1. Classifying movie reviews

- Two-class classification, or binary classification, is one of the most common kinds of machine learning problems. In this example, you'll learn to classify movie reviews as positive or negative, based on the text content of the reviews

1.1. The IMDB dataset

- The IMDB dataset contains 50,000 highly polarized reviews from the Internet Movie Database
  - They're split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting of 50% negative and 50% positive reviews
  - 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.
    - This enables us to focus on model building, training, and evaluation.
    - In chapter 11, you'll learn how to process raw text input from scratch

- Load the data set (when first run it, about 80MB of data will be downloaded to the machine)

Listing 4.1 Loading the IMDB dataset

```
from tensorflow.keras.datasets import imdb
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(
    num_words=10000)
```

  - The argument num_words = 10000 means you'll 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.
    - If we didn't set this limit, we'd be working with 88,585 unique words in the training data, which is unnecessarily large.
    - Many of these words only occur in a single sample, and thus can't be meaningfully used for classification
  - The variables train_data and test_data are lists of reviews; each review is a list of word indices (encoding a sequence of words). train_labels and test_labels are lists of 0s and 1s, where 0 stands for negative and 1 stands for positive

```
>>> train_data[0]
[1, 14, 22, 16, ... 178, 32]
>>> train_labels[0]
1
```

  - Since we're restricting ourselves to the top 10,000 most frequent words, no word index will exceed 10,000:

```
>>> max([max(sequence) for sequence in train_data])
9999
```

- Here's how you can quickly decode one of these reviews back to English words

Listing 4.2 Decoding reviews back to text

```
word_index = imdb.get_word_index()
reverse_word_index = dict(
    [(value, key) for (key, value) in word_index.items()])
decoded_review = " ".join(
    [reverse_word_index.get(i - 3, "?") for i in train_data[0]])
```

word_index is a dictionary mapping words to an integer index.

Reverses it, mapping integer indices to words

Decodes the review. Note that the indices are offset by 3 because 0, 1, and 2 are reserved indices for "padding," "start of sequence," and "unknown."

1.2. Preparing the data

- You can't directly feed lists of integers into a neural network. They all have different lengths, but a neural network expects to process continuous batches of data. You have to turn your lists into tensors. There are two ways to do that
  - Pad your lists so that they all have the same length, turn them into an integer tensor of shape (samples, max_length), and start your model with a layer capable of handling such integer tensors (the *Embedding* layer, which we'll cover in detail later in this book)
  - *Multi-hot encode* your lists to turn them into vectors of 0s and 1s. This would mean, for instance, turning the sequence [8, 5] into a 10,000-dimensional vector that would be all 0s except for indices 9 and 5, which would be 1s. Then you could use a *Dense* layer, capable of handling floating-point vector data, as the first layer in the model.

- Let's go with the second option

Listing 4.3 Encoding the integer sequences via multi-hot encoding

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

Creates an all-zero matrix of shape (len(sequences), dimension)

Sets specific indices of results[i] to 1s

Vectorized training data

Vectorized test data

  - Here's what the samples look like now

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

- We should also vectorize the labels

```
y_train = np.asarray(train_labels).astype("float32")
y_test = np.asarray(test_labels).astype("float32")
```

- Now the data is ready to be fed into a neural network

1.3. Building your model

- The input data is vectors, and the labels are scalars (1s and 0s). A type of model that performs well on such a problem is a plain stack of densely connected layers with relu activations
- There are two key architecture decisions to be made about such a stack of dense layers
  - How many layers to use
  - How many units to choose for each layer
- In chapter 5, you'll learn formal principles to guide you in making these choices. For the time being, you'll have to trust me with the following architecture choices
  - Two intermediate layers with 16 units each
  - A third layer that will output the scalar prediction regarding the sentiment of the current review

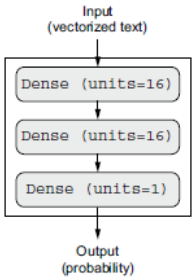


Figure 4.1 The three-layer model

- The following lists shows the Keras implementation, similar to the MNIST example you saw previously

