Outputs

For Assignment 5, I executed the code locally and had the following outputs with the example code. The chapter code for the movie reviews, newswires, and house prices examples were found through the Deep Learning with Python Github Link: deep-learning-with-python-notebooks/first_edition at master · fchollet/deep-learning-with-python-notebooks (github.com).

Assignment 5.1 Code and Outputs:

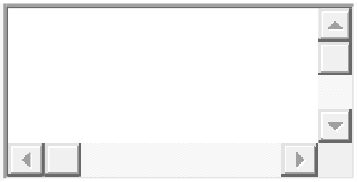
Assignment 5.1

DSC650-T302 Big Data (2235-1)

Jake Meyer

04/13/2023

In [1]:



```
import keras
keras.__version__
```

Out[1]:

```
'2.11.0'
```

Classifying movie reviews: a binary classification example

This notebook contains the code samples found in Chapter 3, Section 5 of Deep Learning with Python. Note that the original text features far more content, in particular further explanations and figures: in this notebook, you will only find source code and related comments.

Two-class classification, or binary classification, may be the most widely applied kind of machine learning problem. In this example, we will learn to classify movie reviews into "positive" reviews and "negative" reviews, just based on the text content of the reviews.

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

The IMDB dataset

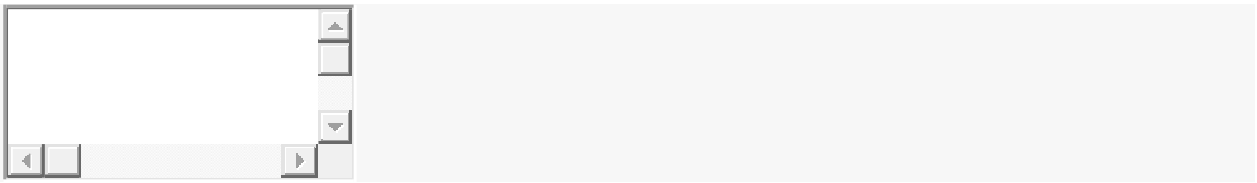
We'll be working with "IMDB dataset", a set of 50,000 highly-polarized reviews from the Internet Movie Database. They are split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting in 50% negative and 50% positive reviews.

Why do we have these two separate training and test sets? You should never test a machine learning model on the same data that you used to train it! Just because a model performs well on its training data doesn't mean that it will perform well on data it has never seen, and what you actually care about is your model's performance on new data (since you already know the labels of your training data -- obviously you don't need your model to predict those). For instance, it is possible that your model could end up merely *memorizing* a mapping between your training samples and their targets -- which would be completely useless for the task of predicting targets for data never seen before. We will go over this point in much more detail in the next chapter.

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.

The following code will load the dataset (when you run it for the first time, about 80MB of data will be downloaded to your machine):

In [2]:



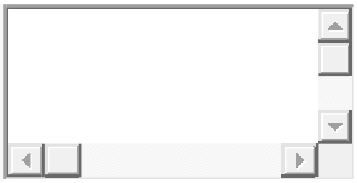
```
from keras.datasets import imdb

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

The argument `num_words=10000` means that we will only keep the top 10,000 most frequently occurring words in the training data. Rare words will be discarded. This allows us to work with vector data of manageable size.

The variables `train_data` and `test_data` are lists of reviews, each review being 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":

In [3]:

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023



```
train_data[0]
```

Out[3]:

```
[1,
 14,
 22,
 16,
 43,
 530,
 973,
 1622,
 1385,
 65,
 458,
 4468,
 66,
 3941,
 4,
 173,
 36,
 256,
 5,
 25,
 100,
 43,
 838,
 112,
 50,
 670,
 2,
 9,
 35,
 480,
 284,
 5,
 150,
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

```
 4,
 172,
 112,
 167,
 2,
 336,
 385,
 39,
 4,
 172,
 4536,
 1111,
 17,
 546,
 38,
 13,
 447,
 4,
 192,
 50,
 16,
 6,
 147,
 2025,
 19,
 14,
 22,
 4,
 1920,
 4613,
 469,
 4,
 22,
 71,
 87,
 12,
 16,
 43,
 530,
 38,
 76,
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

```
 15,
 13,
 1247,
 4,
 22,
 17,
 515,
 17,
 12,
 16,
 626,
 18,
 2,
 5,
 62,
 386,
 12,
 8,
 316,
 8,
 106,
 5,
 4,
 2223,
 5244,
 16,
 480,
 66,
 3785,
 33,
 4,
 130,
 12,
 16,
 38,
 619,
 5,
 25,
 124,
 51,
 36,
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

```
 135,
 48,
 25,
 1415,
 33,
 6,
 22,
 12,
 215,
 28,
 77,
 52,
 5,
 14,
 407,
 16,
 82,
 2,
 8,
 4,
 107,
 117,
 5952,
 15,
 256,
 4,
 2,
 7,
 3766,
 5,
 723,
 36,
 71,
 43,
 530,
 476,
 26,
 400,
 317,
 46,
 7,
```

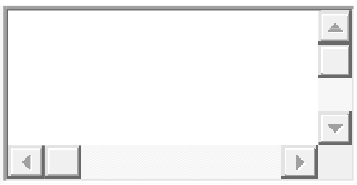
DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

```
 4,
 2,
 1029,
 13,
 104,
 88,
 4,
 381,
 15,
 297,
 98,
 32,
 2071,
 56,
 26,
 141,
 6,
 194,
 7486,
 18,
 4,
 226,
 22,
 21,
 134,
 476,
 26,
 480,
 5,
 144,
 30,
 5535,
 18,
 51,
 36,
 28,
 224,
 92,
 25,
 104,
 4,
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

```
 226,
 65,
 16,
 38,
 1334,
 88,
 12,
 16,
 283,
 5,
 16,
 4472,
 113,
 103,
 32,
 15,
 16,
 5345,
 19,
 178,
 32]
```

In [4]:



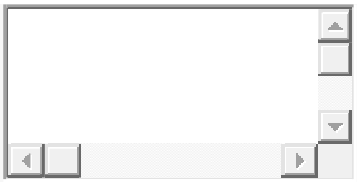
```
train_labels[0]
```

Out[4]:

```
1
```

Since we restricted ourselves to the top 10,000 most frequent words, no word index will exceed 10,000:

In [5]:



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

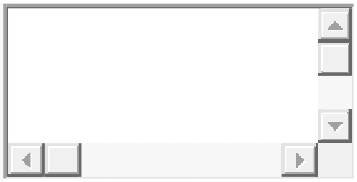
DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

Out[5]:

```
9999
```

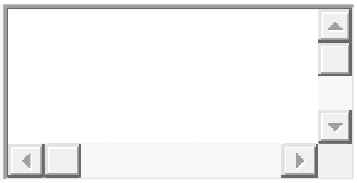
For kicks, here's how you can quickly decode one of these reviews back to English words:

In [6]:



```
# word_index is a dictionary mapping words to an integer index
word_index = imdb.get_word_index()
# We reverse it, mapping integer indices to words
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
# We decode the review; note that our indices were offset by 3
# because 0, 1 and 2 are reserved indices for "padding", "start of sequence", and "unknown".
decoded_review = ' '.join([reverse_word_index.get(i - 3, '?') for i in train_data[0]])
```

In [7]:



```
decoded_review
```

Out[7]:

```
"? this film was just brilliant casting location scenery story direction ever
yone's really suited the part they played and you could just imagine being th
ere robert ? is an amazing actor and now the same being director ? father cam
e from the same scottish island as myself so i loved the fact there was a rea
l connection with this film the witty remarks throughout the film were great
it was just brilliant so much that i bought the film as soon as it was releas
ed for ? and would recommend it to everyone to watch and the fly fishing was
amazing really cried at the end it was so sad and you know what they say if y
ou cry at a film it must have been good and this definitely was also ? to the
 two little boy's that played the ? of norman and paul they were just brillia
nt children are often left out of the ? list i think because the stars that p
lay them all grown up are such a big profile for the whole film but these chi
ldren are amazing and should be praised for what they have done don't you thi
nk the whole story was so lovely because it was true and was someone's life a
fter all that was shared with us all"
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

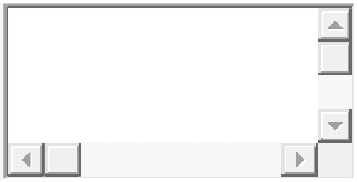
Preparing the data

We cannot feed lists of integers into a neural network. We have to turn our lists into tensors. There are two ways we could do that:

- We could pad our lists so that they all have the same length, and turn them into an integer tensor of shape `(samples, word_indices)`, then use as first layer in our network a layer capable of handling such integer tensors (the `Embedding` layer, which we will cover in detail later in the book).
- We could one-hot-encode our lists to turn them into vectors of 0s and 1s. Concretely, this would mean for instance turning the sequence `[3, 5]` into a 10,000-dimensional vector that would be all-zeros except for indices 3 and 5, which would be ones. Then we could use as first layer in our network a `Dense` layer, capable of handling floating point vector data.

