Binary_and_Multi_Class_Classification_Using_Neural_Networks

January 25, 2024

1 Getting started with neural networks: Classification and regression

1.1 Classifying movie reviews: A binary classification example

1.1.1 The IMDB dataset

Loading the IMDB dataset

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

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/imdb.npz
17464789/17464789 [==============================] - 1s 0us/step
```

[ ]:
```
train_data[0]
```

[ ]:
```
[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,
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,
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,
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,
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,
226,
65,
16,
38,
1334,
88,
12,
16,
283,

```
5,
16,
4472,
113,
103,
32,
15,
16,
5345,
19,
178,
32]
```

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

[ ]:
```
1
```

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

[ ]:
```
9999
```

Decoding reviews back to text

The code takes the integer indices of words in a movie review from the IMDb dataset, decodes these indices into words using the reverse_word_index, and then joins them into a string. The resulting decoded_review is a human-readable representation of the review text. This process is often done when you want to inspect or print out a review to understand its content in a more interpretable form.

[ ]:
```
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]])
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/imdb_word_index.json
1641221/1641221 [==============================] - 0s 0us/step
```

[ ]:
```
decoded_review
```

[ ]:
```
"? this film was just brilliant casting location scenery story direction
everyone's really suited the part they played and you could just imagine being
there robert ? is an amazing actor and now the same being director ? father came
from the same scottish island as myself so i loved the fact there was a real
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 released 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 you 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 brilliant children are
often left out of the ? list i think because the stars that play them all grown
up are such a big profile for the whole film but these children are amazing and
should be praised for what they have done don't you think the whole story was so
lovely because it was true and was someone's life after all that was shared with
us all"
```

1.1.2 Preparing the data

Encoding the integer sequences via multi-hot encoding

[ ]:
```python
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)
```

[ ]:
```python
x_train[0]
```

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

[ ]:
```python
y_train = np.asarray(train_labels).astype("float32")
y_test = np.asarray(test_labels).astype("float32")
```