Listing 4.4 Model definition

```
from tensorflow import keras
from tensorflow.keras import layers

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

  - Recall that, dense layer with a relu activation implements the following chain of tensor operations

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

  - Having 16 units means the weight matrix W will have shape (input_dimension, 16): the dot product with W will project the input data onto a 16-disneisonal representation space (and then you'll add the bias vector b and apply the relu operations).
    - You can intuitively understand the dimensionality of your representation space as "how much freedom you're allowing the model to have when learning internal representations."
    - Having more units allows your model to learn more-complex representations, but it makes the model 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).
  - The intermediate layers use relu as their activation function, and the final layer uses a sigmoid activation so as to output a probability (how likely the sample is to have the target "1")
    - A rectified linear unit is a function meant to zero out negative values, whereas a sigmoid "squashes" arbitrary values into the [0, 1] interval, outputting something that can be interpreted as a probability

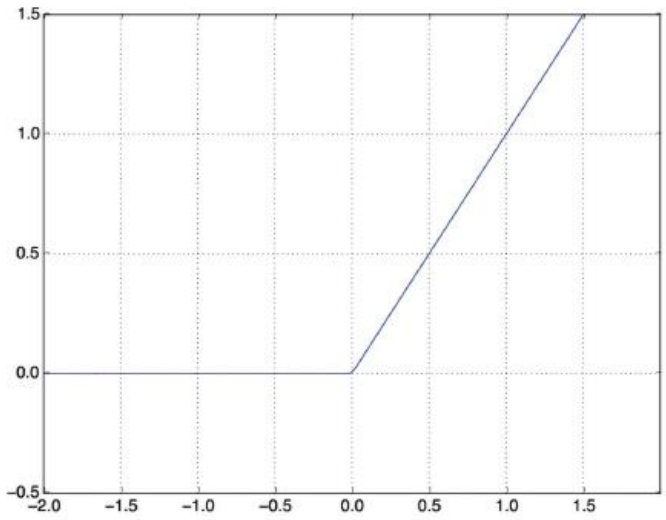


Figure 4.2 The rectified linear unit function

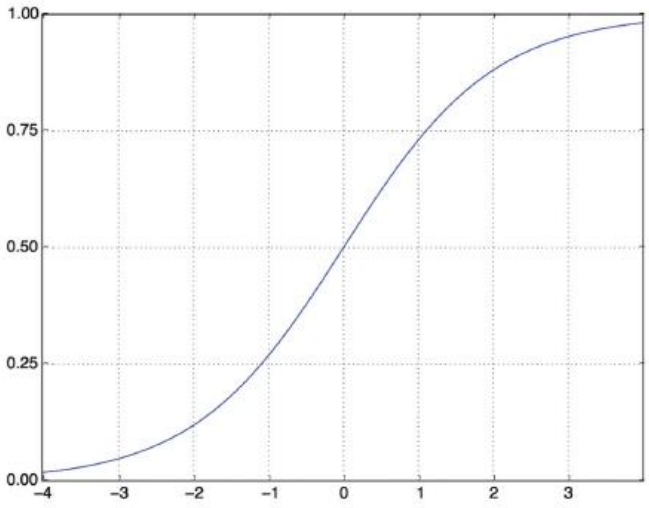


Figure 4.3 The sigmoid function

- Finally, you need to choose a loss function and an optimizer. Because you're facing a binary classification problem and the output of your model is probability, it's best to use the binary_crossentropy loss.
  - It isn't the only viable choice: for instance, you could use mean_squared_error.
  - But crossentropy is usually the best choice when you're dealing with models that output probabilities.
  - Crossentropy is a quantity from the field of information theory that measures the distance between probability distribution or, in this case, between the ground-truth distribution and your prediction.

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

- Here is how we would compile for the program

Listing 4.5 Compiling the model

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

1.4. Validating your approach

- As learned in chapter 3, a deep learning model should never be evaluated on its training data – it's standard practice to use a validation set to monitor the accuracy of the model during training.
  - Here, we'll create a validation set by setting apart 10,000 samples from the original training data.

Listing 4.6 Setting aside a validation set

```
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 the model for 20 epochs (2o iterations over all samples in the training data) in mini-batches of 512 samples. At the same time, we will monitor loss and accuracy on the 10,000 samples that we set apart. We do so by passing the validation data as the validation_data argument.