We will go with the latter solution. Let's vectorize our data, which we will do manually for maximum clarity:

In [8]:

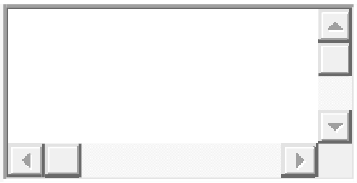


```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    # Create an all-zero matrix of shape (len(sequences), dimension)
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.  # set specific indices of results[i] to 1s
    return results

# Our vectorized training data
x_train = vectorize_sequences(train_data)
# Our vectorized test data
x_test = vectorize_sequences(test_data)
```

Here's what our samples look like now:

In [9]:

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023



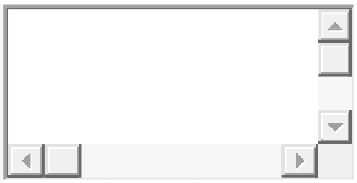
x_train[0]

Out[9]:

```
array([0., 1., 1., ..., 0., 0., 0.])
```

We should also vectorize our labels, which is straightforward:

In [10]:



```
# Our vectorized labels
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

Now our data is ready to be fed into a neural network.

Building our network

Our input data is simply vectors, and our labels are scalars (1s and 0s): this is the easiest setup you will ever encounter. A type of network that performs well on such a problem would be a simple stack of fully-connected (`Dense`) layers with `relu` activations: `Dense(16, activation='relu')`

The argument being passed to each `Dense` layer (16) is the number of "hidden units" of the layer. What's a hidden unit? It's a dimension in the representation space of the layer. You may remember from the previous chapter that each such `Dense` layer with a `relu` activation implements the following chain of tensor operations:

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

Having 16 hidden units means that the weight matrix `W` will have shape `(input_dimension, 16)`, i.e. the dot product with `W` will project the input data onto a 16-dimensional representation space (and then we would add the bias vector `b` and apply the `relu` operation). You can intuitively understand the dimensionality of your representation space as "how much freedom you are allowing the network to have when learning internal representations". Having more hidden units (a higher-dimensional representation space) allows your network to learn more complex representations, but it makes your network more computationally expensive and may

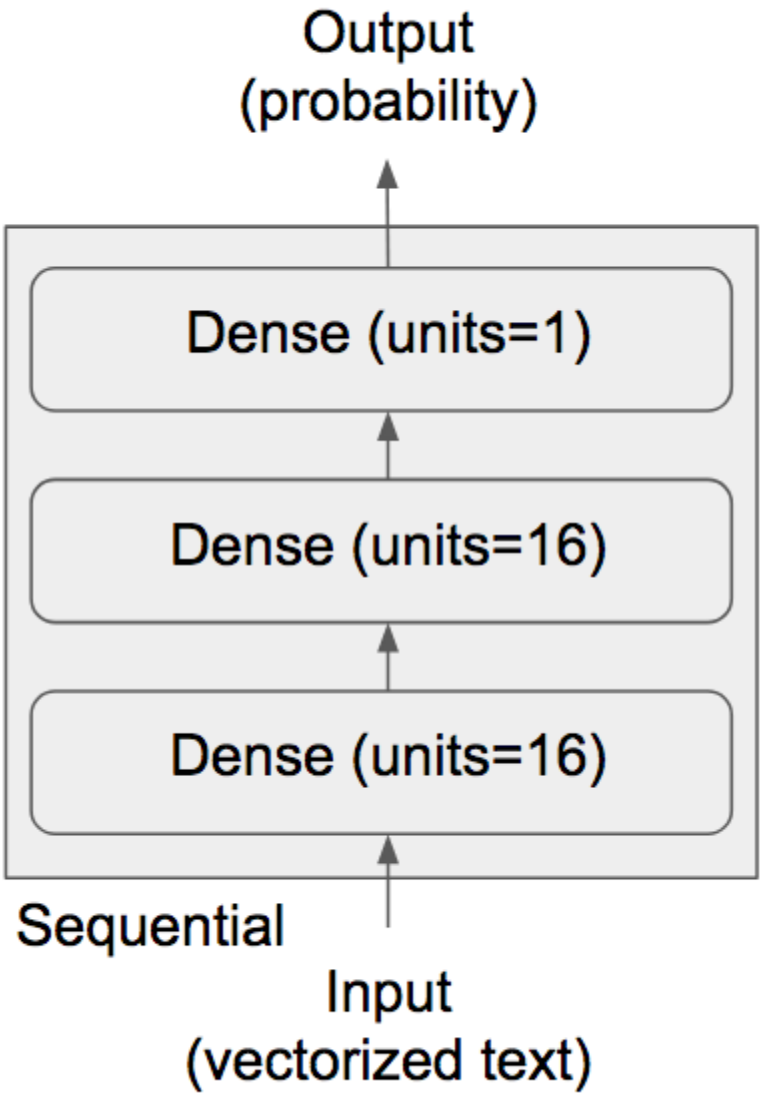
lead to learning unwanted patterns (patterns that will improve performance on the training data but not on the test data).

There are two key architecture decisions to be made about such stack of dense layers:

- How many layers to use.
- How many "hidden units" to chose for each layer.

In the next chapter, you will learn formal principles to guide you in making these choices. For the time being, you will have to trust us with the following architecture choice: two intermediate layers with 16 hidden units each, and a third layer which will output the scalar prediction regarding the sentiment of the current review. The intermediate layers will use `relu` as their "activation function", and the final layer will use a sigmoid activation so as to output a probability (a score between 0 and 1, indicating how likely the sample is to have the target "1", i.e. how likely the review is to be positive). A `relu` (rectified linear unit) is a function meant to zero-out negative values, while a sigmoid "squashes" arbitrary values into the `[0, 1]` interval, thus outputting something that can be interpreted as a probability.
Here's what our network looks like:

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023



And here's the Keras implementation, very similar to the MNIST example you saw previously:

In [11]:

```
from keras import models
```

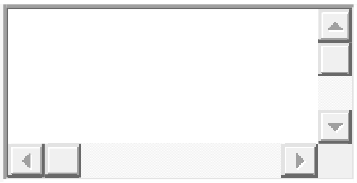
DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

```
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Lastly, we need to pick a loss function and an optimizer. Since we are facing a binary classification problem and the output of our network is a probability (we end our network with a single-unit layer with a sigmoid activation), is it best to use the `binary_crossentropy` loss. It isn't the only viable choice: you could use, for instance, `mean_squared_error`. But crossentropy is usually the best choice when you are dealing with models that output probabilities. Crossentropy is a quantity from the field of Information Theory, that measures the "distance" between probability distributions, or in our case, between the ground-truth distribution and our predictions.

Here's the step where we configure our model with the `rmsprop` optimizer and the `binary_crossentropy` loss function. Note that we will also monitor accuracy during training.

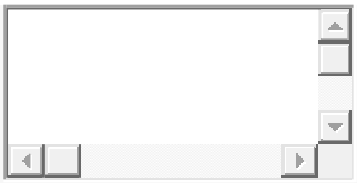
In [12]:



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

We are passing our optimizer, loss function and metrics as strings, which is possible because `rmsprop`, `binary_crossentropy` and `accuracy` are packaged as part of Keras. Sometimes you may want to configure the parameters of your optimizer, or pass a custom loss function or metric function. This former can be done by passing an optimizer class instance as the `optimizer` argument:

In [13]:

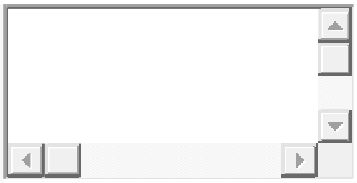


```
from keras import optimizers

model.compile(optimizer=optimizers.RMSprop(learning_rate=0.001),
```

```
        loss='binary_crossentropy',
        metrics=['accuracy'])
```

The latter can be done by passing function objects as the `loss` or `metrics` arguments:

In [15]:

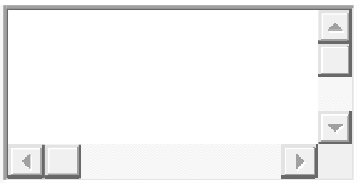


```
from keras import losses
from keras import metrics

model.compile(optimizer=optimizers.RMSprop(learning_rate=0.001),
        loss=losses.binary_crossentropy,
        metrics=[metrics.binary_accuracy])
```

Validating our approach

In order to monitor during training the accuracy of the model on data that it has never seen before, we will create a "validation set" by setting apart 10,000 samples from the original training data:

In [16]:

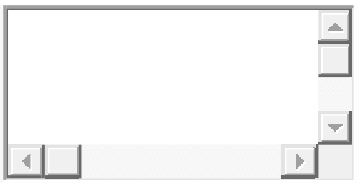


```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]

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

We will now train our model for 20 epochs (20 iterations over all samples in the `x_train` and `y_train` tensors), in mini-batches of 512 samples. At this same time we will monitor loss and accuracy on the 10,000 samples that we set apart. This is done by passing the validation data as the `validation_data` argument:

In [17]:

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023



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

```
Epoch 1/20
30/30 [==============================] - 3s 86ms/step - loss: 0.5195 - binary
_accuracy: 0.7813 - val_loss: 0.3866 - val_binary_accuracy: 0.8682
Epoch 2/20
30/30 [==============================] - 0s 14ms/step - loss: 0.3093 - binary
_accuracy: 0.8989 - val_loss: 0.3020 - val_binary_accuracy: 0.8900
Epoch 3/20
30/30 [==============================] - 0s 13ms/step - loss: 0.2227 - binary
_accuracy: 0.9261 - val_loss: 0.2759 - val_binary_accuracy: 0.8914
Epoch 4/20
30/30 [==============================] - 0s 11ms/step - loss: 0.1766 - binary
_accuracy: 0.9422 - val_loss: 0.2742 - val_binary_accuracy: 0.8900
Epoch 5/20
30/30 [==============================] - 0s 11ms/step - loss: 0.1416 - binary
_accuracy: 0.9557 - val_loss: 0.2918 - val_binary_accuracy: 0.8843
Epoch 6/20
30/30 [==============================] - 0s 11ms/step - loss: 0.1177 - binary
_accuracy: 0.9626 - val_loss: 0.2919 - val_binary_accuracy: 0.8870
Epoch 7/20
30/30 [==============================] - 0s 11ms/step - loss: 0.0979 - binary
_accuracy: 0.9697 - val_loss: 0.3297 - val_binary_accuracy: 0.8766
Epoch 8/20
30/30 [==============================] - 0s 11ms/step - loss: 0.0801 - binary
_accuracy: 0.9769 - val_loss: 0.3377 - val_binary_accuracy: 0.8773
Epoch 9/20
30/30 [==============================] - 0s 11ms/step - loss: 0.0677 - binary
_accuracy: 0.9811 - val_loss: 0.3529 - val_binary_accuracy: 0.8774
Epoch 10/20
30/30 [==============================] - 0s 12ms/step - loss: 0.0546 - binary
_accuracy: 0.9855 - val_loss: 0.3791 - val_binary_accuracy: 0.8765
Epoch 11/20
```

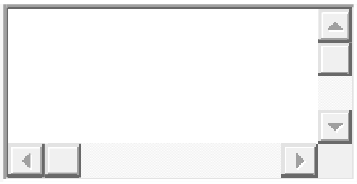
DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