1.1.3 Building your model

Model definition

[ ]:
```python
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")
])
```

Compiling the model

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

1.1.4 Validating your approach

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:]
```

Training your model

[ ]:
```
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 [==============================] - 4s 117ms/step - loss: 0.5616 -
accuracy: 0.7533 - val_loss: 0.4326 - val_accuracy: 0.8611
Epoch 2/20
30/30 [==============================] - 1s 47ms/step - loss: 0.3562 - accuracy:
0.8879 - val_loss: 0.3364 - val_accuracy: 0.8782
Epoch 3/20
30/30 [==============================] - 1s 35ms/step - loss: 0.2629 - accuracy:
0.9161 - val_loss: 0.2914 - val_accuracy: 0.8892
Epoch 4/20
30/30 [==============================] - 1s 37ms/step - loss: 0.2119 - accuracy:
0.9323 - val_loss: 0.2882 - val_accuracy: 0.8842
Epoch 5/20
30/30 [==============================] - 1s 35ms/step - loss: 0.1758 - accuracy:
0.9437 - val_loss: 0.2768 - val_accuracy: 0.8890
Epoch 6/20
30/30 [==============================] - 1s 37ms/step - loss: 0.1493 - accuracy:
0.9525 - val_loss: 0.3029 - val_accuracy: 0.8808
Epoch 7/20
30/30 [==============================] - 1s 48ms/step - loss: 0.1286 - accuracy:
0.9623 - val_loss: 0.2881 - val_accuracy: 0.8854
Epoch 8/20
30/30 [==============================] - 1s 36ms/step - loss: 0.1071 - accuracy:
0.9699 - val_loss: 0.3257 - val_accuracy: 0.8788
Epoch 9/20
30/30 [==============================] - 1s 44ms/step - loss: 0.0959 - accuracy:
0.9725 - val_loss: 0.3108 - val_accuracy: 0.8843
Epoch 10/20
30/30 [==============================] - 2s 55ms/step - loss: 0.0816 - accuracy:
0.9791 - val_loss: 0.3281 - val_accuracy: 0.8818
Epoch 11/20