Listing 4.7 Training your model

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

  - On CPU, this will take less than 2 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.

```
>>> history_dict = history.history
>>> history_dict.keys()
[u"accuracy", u"loss", u"val_accuracy", u"val_loss"]
```

    - The dictionary contains four entries: one er metric that was being monitored during training and during validation.
    - In the following two listings, let's use Matplotlib to plot the training and validation loss side by side, and training and validation accuracy.
      - Note that your own results may vary slightly due to a different random initialization of your model

Listing 4.8 Plotting the training and validation loss

```
import matplotlib.pyplot as plt
history_dict = history.history
loss_values = history_dict["loss"]
val_loss_values = history_dict["val_loss"]
epochs = range(1, len(loss_values) + 1)
plt.plot(epochs, loss_values, "bo", label="Training loss")
plt.plot(epochs, val_loss_values, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

"bo" is for "blue dot."

"b" is for "solid blue line."

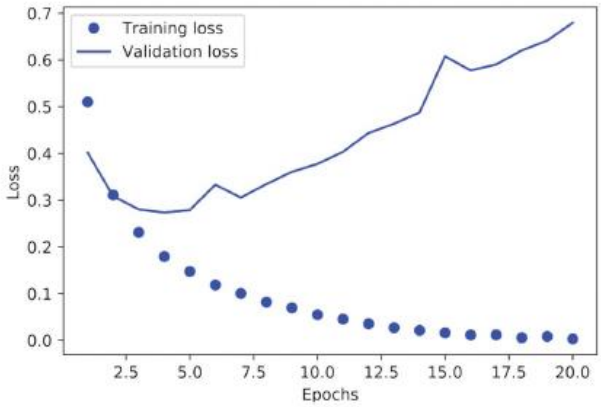


Figure 4.4 Training and validation loss

Listing 4.9 Plotting the training and validation accuracy

```
plt.clf()
acc = history_dict["accuracy"]
val_acc = history_dict["val_accuracy"]
plt.plot(epochs, acc, "bo", label="Training acc")
plt.plot(epochs, val_acc, "b", label="Validation acc")
plt.title("Training and validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```

Clears the figure

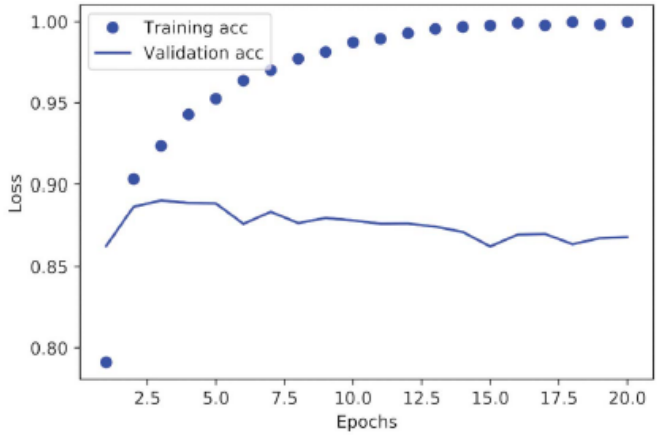


Figure 4.5 Training and validation accuracy

- 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're trying to minimize should be less with every iteration
- But validation loss and accuracy seem to peak at the fourth epoch.

- ✧ This is an example of what we warned 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're seeing is *overfitting*: overoptimizing on the training data, end up learning representations that are specific to the training data and don't generalize to data outside of the training set.

➢ In this case, to prevent overfitting, you could stop training after four epochs.

- ✧ In general, you can use a range of techniques to mitigate overfitting, which will be covered in chapter 5

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

Listing 4.10 Retraining a model from scratch

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

    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)
```

The final results are as follows:

```
>>> results
[0.2929924130630493, 0.88327999999999995]
```

The first number, 0.29, is the test loss, and the second number, 0.88, is the test accuracy.

➢ This fairly naïve approach achieves an accuracy of 88%. With state-of-the-art approaches, you should be able to get close to 95%

1.5. Using a trained model to generate predictions on new data

- After having trained a model, you'll want to use it in a practical setting.

➢ You can generate the likelihood of reviews being positive by using the *predict* method, as you've learned in chapter 3