```
30/30 [==============================] - 0s 12ms/step - loss: 0.0454 - binary
_accuracy: 0.9879 - val_loss: 0.4045 - val_binary_accuracy: 0.8761
Epoch 12/20
30/30 [==============================] - 0s 13ms/step - loss: 0.0346 - binary
_accuracy: 0.9933 - val_loss: 0.4312 - val_binary_accuracy: 0.8758
Epoch 13/20
30/30 [==============================] - 0s 12ms/step - loss: 0.0286 - binary
_accuracy: 0.9944 - val_loss: 0.4663 - val_binary_accuracy: 0.8724
Epoch 14/20
30/30 [==============================] - 0s 11ms/step - loss: 0.0231 - binary
_accuracy: 0.9954 - val_loss: 0.5000 - val_binary_accuracy: 0.8734
Epoch 15/20
30/30 [==============================] - 0s 12ms/step - loss: 0.0171 - binary
_accuracy: 0.9981 - val_loss: 0.5315 - val_binary_accuracy: 0.8723
Epoch 16/20
30/30 [==============================] - 0s 12ms/step - loss: 0.0153 - binary
_accuracy: 0.9973 - val_loss: 0.6289 - val_binary_accuracy: 0.8573
Epoch 17/20
30/30 [==============================] - 0s 12ms/step - loss: 0.0094 - binary
_accuracy: 0.9991 - val_loss: 0.6077 - val_binary_accuracy: 0.8708
Epoch 18/20
30/30 [==============================] - 0s 12ms/step - loss: 0.0108 - binary
_accuracy: 0.9977 - val_loss: 0.6350 - val_binary_accuracy: 0.8689
Epoch 19/20
30/30 [==============================] - 0s 11ms/step - loss: 0.0049 - binary
_accuracy: 0.9997 - val_loss: 0.6779 - val_binary_accuracy: 0.8659
Epoch 20/20
30/30 [==============================] - 0s 11ms/step - loss: 0.0103 - binary
_accuracy: 0.9972 - val_loss: 0.7144 - val_binary_accuracy: 0.8657
```

On CPU, this will take less than two seconds per epoch -- training is over in 20 seconds. At the end of every epoch, there is a slight pause as the model computes its loss and accuracy on the 10,000 samples of the validation data.

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

In [18]:



DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

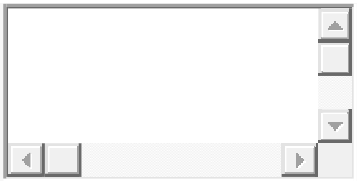
```
history_dict = history.history
history_dict.keys()
```

Out[18]:

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

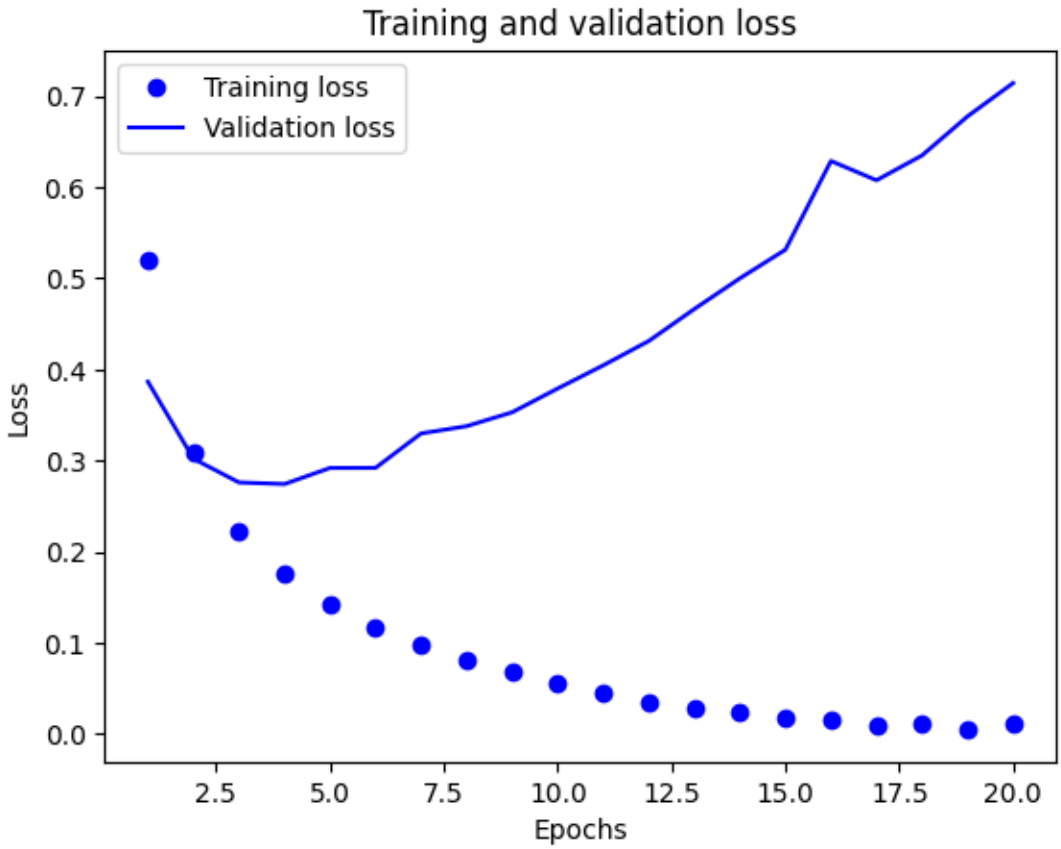
It contains 4 entries: one per metric that was being monitored, during training and during validation. Let's use Matplotlib to plot the training and validation loss side by side, as well as the training and validation accuracy:

In [19]:



```
import matplotlib.pyplot as plt

acc = history.history['binary_accuracy']
val_acc = history.history['val_binary_accuracy']
loss = history.history['loss']
val_loss = history.history['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()
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

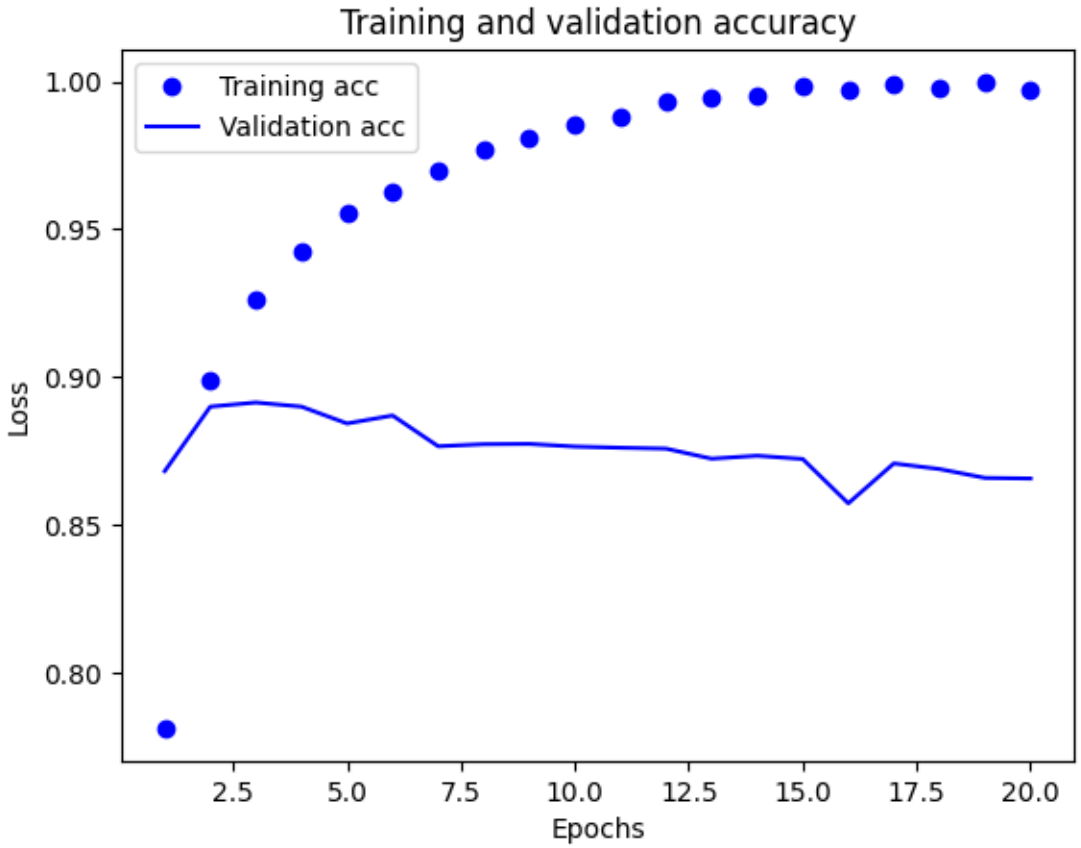


In [20]:

```
plt.clf()   # clear figure
acc_values = history_dict['binary_accuracy']
val_acc_values = history_dict['val_binary_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()
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

plt.show()



The dots are the training loss and accuracy, while the solid lines are the validation loss and accuracy. Note that your own results may vary slightly due to a different random initialization of your network.

As you can see, the training loss decreases with every epoch and the training accuracy increases with every epoch. That's what you would expect when running gradient descent optimization -- the quantity you are trying to minimize should get lower with every iteration. But that isn't the case for the validation loss and accuracy: they seem to peak at the fourth epoch. This is an example of what we were warning against earlier: a model that performs better on the training data isn't necessarily a model that will do better on data it has never seen before. In precise terms, what you are seeing is "overfitting": after the second epoch, we are over-optimizing on the training data, and we ended up learning representations that are specific to the training data and do not generalize to data outside of the training set.

In this case, to prevent overfitting, we could simply stop training after three epochs. In general, there is a range of techniques you can leverage to mitigate overfitting, which we will cover in the next chapter.

Let's train a new network from scratch for four epochs, then evaluate it on our test data:

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

In [21]:

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(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: 0.4611 - accurac
y: 0.8173
Epoch 2/4
49/49 [==============================] - 0s 7ms/step - loss: 0.2714 - accurac
y: 0.9053
Epoch 3/4
49/49 [==============================] - 0s 8ms/step - loss: 0.2145 - accurac
y: 0.9227
Epoch 4/4
49/49 [==============================] - 0s 7ms/step - loss: 0.1815 - accurac
y: 0.9348
782/782 [==============================] - 1s 1ms/step - loss: 0.3001 - accur
acy: 0.8797
```

In [22]:

```
results
```

Out[22]:

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

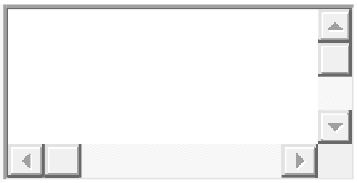
```
[0.30006352066993713, 0.8797199726104736]
```

Our fairly naive approach achieves an accuracy of 88%. With state-of-the-art approaches, one should be able to get close to 95%.