30/30 [==============================] - 1s 44ms/step - loss: 0.0699 - accuracy:
```

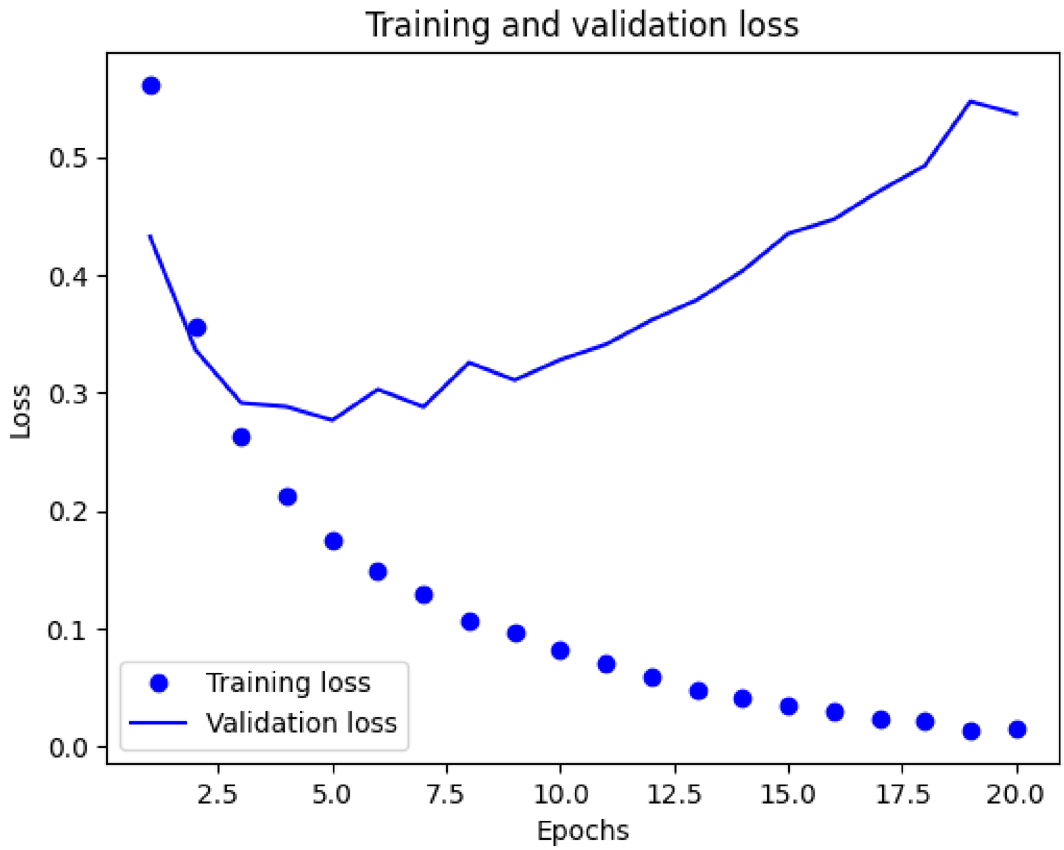
```
0.9827 - val_loss: 0.3411 - val_accuracy: 0.8822
Epoch 12/20
30/30 [==============================] - 1s 37ms/step - loss: 0.0590 - accuracy:
0.9859 - val_loss: 0.3617 - val_accuracy: 0.8794
Epoch 13/20
30/30 [==============================] - 1s 37ms/step - loss: 0.0484 - accuracy:
0.9905 - val_loss: 0.3789 - val_accuracy: 0.8787
Epoch 14/20
30/30 [==============================] - 1s 33ms/step - loss: 0.0416 - accuracy:
0.9927 - val_loss: 0.4035 - val_accuracy: 0.8754
Epoch 15/20
30/30 [==============================] - 1s 36ms/step - loss: 0.0339 - accuracy:
0.9951 - val_loss: 0.4350 - val_accuracy: 0.8721
Epoch 16/20
30/30 [==============================] - 1s 37ms/step - loss: 0.0294 - accuracy:
0.9953 - val_loss: 0.4472 - val_accuracy: 0.8737
Epoch 17/20
30/30 [==============================] - 1s 36ms/step - loss: 0.0234 - accuracy:
0.9971 - val_loss: 0.4712 - val_accuracy: 0.8723
Epoch 18/20
30/30 [==============================] - 1s 35ms/step - loss: 0.0215 - accuracy:
0.9963 - val_loss: 0.4928 - val_accuracy: 0.8714
Epoch 19/20
30/30 [==============================] - 1s 35ms/step - loss: 0.0136 - accuracy:
0.9994 - val_loss: 0.5473 - val_accuracy: 0.8688
Epoch 20/20
30/30 [==============================] - 1s 38ms/step - loss: 0.0158 - accuracy:
0.9975 - val_loss: 0.5367 - val_accuracy: 0.8701
```

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

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

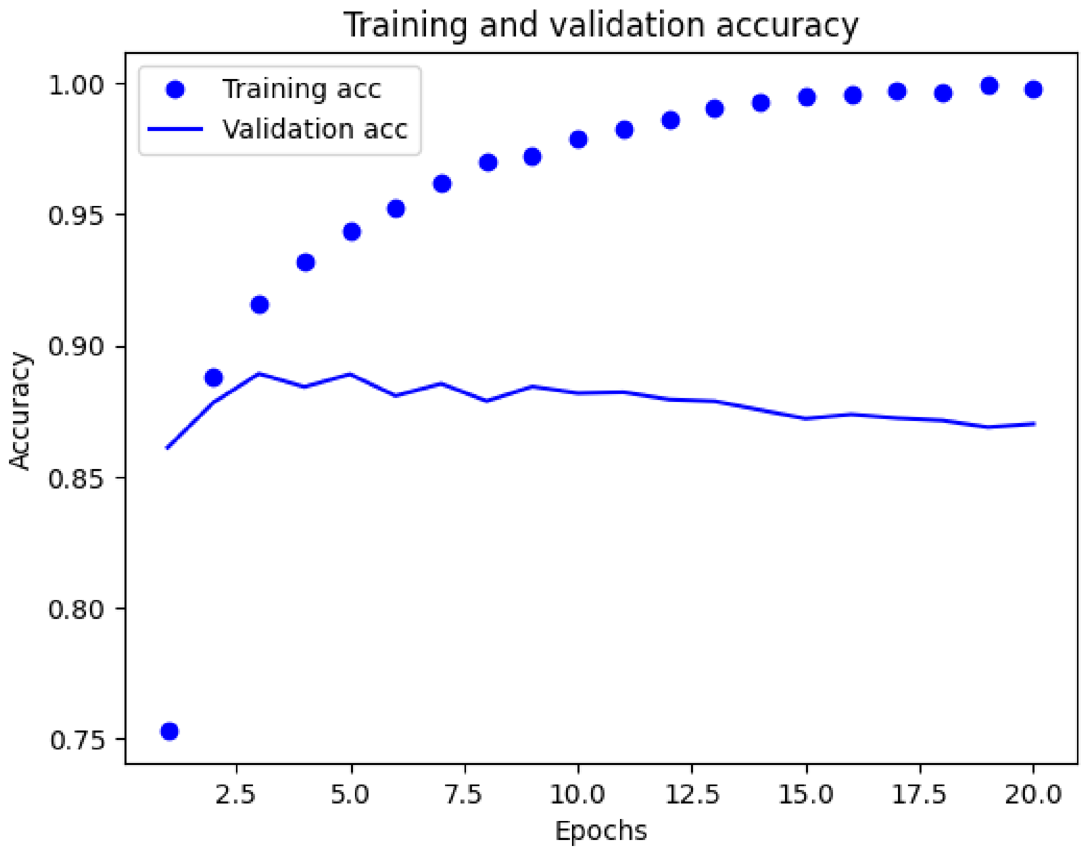
Plotting the training and validation loss

[ ]:
```
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()
```



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()
```