```
>>> model.predict(x_test)
array([[ 0.98006207]
       [ 0.99758697]
       [ 0.99975556]
       ...,
       [ 0.82167041]
       [ 0.02885115]
       [ 0.65371346]], dtype=float32)
```

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

1.6. Further experiments

- The following experiments will help convince you that the architecture choices you've made are all fairly reasonable, although there's still room for improvement
  - You used two representation layers before the final classification layer. Try using one or three representation layers, and see how doing so affects validation and test accuracy
  - Try using layers with more units or fewer units: 32 units, 64 units, and so on
  - Try using the mse loss function instead of binary_crossentropy.
  - Try using the tanh activation (an activation that was popular in the early days of neural networks) instead of relu.

1.7. Wrapping up

- Here's what you should take away from this example
  - You usually need to do quite a bit of preprocessing on your raw data in order to be able to feed it – as tensors – into a neural network
    - Sequences of words 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'll likely use them frequently
  - In a binary classification problem, your model should end with a dense layer with one unit and a sigmoid activation: the output of your model 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, 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 they've never seen before.
    - Be sure to always monitor performance on data that is outside of the training set.

2. Multiclass classification example

- In the previous section, you saw how to classify vector inputs into mutually exclusive classes using a densely connected neural network. What happens when you have more than two classes?
  - In this section, we'll build a model to classify Reuters newswires into 46 mutually exclusive topics.
  - Because we have many classes, this problem is an instance of multiclass classification, and because each data point should be classified into only one category, the problem is more specifically an instance of *single-label multiclass classification*.

✧ If each data point could belong to multiple categories (in this case, topics), we'd be facing a *multilabel multiclass classification*.

2.1. The Reuters dataset

- The *Reuters dataset* is a set of short newswires and their topics, published by Reuters in 1986.
  - ➢ It's a 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.

Listing 4.11 Loading the Reuters dataset

```
from tensorflow.keras.datasets import reuters
(train_data, train_labels), (test_data, test_labels) = reuters.load_data(
    num_words=10000)
```

  - ➢ The argument num_words = 10000 restricts the data to the 10,000 most frequently occurring words found in the data
  - ➢ You have 8,982 training examples and 2,246 test examples

```
>>> len(train_data)
8982
>>> len(test_data)
2246
```

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

```
>>> train_data[10]
[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

Listing 4.12 Decoding newswires back to text

```
word_index = reuters.get_word_index()
reverse_word_index = dict(
    [(value, key) for (key, value) in word_index.items()])
decoded_newswire = " ".join(
    [reverse_word_index.get(i - 3, "?") for i in     train_data[0]])
```

Note that the indices are offset by 3 because 0, 1, and 2 are reserved indices for "padding," "start of sequence," and "unknown."

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

```
>>> train_labels[10]
3
```

2.2. Preparing the data

- Vectorize the data with the exact same code in the previous example

Listing 4.13 Encoding the input data

```
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

Vectorized training data

Vectorized test data

- To vectorize the labels, there are two possibilities
  - Cast the label list as an integer tensor, or
  - Use *one-hot encoding*.
    - In this case, one-hot encoding of the labels consists of embedding each label as an all-zero vector with a 1 in the place of the label index.

Listing 4.14 Encoding the labels

```
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
y_train = to_one_hot(train_labels)
y_test = to_one_hot(test_labels)
```

Vectorized training labels

Vectorized test labels

- We could also use the built-in way to do this in Keras

```
from tensorflow.keras.utils import to_categorical
y_train = to_categorical(train_labels)
y_test = to_categorical(test_labels)
```

2.3. Building your model

- This topic-classification problem looks similar to the previous movie-review classification problem: in both cases, we're trying to classify short snippets of text. But there is a new constraint here: the number of output classes has gone from 2 to 46. The dimensionality of the output space is much larger
  - In a stack of dense layers, 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 the previous example, we used 16-dimensional intermediate layers, but a 16-dimensiaonl space may be too limited to learn to separate 46 different classes: such small layers may act as information bottleneck, permanently dropping relevant information.
  - For this reason, we use larger layers, let's use 64 units.

Listing 4.15 Model definition