Using a trained network to generate predictions on new data

After having trained a network, you will want to use it in a practical setting. You can generate the likelihood of reviews being positive by using the `predict` method:

In [23]:



model.predict(x_test)

```
782/782 [==============================] - 2s 879us/step
```

Out[23]:

```
array([[0.15165554],
       [0.9995096 ],
       [0.54454327],
       ...,
       [0.07699458],
       [0.05212893],
       [0.41890237]], dtype=float32)
```

As you can see, the network is very confident for some samples (0.99 or more, or 0.01 or less) but less confident for others (0.6, 0.4).

Further experiments

- We were using 2 hidden layers. Try to use 1 or 3 hidden layers and see how it affects validation and test accuracy.
- Try to use layers with more hidden units or less hidden units: 32 units, 64 units...
- Try to use the `mse` loss function instead of `binary_crossentropy`.
- Try to use the `tanh` activation (an activation that was popular in the early days of neural networks) instead of `relu`.

These experiments will help convince you that the architecture choices we have made are all fairly reasonable, although they can still be improved!

Conclusions

Here's what you should take away from this example:

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

- There's usually quite a bit of preprocessing you need to do on your raw data in order to be able to feed it -- as tensors -- into a neural network. In the case of sequences of words, they can be encoded as binary vectors -- but there are other encoding options too.
- Stacks of `Dense` layers with `relu` activations can solve a wide range of problems (including sentiment classification), and you will likely use them frequently.
- In a binary classification problem (two output classes), your network should end with a `Dense` layer with 1 unit and a `sigmoid` activation, i.e. the output of your network should be a scalar between 0 and 1, encoding a probability.
- With such a scalar sigmoid output, on a binary classification problem, the loss function you should use is `binary_crossentropy`.
- The `rmsprop` optimizer is generally a good enough choice of optimizer, whatever your problem. That's one less thing for you to worry about.
- As they get better on their training data, neural networks eventually start *overfitting* and end up obtaining increasingly worse results on data never-seen-before. Make sure to always monitor performance on data that is outside of the training set.

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

Assignment 5.2 Code and Outputs:

Assignment 5.2

DSC650-T302 Big Data (2235-1)

Jake Meyer

04/13/2023

In [1]:

```
import keras
keras.__version__
```

Out[1]:

```
'2.11.0'
```

Classifying newswires: a multi-class classification example

This notebook contains the code samples found in Chapter 3, Section 5 of Deep Learning with Python. Note that the original text features far more content, in particular further explanations and figures: in this notebook, you will only find source code and related comments.

In the previous section we saw how to classify vector inputs into two mutually exclusive classes using a densely-connected neural network. But what happens when you have more than two classes?

In this section, we will build a network to classify Reuters newswires into 46 different mutually-exclusive topics. Since we have many classes, this problem is an instance of "multi-class classification", and since each data point should be classified into only one category, the problem is more specifically an instance of "single-label, multi-class classification". If each data point could have

belonged to multiple categories (in our case, topics) then we would be facing a "multi-label, multi-class classification" problem.

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:

In [2]:

```
from keras.datasets import reuters

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

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:

In [3]:

```
len(train_data)
```

Out[3]:

```
8982
```

In [4]:

```
len(test_data)
```

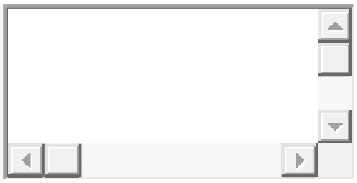
DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

Out[4]:

```
2246
```

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

In [5]:



train_data[10]

Out[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,
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

```
 4329,
 17,
 12]
```

Here's how you can decode it back to words, in case you are curious:

In [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]])
```

In [7]:

```
decoded_newswire
```

Out[7]:

```
'? ? ? said as a result of its december acquisition of space co it expects ea
rnings per share in 1987 of 1 15 to 1 30 dlrs per share up from 70 cts in 198
6 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 ml
n dlrs it said cash flow per share this year should be 2 50 to three dlrs reu
ter 3'
```

The label associated with an example is an integer between 0 and 45: a topic index.

In [8]:

```
train_labels[10]
```

Out[8]:

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

3

Preparing the data

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

In [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)
```

To vectorize the labels, there are two possibilities: we could just cast the label list as an integer tensor, or we could use a "one-hot" encoding. One-hot encoding is a widely used format for categorical data, also called "categorical encoding". For a more detailed explanation of one-hot encoding, you can refer to Chapter 6, Section 1. In our case, 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.:

In [10]:

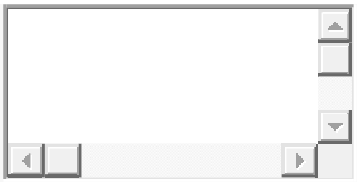
```
def to_one_hot(labels, dimension=46):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results

# Our vectorized training labels
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

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

Note that there is a built-in way to do this in Keras, which you have already seen in action in our MNIST example:

In [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)
```

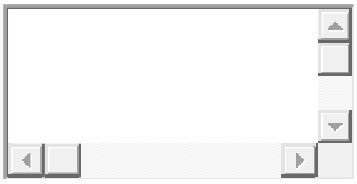
Building our network

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:

In [12]:



```
from keras import models
from keras import layers

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

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

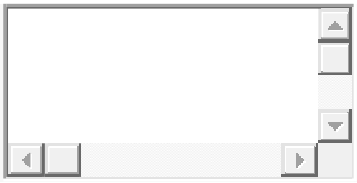
```
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
```

There are two other things you should note about this architecture:

- We are ending the network with a `Dense` layer of size 46. This means that for each input sample, our network will output a 46-dimensional vector. Each entry in this vector (each dimension) will encode a different output class.
- The last layer uses a `softmax` activation. You have already seen this pattern in the MNIST example. It means that the network will output a *probability distribution* over the 46 different output classes, i.e. for every input sample, the network will produce a 46-dimensional output vector where `output[i]` is the probability that the sample belongs to class `i`. The 46 scores will sum to 1.

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.

In [13]:

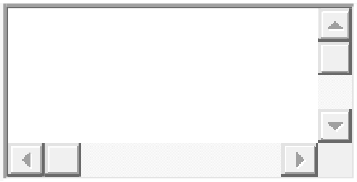


```
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Validating our approach

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

In [14]:

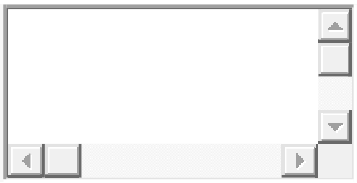


```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]

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

Now let's train our network for 20 epochs:

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

In [15]:



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

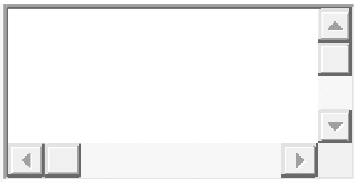
```
Epoch 1/20
16/16 [==============================] - 1s 26ms/step - loss: 2.8263 - accura
cy: 0.5045 - val_loss: 1.9650 - val_accuracy: 0.5970
Epoch 2/20
16/16 [==============================] - 0s 15ms/step - loss: 1.6263 - accura
cy: 0.6759 - val_loss: 1.4209 - val_accuracy: 0.6970
Epoch 3/20
16/16 [==============================] - 0s 16ms/step - loss: 1.2095 - accura
cy: 0.7483 - val_loss: 1.1918 - val_accuracy: 0.7400
Epoch 4/20
16/16 [==============================] - 0s 16ms/step - loss: 0.9847 - accura
cy: 0.7910 - val_loss: 1.1432 - val_accuracy: 0.7430
Epoch 5/20
16/16 [==============================] - 0s 15ms/step - loss: 0.8195 - accura
cy: 0.8260 - val_loss: 1.0054 - val_accuracy: 0.7990
Epoch 6/20
16/16 [==============================] - 0s 15ms/step - loss: 0.6859 - accura
cy: 0.8579 - val_loss: 0.9592 - val_accuracy: 0.7940
Epoch 7/20
16/16 [==============================] - 0s 16ms/step - loss: 0.5777 - accura
cy: 0.8809 - val_loss: 0.9251 - val_accuracy: 0.8060
Epoch 8/20
16/16 [==============================] - 0s 16ms/step - loss: 0.4865 - accura
cy: 0.9013 - val_loss: 0.9032 - val_accuracy: 0.8120
Epoch 9/20
16/16 [==============================] - 0s 16ms/step - loss: 0.4165 - accura
cy: 0.9147 - val_loss: 0.9006 - val_accuracy: 0.8220
Epoch 10/20
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