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)
```

```
Epoch 1/4
49/49 [==============================] - 3s 26ms/step - loss: 0.4633 - accuracy:
0.8036
Epoch 2/4
49/49 [==============================] - 1s 26ms/step - loss: 0.2734 - accuracy:
0.9046
Epoch 3/4
```

```
49/49 [==============================] - 1s 26ms/step - loss: 0.2131 - accuracy:
0.9240
Epoch 4/4
49/49 [==============================] - 1s 26ms/step - loss: 0.1797 - accuracy:
0.9376
782/782 [==============================] - 4s 4ms/step - loss: 0.2948 -
accuracy: 0.8824
```

The function evaluate returns a list loss_and_metrics, where the first element is the loss value, and subsequent elements are the values of the specified metrics.

[ ]:
```
results
```

[ ]:
```
[0.29477736353874207, 0.8823999762535095]
```

1.1.5 Using a trained model to generate predictions on new data

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

```
782/782 [==============================] - 2s 2ms/step
```

[ ]:
```
array([[0.18113604],
       [0.997376  ],
       [0.41908303],
       …,
       [0.11241841],
       [0.06462241],
       [0.5789311 ]], dtype=float32)
```

1.1.6 Further experiments

1.1.7 Wrapping up

1.2 Classifying newswires: A multiclass classification example

1.2.1 The Reuters dataset

Loading the Reuters dataset

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

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/reuters.npz
2110848/2110848 [==============================] - 1s 0us/step
```

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

[ ]:
```
8982
```

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

```
[ ]: 2246
```

```
[ ]: 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]
```

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]])
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/reuters_word_index.json
550378/550378 [==============================] - 0s 1us/step
```

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

[ ]: 3

1.2.2 Preparing the data

Encoding the input data

[ ]:
```
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

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)
```

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

Categorical Crossentropy Loss:

Many multi-class classification problems, including the Reuters dataset, use categorical crossentropy as the loss function. Categorical crossentropy expects labels to be in a one-hot encoded format. Each document is associated with one class, and the neural network aims to predict the probability distribution over all classes.

Model Output Layer:

In a neural network designed for multi-class classification, the output layer typically has as many neurons as there are classes. The softmax activation function is commonly used in the output layer, and it requires one-hot encoded labels to calculate probabilities for each class.

Consistency and Compatibility:

Using one-hot encoding ensures consistency in data representation and is a common practice when dealing with categorical variables in machine learning. It aligns with the conventions and expectations of machine learning libraries like Keras.

1.2.3 Building your model

Model definition

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

```
    layers.Dense(46, activation="softmax")
])
```

Compiling the model

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

1.2.4 Validating your approach

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:]
```