```
model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(64, activation="relu"),
    layers.Dense(46, activation="softmax")
])
```

- There are two other things you should note about this architecture
  - For each input sample, the network will output a 46-dimensional vector due to the final dense layer with 46 neurons. Each entry in this vector will encode a different output class

- The last layer uses a *softmax* activation. It means the model will output a *probability distribution* over the 46 different output classes – for every input sample, the model 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: here, between the probability distribution output by the model and the true distribution of the labels. By minimizing the distance between these two distributions, you train the model to output something as close as possible to the true labels.

Listing 4.16 Compiling the model

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

2.4. Validating the approach

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

Listing 4.17 Setting aside a validation set

```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]
y_val = y_train[:1000]
partial_y_train = y_train[1000:]
```

- Train for 20 epochs

Listing 4.18 Training the model

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

- Display the loss and accuracy curves

Listing 4.19 Plotting the training and validation loss

```
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()
```

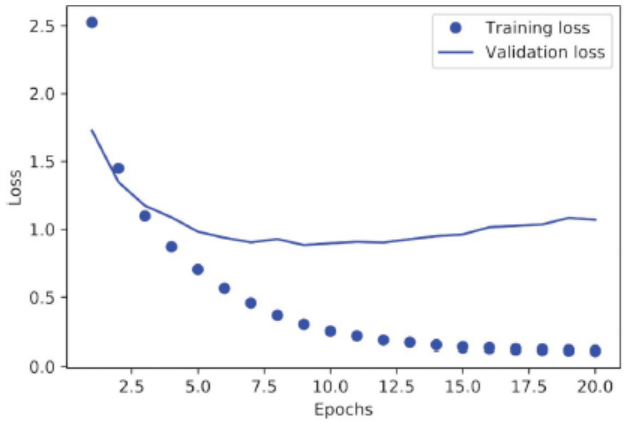


Figure 4.6 Training and validation loss

Listing 4.20 Plotting the training and validation accuracy

```
plt.clf()                    # Clears the figure
acc = history.history["accuracy"]
val_acc = history.history["val_accuracy"]
plt.plot(epochs, acc, "bo", label="Training accuracy")
plt.plot(epochs, val_acc, "b", label="Validation accuracy")
plt.title("Training and validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```

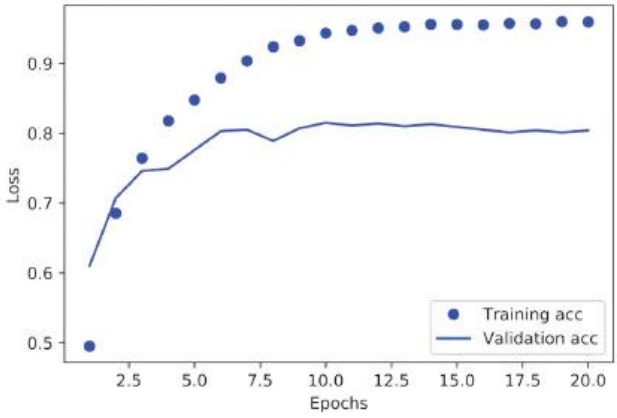


Figure 4.7 Training and validation accuracy

- The model begins to overfit after nine epochs. Let's train a new model from scratch for nine epochs and then evaluate it on the test set

Listing 4.21 Retraining a model from scratch

```
model = keras.Sequential([
    layers.Dense(64, activation="relu"),

    layers.Dense(64, activation="relu"),
    layers.Dense(46, activation="softmax")
])
model.compile(optimizer="rmsprop",
              loss="categorical_crossentropy",
              metrics=["accuracy"])
model.fit(x_train,
          y_train,
          epochs=9,
          batch_size=512)
results = model.evaluate(x_test, y_test)
```

Here are the final results:

```
>>> results
[0.9565213431445807, 0.7969239536954589]
```

- This approach reaches an accuracy of ~80%.
  - With a balanced binary classification problem, the accuracy reached by a purely random classifier would be 50%.
  - But in this case, we have 46 classes, and they may not be equally represented. What would be the accuracy of a random baseline? We could try quickly implementing one to check this empirically:

```
>>> import copy
>>> test_labels_copy = copy.copy(test_labels)
>>> np.random.shuffle(test_labels_copy)
>>> hits_array = np.array(test_labels) == np.array(test_labels_copy)
>>> hits_array.mean()
0.18655387355298308
```

- As you can see, a random classifier would score around 19% classification accuracy, so the results of our model seem pretty good in that light.