```
16/16 [==============================] - 0s 19ms/step - loss: 0.3537 - accura
cy: 0.9282 - val_loss: 0.8783 - val_accuracy: 0.8160
Epoch 11/20
16/16 [==============================] - 0s 19ms/step - loss: 0.3086 - accura
cy: 0.9349 - val_loss: 0.8843 - val_accuracy: 0.8080
Epoch 12/20
16/16 [==============================] - 0s 16ms/step - loss: 0.2721 - accura
cy: 0.9410 - val_loss: 0.8794 - val_accuracy: 0.8110
Epoch 13/20
16/16 [==============================] - 0s 18ms/step - loss: 0.2388 - accura
cy: 0.9450 - val_loss: 0.8761 - val_accuracy: 0.8200
Epoch 14/20
16/16 [==============================] - 0s 16ms/step - loss: 0.2178 - accura
cy: 0.9485 - val_loss: 0.8938 - val_accuracy: 0.8150
Epoch 15/20
16/16 [==============================] - 0s 18ms/step - loss: 0.2000 - accura
cy: 0.9503 - val_loss: 0.9337 - val_accuracy: 0.8050
Epoch 16/20
16/16 [==============================] - 0s 16ms/step - loss: 0.1789 - accura
cy: 0.9518 - val_loss: 0.9193 - val_accuracy: 0.8170
Epoch 17/20
16/16 [==============================] - 0s 16ms/step - loss: 0.1674 - accura
cy: 0.9534 - val_loss: 0.9339 - val_accuracy: 0.8030
Epoch 18/20
16/16 [==============================] - 0s 16ms/step - loss: 0.1555 - accura
cy: 0.9553 - val_loss: 0.9532 - val_accuracy: 0.8010
Epoch 19/20
16/16 [==============================] - 0s 16ms/step - loss: 0.1473 - accura
cy: 0.9545 - val_loss: 0.9515 - val_accuracy: 0.8030
Epoch 20/20
16/16 [==============================] - 0s 16ms/step - loss: 0.1405 - accura
cy: 0.9569 - val_loss: 0.9591 - val_accuracy: 0.8090
```

Let's display its loss and accuracy curves:

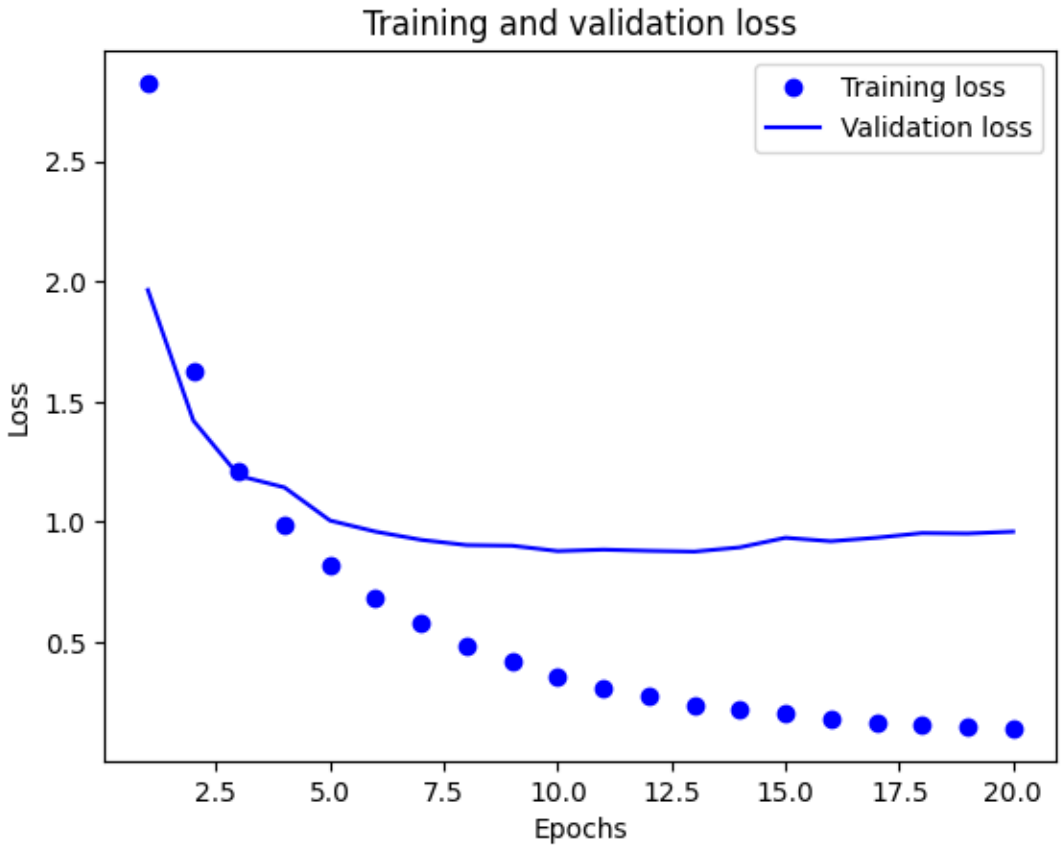
In [16]:



```
import matplotlib.pyplot as plt
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

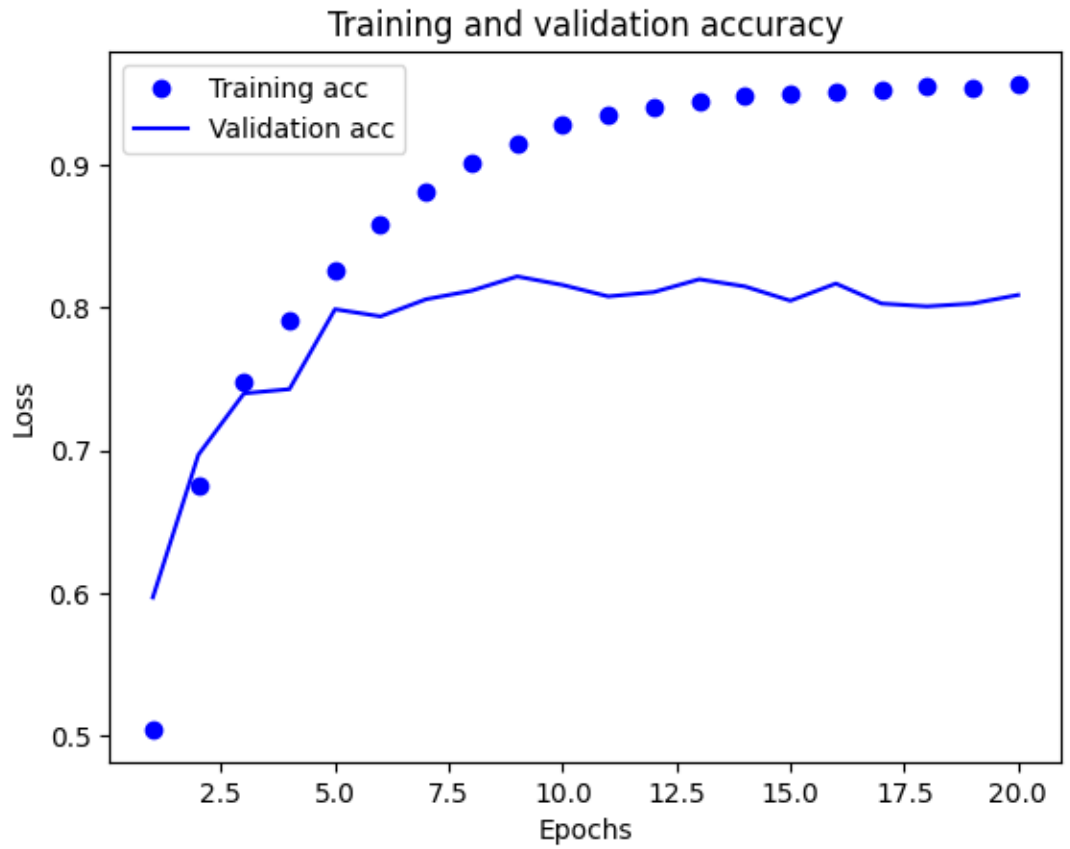
```
loss = history.history['loss']
val_loss = history.history['val_loss']

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()
```



In [18]:

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

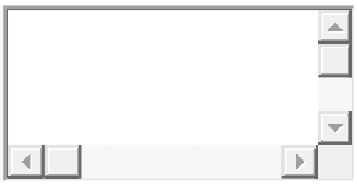
```
plt.clf()   # clear figure

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()
```



DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

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:

In [19]:



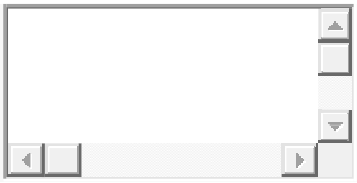
```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

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 22ms/step - loss: 2.7463 - accura
cy: 0.4635 - val_loss: 1.8614 - val_accuracy: 0.6160
Epoch 2/8
16/16 [==============================] - 0s 14ms/step - loss: 1.5567 - accura
cy: 0.6756 - val_loss: 1.3912 - val_accuracy: 0.6810
Epoch 3/8
16/16 [==============================] - 0s 15ms/step - loss: 1.1927 - accura
cy: 0.7385 - val_loss: 1.1938 - val_accuracy: 0.7410
Epoch 4/8
16/16 [==============================] - 0s 15ms/step - loss: 0.9661 - accura
cy: 0.7937 - val_loss: 1.0929 - val_accuracy: 0.7580
Epoch 5/8
16/16 [==============================] - 0s 15ms/step - loss: 0.8008 - accura
cy: 0.8284 - val_loss: 1.0207 - val_accuracy: 0.7830
Epoch 6/8
16/16 [==============================] - 0s 15ms/step - loss: 0.6646 - accura
cy: 0.8608 - val_loss: 0.9763 - val_accuracy: 0.8000
Epoch 7/8
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

```
16/16 [==============================] - 0s 14ms/step - loss: 0.5509 - accura
cy: 0.8890 - val_loss: 0.9327 - val_accuracy: 0.7980
Epoch 8/8
16/16 [==============================] - 0s 14ms/step - loss: 0.4600 - accura
cy: 0.9059 - val_loss: 0.9125 - val_accuracy: 0.8190
71/71 [==============================] - 0s 1ms/step - loss: 0.9637 - accurac
y: 0.7858
```

In [20]:



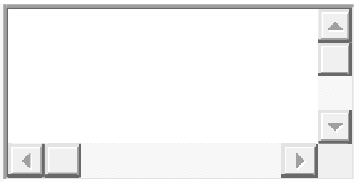
results

Out[20]:

```
[0.9636604189872742, 0.7858415246009827]
```

Our approach reaches an accuracy of ~78%. 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:

In [21]:



```
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)
```

Out[21]:

```
0.1856634016028495
```

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:

In [22]:

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

```
predictions = model.predict(x_test)
```

```
71/71 [==============================] - 0s 1ms/step
```

Each entry in `predictions` is a vector of length 46:

In [23]:

```
predictions[0].shape
```

Out[23]:

```
(46,)
```

The coefficients in this vector sum to 1:

In [24]:

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

Out[24]:

```
1.0000001
```

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

In [25]:

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

Out[25]:

```
3
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

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:

In [26]:

```
y_train = np.array(train_labels)
y_test = np.array(test_labels)
```

The only thing it would change is the choice of the loss function. Our previous loss, `categorical_crossentropy`, expects the labels to follow a categorical encoding. With integer labels, we should use `sparse_categorical_crossentropy`:

In [27]:

```
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.

In [28]:

```
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'))
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

```
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 9ms/step - loss: 3.0669 - accurac
y: 0.3909 - val_loss: 2.5516 - val_accuracy: 0.4050
Epoch 2/20
63/63 [==============================] - 0s 7ms/step - loss: 2.3675 - accurac
y: 0.3969 - val_loss: 2.2728 - val_accuracy: 0.4000
Epoch 3/20
63/63 [==============================] - 0s 7ms/step - loss: 2.0690 - accurac
y: 0.4183 - val_loss: 2.0254 - val_accuracy: 0.4000
Epoch 4/20
63/63 [==============================] - 0s 7ms/step - loss: 1.7263 - accurac
y: 0.4920 - val_loss: 1.6463 - val_accuracy: 0.5790
Epoch 5/20
63/63 [==============================] - 0s 7ms/step - loss: 1.3546 - accurac
y: 0.6379 - val_loss: 1.4197 - val_accuracy: 0.6140
Epoch 6/20
63/63 [==============================] - 0s 7ms/step - loss: 1.1729 - accurac
y: 0.6553 - val_loss: 1.3540 - val_accuracy: 0.6270
Epoch 7/20
63/63 [==============================] - 0s 7ms/step - loss: 1.0562 - accurac
y: 0.6860 - val_loss: 1.2999 - val_accuracy: 0.6550
Epoch 8/20
63/63 [==============================] - 0s 7ms/step - loss: 0.9577 - accurac
y: 0.7360 - val_loss: 1.2804 - val_accuracy: 0.6950
Epoch 9/20
63/63 [==============================] - 0s 7ms/step - loss: 0.8728 - accurac
y: 0.7811 - val_loss: 1.2712 - val_accuracy: 0.7060
Epoch 10/20
63/63 [==============================] - 0s 7ms/step - loss: 0.8011 - accurac
y: 0.7982 - val_loss: 1.2760 - val_accuracy: 0.7160
Epoch 11/20
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

```
63/63 [==============================] - 0s 7ms/step - loss: 0.7413 - accurac
y: 0.8091 - val_loss: 1.2726 - val_accuracy: 0.7260
Epoch 12/20
63/63 [==============================] - 0s 7ms/step - loss: 0.6929 - accurac
y: 0.8178 - val_loss: 1.2899 - val_accuracy: 0.7280
Epoch 13/20
63/63 [==============================] - 0s 7ms/step - loss: 0.6485 - accurac
y: 0.8301 - val_loss: 1.2958 - val_accuracy: 0.7290
Epoch 14/20
63/63 [==============================] - 0s 7ms/step - loss: 0.6100 - accurac
y: 0.8343 - val_loss: 1.3493 - val_accuracy: 0.7320
Epoch 15/20
63/63 [==============================] - 0s 7ms/step - loss: 0.5780 - accurac
y: 0.8389 - val_loss: 1.3506 - val_accuracy: 0.7260
Epoch 16/20
63/63 [==============================] - 0s 7ms/step - loss: 0.5472 - accurac
y: 0.8468 - val_loss: 1.4081 - val_accuracy: 0.7160
Epoch 17/20
63/63 [==============================] - 1s 8ms/step - loss: 0.5213 - accurac
y: 0.8555 - val_loss: 1.4643 - val_accuracy: 0.7160
Epoch 18/20
63/63 [==============================] - 1s 8ms/step - loss: 0.5012 - accurac
y: 0.8617 - val_loss: 1.4838 - val_accuracy: 0.7310
Epoch 19/20
63/63 [==============================] - 1s 8ms/step - loss: 0.4750 - accurac
y: 0.8688 - val_loss: 1.4816 - val_accuracy: 0.7190
Epoch 20/20
63/63 [==============================] - 0s 7ms/step - loss: 0.4589 - accurac
y: 0.8732 - val_loss: 1.5267 - val_accuracy: 0.7230
```

Out[28]:

```
<keras.callbacks.History at 0x2b7e8238430>
```

Our network now seems to peak at ~71% 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.

Further experiments

- Try using larger or smaller layers: 32 units, 128 units...
- We were using two hidden layers. Now try to use a single hidden layer, or three hidden layers.

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

Wrapping up

Here's what you should take away from this example:

- If you are trying to classify data points between N classes, your network should end with a `Dense` layer of size N.
- In a single-label, multi-class classification problem, your network should end with a `softmax` activation, so that it will output a probability distribution over the N output classes.
- *Categorical crossentropy* is almost always the loss function you should use for such problems. It minimizes the distance between the probability distributions output by the network, and the true distribution of the targets.
- There are two ways to handle labels in multi-class classification: ** Encoding the labels via "categorical encoding" (also known as "one-hot encoding") and using `categorical_crossentropy` as your loss function. ** Encoding the labels as integers and using the `sparse_categorical_crossentropy` loss function.
- If you need to classify data into a large number of categories, then you should avoid creating information bottlenecks in your network by having intermediate layers that are too small.

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

Assignment 5.3 Code and Output:

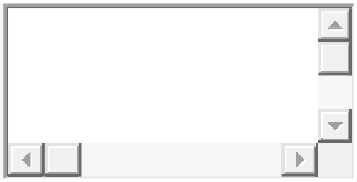
Assignment 5.3

DSC650-T302 Big Data (2235-1)

Jake Meyer

04/13/2023

In [1]:



```
import keras
keras.__version__
```

Out[1]:

```
'2.11.0'
```

Predicting house prices: a regression example

This notebook contains the code samples found in Chapter 3, Section 6 of Deep Learning with Python. Note that the original text features far more content, in particular further explanations and figures: in this notebook, you will only find source code and related comments.

In our two previous examples, we were considering classification problems, where the goal was to predict a single discrete label of an input data point. Another common type of machine learning problem is "regression", which consists of predicting a continuous value instead of a discrete label. For instance, predicting the temperature tomorrow, given meteorological data, or predicting the time that a software project will take to complete, given its specifications.

Do not mix up "regression" with the algorithm "logistic regression": confusingly, "logistic regression" is not a regression algorithm, it is a classification algorithm.

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

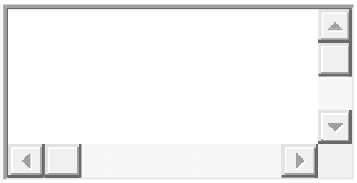
The Boston Housing Price dataset

We will be attempting to predict the median price of homes in a given Boston suburb in the mid-1970s, given a few data points about the suburb at the time, such as the crime rate, the local property tax rate, etc.

The dataset we will be using has another interesting difference from our two previous examples: it has very few data points, only 506 in total, split between 404 training samples and 102 test samples, and each "feature" in the input data (e.g. the crime rate is a feature) has a different scale. For instance some values are proportions, which take a values between 0 and 1, others take values between 1 and 12, others between 0 and 100...

Let's take a look at the data:

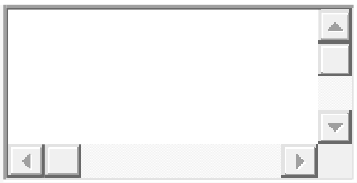
In [2]:



```
from keras.datasets import boston_housing

(train_data, train_targets), (test_data, test_targets) = boston_housing.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/boston_housing.npz
57026/57026 [==============================] - 0s 1us/step
```

In [3]:

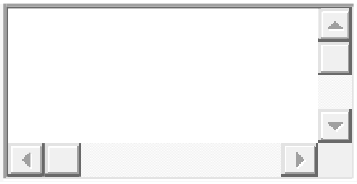


```
train_data.shape
```

Out[3]:

```
(404, 13)
```

In [4]:



DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

test_data.shape

Out[4]:

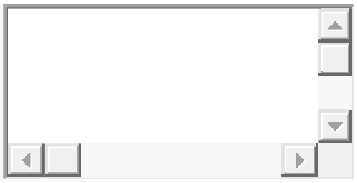
```
(102, 13)
```

As you can see, we have 404 training samples and 102 test samples. The data comprises 13 features. The 13 features in the input data are as follow:

1. Per capita crime rate.
2. Proportion of residential land zoned for lots over 25,000 square feet.
3. Proportion of non-retail business acres per town.
4. Charles River dummy variable (= 1 if tract bounds river; 0 otherwise).
5. Nitric oxides concentration (parts per 10 million).
6. Average number of rooms per dwelling.
7. Proportion of owner-occupied units built prior to 1940.
8. Weighted distances to five Boston employment centres.
9. Index of accessibility to radial highways.
10. Full-value property-tax rate per $10,000.
11. Pupil-teacher ratio by town.
12. 1000 * (Bk - 0.63) ** 2 where Bk is the proportion of Black people by town.
13. % lower status of the population.

The targets are the median values of owner-occupied homes, in thousands of dollars:

In [5]:



train_targets

Out[5]:

```
array([15.2, 42.3, 50. , 21.1, 17.7, 18.5, 11.3, 15.6, 15.6, 14.4, 12.1,
       17.9, 23.1, 19.9, 15.7,  8.8, 50. , 22.5, 24.1, 27.5, 10.9, 30.8,
       32.9, 24. , 18.5, 13.3, 22.9, 34.7, 16.6, 17.5, 22.3, 16.1, 14.9,
       23.1, 34.9, 25. , 13.9, 13.1, 20.4, 20. , 15.2, 24.7, 22.2, 16.7,
       12.7, 15.6, 18.4, 21. , 30.1, 15.1, 18.7,  9.6, 31.5, 24.8, 19.1,
       22. , 14.5, 11. , 32. , 29.4, 20.3, 24.4, 14.6, 19.5, 14.1, 14.3,
       15.6, 10.5,  6.3, 19.3, 19.3, 13.4, 36.4, 17.8, 13.5, 16.5,  8.3,
       14.3, 16. , 13.4, 28.6, 43.5, 20.2, 22. , 23. , 20.7, 12.5, 48.5,
       14.6, 13.4, 23.7, 50. , 21.7, 39.8, 38.7, 22.2, 34.9, 22.5, 31.1,
       28.7, 46. , 41.7, 21. , 26.6, 15. , 24.4, 13.3, 21.2, 11.7, 21.7,
       19.4, 50. , 22.8, 19.7, 24.7, 36.2, 14.2, 18.9, 18.3, 20.6, 24.6,
       18.2,  8.7, 44. , 10.4, 13.2, 21.2, 37. , 30.7, 22.9, 20. , 19.3,
       31.7, 32. , 23.1, 18.8, 10.9, 50. , 19.6,  5. , 14.4, 19.8, 13.8,
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