2.5. Generating predictions on new data

- Calling the model's *predict* method on new samples returns a class probability distribution over all 46 topics for each sample. Let's generate topic predictions for all of the test data

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

  - Each entry in "predictions" is a vector of length 46

```
>>> predictions[0].shape
(46,)
```

  - The coefficients in this vector sum to 1, as they form a probability distribution

```
>>> np.sum(predictions[0])
1.0
```

  - The largest entry is the predicted class – the class with the highest probability

```
>>> np.argmax(predictions[0])
4
```

2.6. A different way to handle the labels and the loss

- Another way to encode the labels would be to cast them as an integer tensor, like this:

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

  - The only thing this approach would change is the choice of the loss function. The loss function used before, categorical_crossentropy, expects the labels to follow a categorical encoding.
    - With integer labels, you should use *sparse_categorical_crossentropy*:

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

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

2.7. The importance of having sufficiently large intermediate layers

- Since the final outputs are 46-dimensional, you should avoid intermediate layers with many fewer than 46 units.
  - Let's now see what happens when we introduce an information bottleneck by having intermediate layers that are significantly less than 46-dimensional: for example, 4-dimensional

Listing 4.22 A model with an information bottleneck

```
model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(4, activation="relu"),
    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))
```

  - The model now peaks at ~71% validation accuracy, and 8% absolute drop.
    - This drop is mostly due to the fact that we're 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 model is able to cram *most* of the necessary information into these four-dimensional representations, but not all of it.

2.8. Further experiments

- I encourage you to try out the following experiments to train your intuition about the kind of configuration decisions you have to make with such models
  - Try using larger or smaller layers: 32 units, 128 units, and so on.
  - You used two intermediate layers before the final softmax classification layer. Now try using a single intermediate layer, or three intermediate layers

2.9. Wrapping up

- Here's what you should take away from this example
  - If you're trying to classify data points among N classes, your model should end with a dense layer of size N.
  - In a single-label, multiclass classification problem, your model 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 model and the true distribution of the targets
- ➢ There are two ways to handle labels in multiclass classification
  - ✧ Encoding the labels via categorical encoding (i.e. one-hot encoding) and use categorical_crossentropy as a 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, you should avoid creating information bottleneck in your model due to intermediate layers that are too small

3. Predicting house prices: regression

- The two previous examples were considered 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.

3.1. The Boston housing price dataset

- In this section, we'll attempt to predict the median price of homes in a given Boston suburb in the mid-1970s, given data about the suburb at the time, such as the crime rate, the local property tax rate, and so on.
  - ➢ The dataset we use has an interesting difference from the two previous examples. It has relatively few data points: only 506, split between 4040 training samples and 102 test samples, each with 13 numerical features.
  - ➢ Each *feature* in the input data (e.g. crime rate) has a different scale. For instance, some values are proportions, which take values between 0 and 1, others take values between 1 and 12, others between 0 and 100, and so on

Listing 4.23 Loading the Boston housing dataset

```
from tensorflow.keras.datasets import boston_housing
(train_data, train_targets), (test_data, test_targets) = (
    boston_housing.load_data())
```

- Let's look at the data

```
>>> train_data.shape
(404, 13)
>>> test_data.shape
(102, 13)
```

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

```
>>> train_targets
[ 15.2,  42.3,  50. ...  19.4,  19.4,  29.1]
```

  - ➢ The prices are typically between $10,000 and $50,000

3.2. Preparing the data

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

Listing 4.24 Normalizing the data

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

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

  - Note that the quantities used for normalizing the test data are computed using the training data.
    - You should never use quantity computed on the test data in your workflow, even for something as simple as data normalization

3.3. Building your model

- Since only few samples are available, we'll use a very small model with two intermediate layers, each with 64 units.
  - In general, the less training data you have, the worse overfitting will be, and using a small model is one way to mitigate overfitting.

Listing 4.25 Model definition

```
def build_model():
    model = keras.Sequential([
        layers.Dense(64, activation="relu"),
        layers.Dense(64, activation="relu"),
        layers.Dense(1)
    ])
    model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
    return model
```

Because we need to instantiate the same model multiple times, we use a function to construct it.

  - The model ends with a single unit and no activation (it will be a linear layer). This is a typical setup for scalar regression (a regression where you're trying to predict a single continuous value). Since the last layer is purely linear, the model is free to learn to predict values in any range.
  - We compile the model with the *mse* loss function, the square of the difference between the predictions and the targets. This is a widely used loss function for regression problems
  - We're also monitoring a new metric during training: *mean absolute error* (MAE). It's the absolute value of the difference between the predictions and the targets.
    - For instance, an MAE of 0.5 on this problem would mean your predictions are off by $500 on average

3.4. Validating your approach using K-fold validation

- To evaluate our model while we keep adjusting its parameters (e.g. number of epochs used for training), we could split the data into a training set and a validation set
  - However, since we have very few data points, the validation set would end up being very small (e.g. only 100 examples). As a consequence, the validation scores might change a lot depending on which data points were chosen for validation and which were for training:
  - The validation scores might have high variance with regard to the validation split. This would prevent us from reliably evaluating our model.

- The best practice in such situation is to use *K-fold cross-validation*.

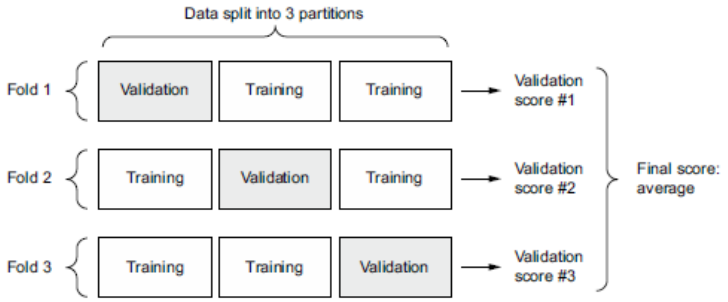


Figure 4.8 K-fold cross-validation with K=3

  - It consists of splitting the available data into K partitions (typically 4, or 5), 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 is then the average of the K validation scores obtained.

Listing 4.26 K-fold validation

```
k = 4
num_val_samples = len(train_data) // k
num_epochs = 100
all_scores = []
for i in range(k):
    print(f"Processing fold #{i}")
    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]
    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:]],
        axis=0)
    model = build_model()
    model.fit(partial_train_data, partial_train_targets,
              epochs=num_epochs, batch_size=16, verbose=0)
    val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
    all_scores.append(val_mae)
```

Prepares the validation data: data from partition #k

Prepares the training data: data from all other partitions

Builds the Keras model (already compiled)

Trains the model (in silent mode, verbose = 0)

Evaluates the model on the validation data

- Running this with 100 epochs yield the following results

```
>>> all_scores
[2.112449, 3.0801501, 2.6483836, 2.4275346]
>>> np.mean(all_scores)
2.5671294
```

- The different runs do indeed show rather different validation scores, from 2.1 to 3.1. The average 2.6 is a much more reliable metric than any single score – which is the point of K-fold cross-validation.
  - In this case, we're off by $2,600 on average, which is significant considering that the prices range from $10,000 to $50,000

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

Listing 4.27 Saving the validation logs at each fold

```
num_epochs = 500
all_mae_histories = []
for i in range(k):
    print(f"Processing fold #{i}")
    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]
    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:]],
        axis=0)
    model = build_model()
    history = model.fit(partial_train_data, partial_train_targets,
                        validation_data=(val_data, val_targets),
                        epochs=num_epochs, batch_size=16, verbose=0)
    mae_history = history.history["val_mae"]
    all_mae_histories.append(mae_history)
```

Prepares the validation data: data from partition #k

Prepares the training data: data from all other partitions

Builds the Keras model (already compiled)

Trains the model (in silent mode, verbose=0)