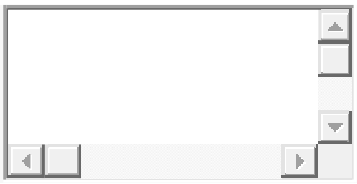
```
19.6, 23.9, 24.5, 25. , 19.9, 17.2, 24.6, 13.5, 26.6, 21.4, 11.9,
22.6, 19.6,  8.5, 23.7, 23.1, 22.4, 20.5, 23.6, 18.4, 35.2, 23.1,
27.9, 20.6, 23.7, 28. , 13.6, 27.1, 23.6, 20.6, 18.2, 21.7, 17.1,
 8.4, 25.3, 13.8, 22.2, 18.4, 20.7, 31.6, 30.5, 20.3,  8.8, 19.2,
19.4, 23.1, 23. , 14.8, 48.8, 22.6, 33.4, 21.1, 13.6, 32.2, 13.1,
23.4, 18.9, 23.9, 11.8, 23.3, 22.8, 19.6, 16.7, 13.4, 22.2, 20.4,
21.8, 26.4, 14.9, 24.1, 23.8, 12.3, 29.1, 21. , 19.5, 23.3, 23.8,
17.8, 11.5, 21.7, 19.9, 25. , 33.4, 28.5, 21.4, 24.3, 27.5, 33.1,
16.2, 23.3, 48.3, 22.9, 22.8, 13.1, 12.7, 22.6, 15. , 15.3, 10.5,
24. , 18.5, 21.7, 19.5, 33.2, 23.2,  5. , 19.1, 12.7, 22.3, 10.2,
13.9, 16.3, 17. , 20.1, 29.9, 17.2, 37.3, 45.4, 17.8, 23.2, 29. ,
22. , 18. , 17.4, 34.6, 20.1, 25. , 15.6, 24.8, 28.2, 21.2, 21.4,
23.8, 31. , 26.2, 17.4, 37.9, 17.5, 20. ,  8.3, 23.9,  8.4, 13.8,
 7.2, 11.7, 17.1, 21.6, 50. , 16.1, 20.4, 20.6, 21.4, 20.6, 36.5,
 8.5, 24.8, 10.8, 21.9, 17.3, 18.9, 36.2, 14.9, 18.2, 33.3, 21.8,
19.7, 31.6, 24.8, 19.4, 22.8,  7.5, 44.8, 16.8, 18.7, 50. , 50. ,
19.5, 20.1, 50. , 17.2, 20.8, 19.3, 41.3, 20.4, 20.5, 13.8, 16.5,
23.9, 20.6, 31.5, 23.3, 16.8, 14. , 33.8, 36.1, 12.8, 18.3, 18.7,
19.1, 29. , 30.1, 50. , 50. , 22. , 11.9, 37.6, 50. , 22.7, 20.8,
23.5, 27.9, 50. , 19.3, 23.9, 22.6, 15.2, 21.7, 19.2, 43.8, 20.3,
33.2, 19.9, 22.5, 32.7, 22. , 17.1, 19. , 15. , 16.1, 25.1, 23.7,
28.7, 37.2, 22.6, 16.4, 25. , 29.8, 22.1, 17.4, 18.1, 30.3, 17.5,
24.7, 12.6, 26.5, 28.7, 13.3, 10.4, 24.4, 23. , 20. , 17.8,  7. ,
11.8, 24.4, 13.8, 19.4, 25.2, 19.4, 19.4, 29.1])
```

The prices are typically between $10,000 and 10,000 ??? 50,000. If that sounds cheap, remember this was the mid-1970s, and these prices are not inflation-adjusted.

Preparing the data

It would be problematic to feed into a neural network values that all take wildly different ranges. The network might be able to automatically adapt to such heterogeneous data, but it would definitely make learning more difficult. A widespread best practice to deal with such data is to do feature-wise normalization: for each feature in the input data (a column in the input data matrix), we will subtract the mean of the feature and divide by the standard deviation, so that the feature will be centered around 0 and will have a unit standard deviation. This is easily done in Numpy:

In [6]:



```
mean = train_data.mean(axis=0)
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

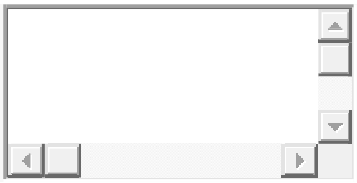
```
train_data -= mean
std = train_data.std(axis=0)
train_data /= std

test_data -= mean
test_data /= std
```

Note that the quantities that we use for normalizing the test data have been computed using the training data. We should never use in our workflow any quantity computed on the test data, even for something as simple as data normalization.

Building our network

Because so few samples are available, we will be using a very small network with two hidden layers, each with 64 units. In general, the less training data you have, the worse overfitting will be, and using a small network is one way to mitigate overfitting.

In [7]:



```
from keras import models
from keras import layers

def build_model():
  # Because we will need to instantiate
  # the same model multiple times,
  # we use a function to construct it.
  model = models.Sequential()
  model.add(layers.Dense(64, activation='relu',
                         input_shape=(train_data.shape[1],)))
  model.add(layers.Dense(64, activation='relu'))
  model.add(layers.Dense(1))
  model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
  return model
```

Our network ends with a single unit, and no activation (i.e. it will be linear layer). This is a typical setup for scalar regression (i.e. regression where we are trying to predict a single continuous value). Applying an activation function would constrain the range that the output can take; for instance if we applied a `sigmoid` activation function to our last layer, the network could only learn to predict values between 0 and 1. Here, because the last layer is purely linear, the network is free to learn to predict values in any range.

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

Note that we are compiling the network with the `mse` loss function -- Mean Squared Error, the square of the difference between the predictions and the targets, a widely used loss function for regression problems.

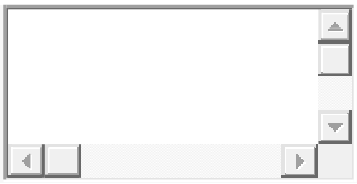
We are also monitoring a new metric during training: `mae`. This stands for Mean Absolute Error. It is simply the absolute value of the difference between the predictions and the targets. For instance, a MAE of 0.5 on this problem would mean that our predictions are off by $500 on average.

Validating our approach using K-fold validation

To evaluate our network while we keep adjusting its parameters (such as the number of epochs used for training), we could simply split the data into a training set and a validation set, as we were doing in our previous examples. However, because we have so few data points, the validation set would end up being very small (e.g. about 100 examples). A consequence is that our validation scores may change a lot depending on *which* data points we choose to use for validation and which we choose for training, i.e. the validation scores may have a high *variance* with regard to the validation split. This would prevent us from reliably evaluating our model.

The best practice in such situations is to use K-fold cross-validation. It consists of splitting the available data into K partitions (typically K=4 or 5), then instantiating K identical models, and training each one on K-1 partitions while evaluating on the remaining partition. The validation score for the model used would then be the average of the K validation scores obtained.
In terms of code, this is straightforward:

In [8]:



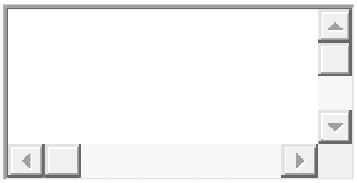
```
import numpy as np

k = 4
num_val_samples = len(train_data) // k
num_epochs = 100
all_scores = []
for i in range(k):
    print('processing fold #', i)
    # Prepare the validation data: data from partition # k
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]

    # Prepare the training data: data from all other partitions
    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

```
        train_data[(i + 1) * num_val_samples:]],
       axis=0)
    partial_train_targets = np.concatenate(
       [train_targets[:i * num_val_samples],
        train_targets[(i + 1) * num_val_samples:]],
       axis=0)

    # Build the Keras model (already compiled)
    model = build_model()
    # Train the model (in silent mode, verbose=0)
    model.fit(partial_train_data, partial_train_targets,
              epochs=num_epochs, batch_size=1, verbose=0)
    # Evaluate the model on the validation data
    val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
    all_scores.append(val_mae)
```

```
processing fold # 0
processing fold # 1
processing fold # 2
processing fold # 3
```

In [9]:

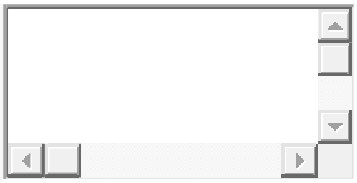


```
all_scores
```

Out[9]:

```
[2.158801794052124, 2.768878221511841, 2.6661810874938965, 2.380241394042968
8]
```

In [10]:



```
np.mean(all_scores)
```

Out[10]:

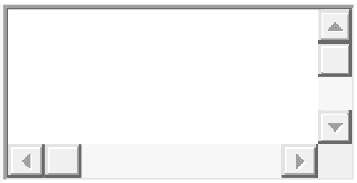
```
2.4935256242752075
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

As you can notice, the different runs do indeed show rather different validation scores, from 2.1 to 2.9. Their average (2.4) is a much more reliable metric than any single of these scores -- that's the entire point of K-fold cross-validation. In this case, we are off by $2,400 on average, which is still significant considering that the prices range from $2,400 to $10,000 to $50,000.

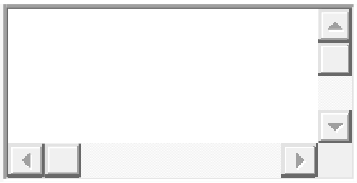
Let's try training the network for a bit longer: 500 epochs. To keep a record of how well the model did at each epoch, we will modify our training loop to save the per-epoch validation score log:

In [11]:



```
from keras import backend as K

# Some memory clean-up
K.clear_session()
```

In [14]:



```
num_epochs = 500
all_mae_histories = []
for i in range(k):
    print('processing fold #', i)
    # Prepare the validation data: data from partition # k
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]

    # Prepare the training data: data from all other partitions
    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
```

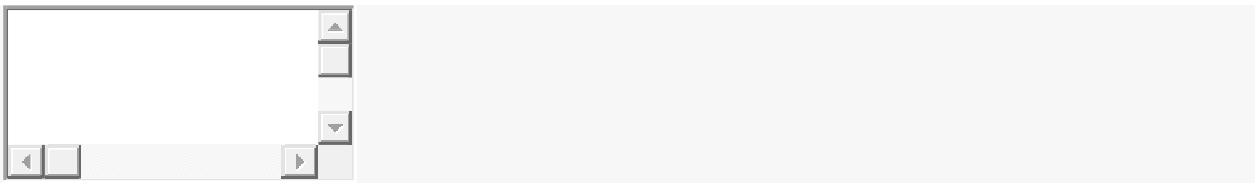
DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