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

Listing 4.28 Building the history of successive mean K-fold validation scores

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

- Let's plot this

Listing 4.29 Plotting validation scores

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

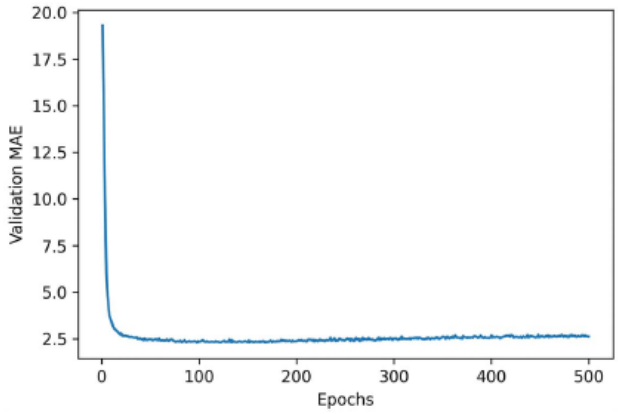
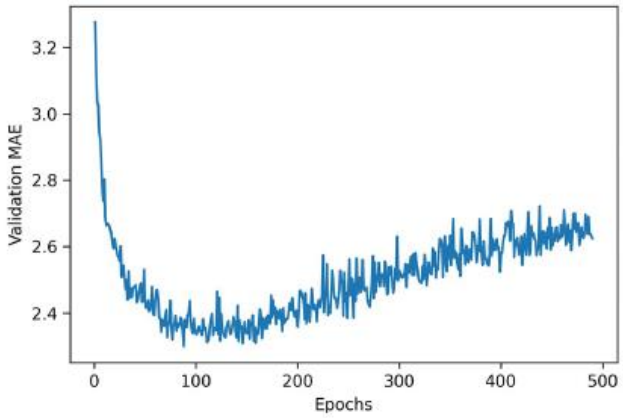


Figure 4.9 Validation MAE by epoch

- It may be a little difficult to read the plot, due to a scaling issue: the validation MAE for the first few epochs is dramatically higher than the values that follow. Let's omit the first 10 data points, which are on a different scale than the rest of the curve

Listing 4.30 Plotting validation scores, excluding the first 10 data points

```
truncated_mae_history = average_mae_history[10:]
plt.plot(range(1, len(truncated_mae_history) + 1), truncated_mae_history)
plt.xlabel("Epochs")
plt.ylabel("Validation MAE")
plt.show()
```



- Validation MAE stops improving significantly after 120-140 epochs (this number includes the 10 epochs we omitted). Past that point, we start overfitting

- After you're finished tunning other parameters of the model (in addition to the number of epochs, you could also adjust the size of the intermediate layers), you can train a final production model on all of the training data, with the best parameters, and then look at its performance on the test data

Listing 4.31 Training the final model

```
model = build_model()                    # Gets a fresh, compiled model
model.fit(train_data, train_targets,     # Trains it on the entirety of the data
          epochs=130, batch_size=16, verbose=0)
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)
```

- The final result is

```
>>> test_mae_score
2.4642276763916016
```

- We're still off by a bit under $2,500. It's an improvement! Just like with the two previous tasks, you can try varying the number of layers in the model, or the number of units per layer, to see if you can squeeze out a lower test error.

3.5. Generating predictions on new data

- With scalar regression model, *predict( )* returns the model's guess for the sample's price in thousands of dollars
  - Recall that with binary classification model, it retrieves a scalar score between 0 and 1 for each input sample

- With multiclass classification model, we retrieved a probability distribution over all classes for each sample

```
>>> predictions = model.predict(test_data)
>>> predictions[0]
array([9.990133], dtype=float32)
```

3.6. Wrapping up

- Here is what you should take away from this scalar regression example
  - Regression is done using different loss functions than we used for classification.
    - MSE is a loss function commonly used for regression
  - Similarly, evaluation metrics to be used for regression differ from those used for classification; naturally, the concept of accuracy doesn't apply for regression.
    - A common regression metric is MAE
  - When features in the input data values in different ranges, each feature should be scaled independently as a preprocessing step
  - When there is a little data available, using K-fold validation is a great way to reliably evaluate a model
  - When little training data is available, it's preferable to use a small model with few intermediate layers (typically only one or two), in order to avoid severe overfitting.

Summary

- The three most common kinds of machine learning tasks on vector data are binary classification, multiclass classification, and scalar regression.

- You'll usually need to preprocess raw data before feeding it into a neural network

- When your data has features with different ranges, scale each feature independently as part of preprocessing

- As training progresses, neural networks eventually begin to overfit and obtain worse results on never-before-seen data

- If you don't have much training data, use a small model with only one or two intermediate layers, to avoid severe overfitting.

- If your data is divided into many categories, you may cause information bottlenecks if you make the intermediate layers too small

- When you're working with little data, K-fold validation can help reliably evaluate your model