```
    axis=0)

  # Build the Keras model (already compiled)
  model = build_model()
  # Train the model (in silent mode, verbose=0)
  history = model.fit(partial_train_data, partial_train_targets,
                      validation_data=(val_data, val_targets),
                      epochs=num_epochs, batch_size=1, verbose=0)
  mae_history = history.history['val_mae']
  all_mae_histories.append(mae_history)
```

```
processing fold # 0
processing fold # 1
processing fold # 2
processing fold # 3
```

We can then compute the average of the per-epoch MAE scores for all folds:

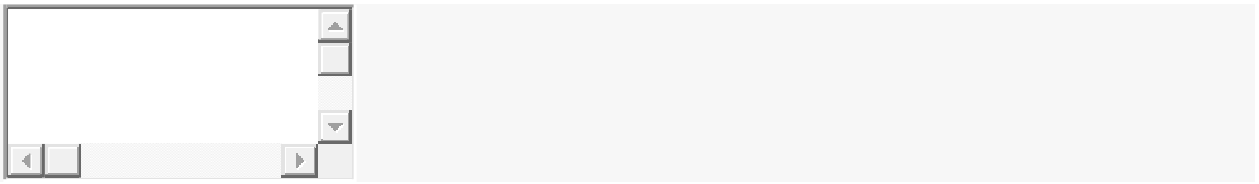
In [15]:



```
average_mae_history = [
  np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)]
```

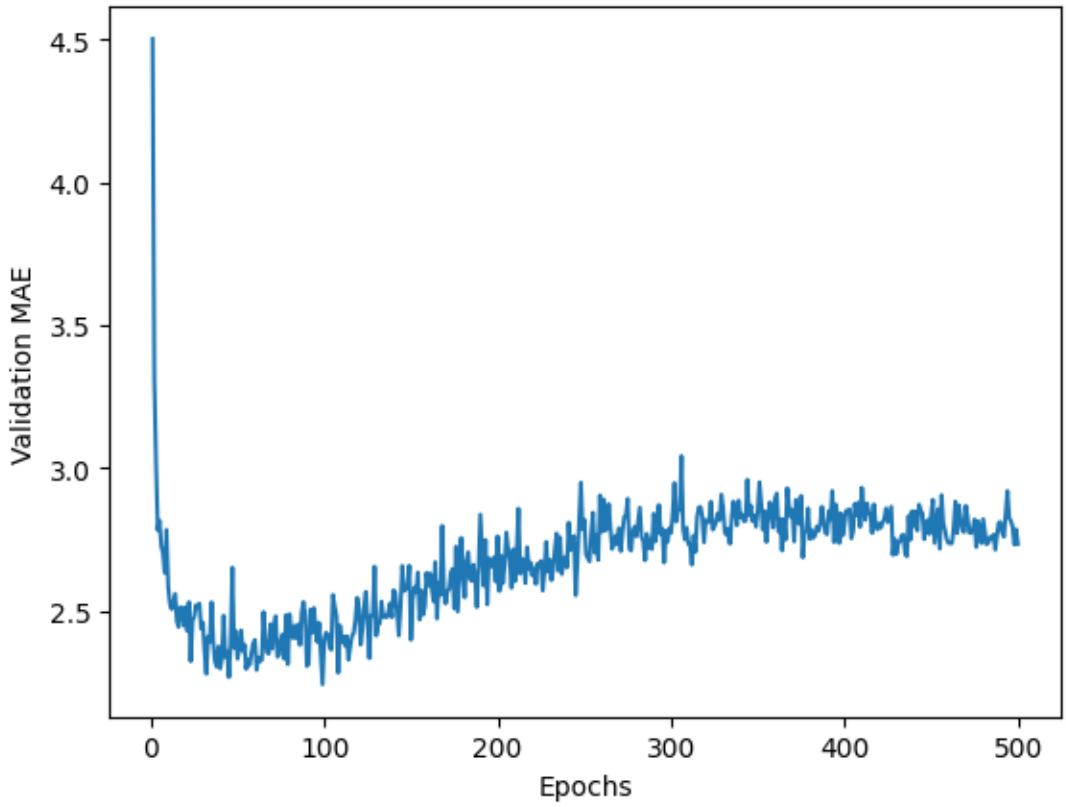
Let's plot this:

In [16]:



```
import matplotlib.pyplot as plt

plt.plot(range(1, len(average_mae_history) + 1), average_mae_history)
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023



It may be a bit hard to see the plot due to scaling issues and relatively high variance. Let's:

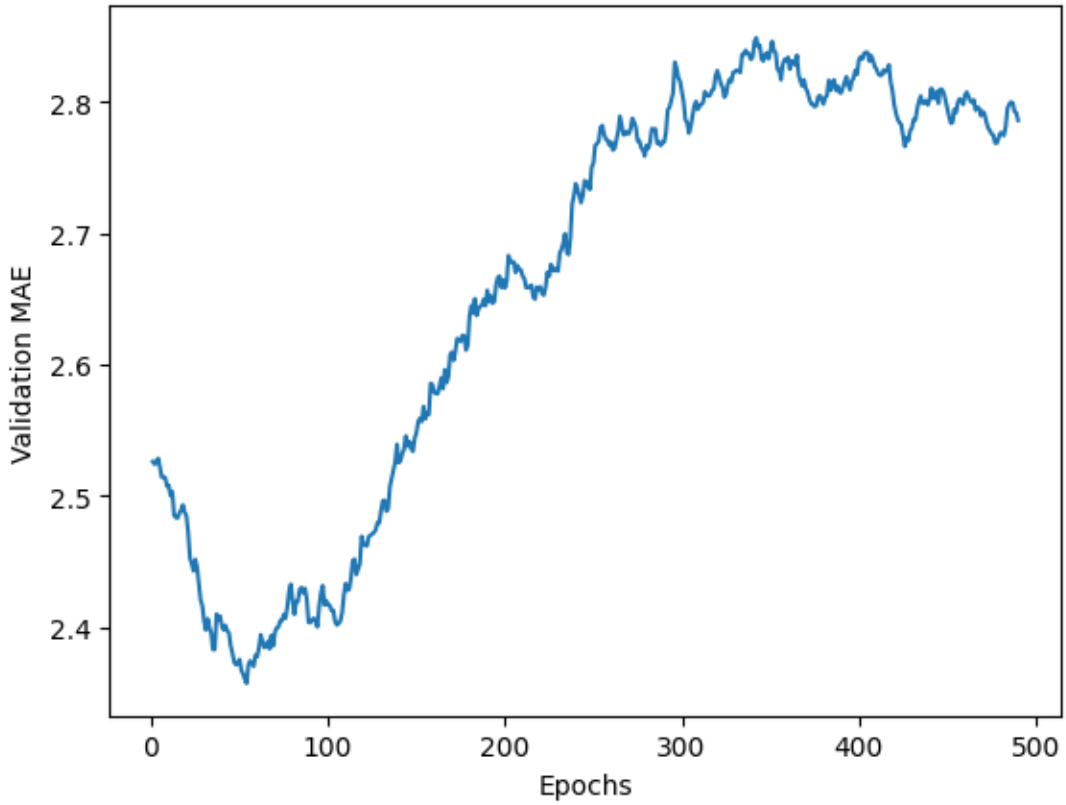
- Omit the first 10 data points, which are on a different scale from the rest of the curve.
- Replace each point with an exponential moving average of the previous points, to obtain a smooth curve.

In [17]:

```
def smooth_curve(points, factor=0.9):
  smoothed_points = []
  for point in points:
    if smoothed_points:
      previous = smoothed_points[-1]
      smoothed_points.append(previous * factor + point * (1 - factor))
    else:
      smoothed_points.append(point)
```

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

```
 return smoothed_points

smooth_mae_history = smooth_curve(average_mae_history[10:])

plt.plot(range(1, len(smooth_mae_history) + 1), smooth_mae_history)
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
```



According to this plot, it seems that validation MAE stops improving significantly after 80 epochs. Past that point, we start overfitting.

Once we are done tuning other parameters of our model (besides the number of epochs, we could also adjust the size of the hidden layers), we can train a final "production" model on all of the training data, with the best parameters, then look at its performance on the test data:

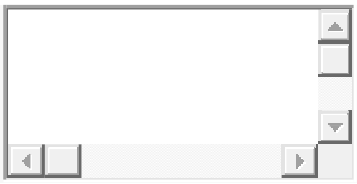
In [18]:

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

```
# Get a fresh, compiled model.
model = build_model()
# Train it on the entirety of the data.
model.fit(train_data, train_targets,
          epochs=80, batch_size=16, verbose=0)
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)
```

```
4/4 [==============================] - 0s 1ms/step - loss: 16.6546 - mae: 2.5
694
```

In [19]:



```
test_mae_score
```

Out[19]:

```
2.569382905960083
```

We are still off by about $2,550.

Wrapping up

Here's what you should take away from this example:

- Regression is done using different loss functions from classification; Mean Squared Error (MSE) is a commonly used loss function for regression.
- Similarly, evaluation metrics to be used for regression differ from those used for classification; naturally the concept of "accuracy" does not apply for regression. A common regression metric is Mean Absolute Error (MAE).
- When features in the input data have values in different ranges, each feature should be scaled independently as a preprocessing step.
- When there is little data available, using K-Fold validation is a great way to reliably evaluate a model.
- When little training data is available, it is preferable to use a small network with very few hidden layers (typically only one or two), in order to avoid severe overfitting.

This example concludes our series of three introductory practical examples. You are now able to handle common types of problems with vector data input:

- Binary (2-class) classification.
- Multi-class, single-label classification.
- Scalar regression.

DSC650-T302 Big Data (2235-1)
Professor Iranitalab
Assignment 05 Code and Outputs
Jake Meyer
04/15/2023

In the next chapter, you will acquire a more formal understanding of some of the concepts you have encountered in these first examples, such as data preprocessing, model evaluation, and overfitting.