Training the model

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

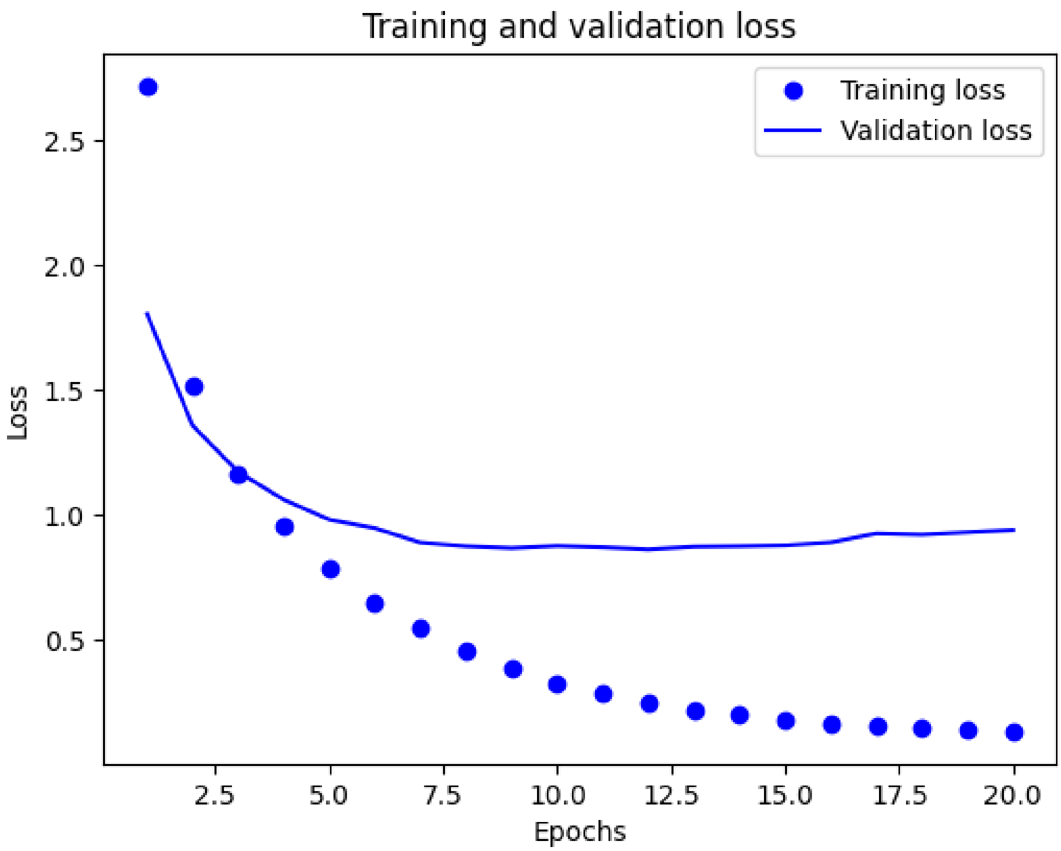
```
Epoch 1/20
16/16 [==============================] - 3s 101ms/step - loss: 2.7187 -
accuracy: 0.4748 - val_loss: 1.8061 - val_accuracy: 0.6050
Epoch 2/20
16/16 [==============================] - 1s 59ms/step - loss: 1.5180 - accuracy:
0.6741 - val_loss: 1.3586 - val_accuracy: 0.7010
Epoch 3/20
16/16 [==============================] - 1s 53ms/step - loss: 1.1628 - accuracy:
0.7456 - val_loss: 1.1731 - val_accuracy: 0.7390
Epoch 4/20
16/16 [==============================] - 1s 54ms/step - loss: 0.9557 - accuracy:
0.7876 - val_loss: 1.0619 - val_accuracy: 0.7700
Epoch 5/20
16/16 [==============================] - 1s 53ms/step - loss: 0.7885 - accuracy:
0.8249 - val_loss: 0.9818 - val_accuracy: 0.7910
Epoch 6/20
16/16 [==============================] - 1s 50ms/step - loss: 0.6509 - accuracy:
0.8598 - val_loss: 0.9478 - val_accuracy: 0.7910
Epoch 7/20
16/16 [==============================] - 1s 51ms/step - loss: 0.5504 - accuracy:
0.8806 - val_loss: 0.8900 - val_accuracy: 0.8190
Epoch 8/20
```

```
16/16 [==============================] - 1s 50ms/step - loss: 0.4545 - accuracy:
0.9012 - val_loss: 0.8754 - val_accuracy: 0.8170
Epoch 9/20
16/16 [==============================] - 1s 51ms/step - loss: 0.3834 - accuracy:
0.9173 - val_loss: 0.8678 - val_accuracy: 0.8140
Epoch 10/20
16/16 [==============================] - 1s 54ms/step - loss: 0.3273 - accuracy:
0.9282 - val_loss: 0.8767 - val_accuracy: 0.8040
Epoch 11/20
16/16 [==============================] - 1s 49ms/step - loss: 0.2880 - accuracy:
0.9354 - val_loss: 0.8710 - val_accuracy: 0.8140
Epoch 12/20
16/16 [==============================] - 1s 48ms/step - loss: 0.2450 - accuracy:
0.9441 - val_loss: 0.8624 - val_accuracy: 0.8130
Epoch 13/20
16/16 [==============================] - 1s 81ms/step - loss: 0.2173 - accuracy:
0.9471 - val_loss: 0.8735 - val_accuracy: 0.8240
Epoch 14/20
16/16 [==============================] - 1s 83ms/step - loss: 0.2009 - accuracy:
0.9475 - val_loss: 0.8755 - val_accuracy: 0.8230
Epoch 15/20
16/16 [==============================] - 1s 56ms/step - loss: 0.1792 - accuracy:
0.9521 - val_loss: 0.8788 - val_accuracy: 0.8250
Epoch 16/20
16/16 [==============================] - 1s 49ms/step - loss: 0.1669 - accuracy:
0.9533 - val_loss: 0.8906 - val_accuracy: 0.8280
Epoch 17/20
16/16 [==============================] - 1s 50ms/step - loss: 0.1554 - accuracy:
0.9549 - val_loss: 0.9268 - val_accuracy: 0.8220
Epoch 18/20
16/16 [==============================] - 1s 47ms/step - loss: 0.1478 - accuracy:
0.9553 - val_loss: 0.9223 - val_accuracy: 0.8190
Epoch 19/20
16/16 [==============================] - 1s 50ms/step - loss: 0.1393 - accuracy:
0.9563 - val_loss: 0.9314 - val_accuracy: 0.8200
Epoch 20/20
16/16 [==============================] - 1s 50ms/step - loss: 0.1339 - accuracy:
0.9565 - val_loss: 0.9403 - val_accuracy: 0.8230
```

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")
```

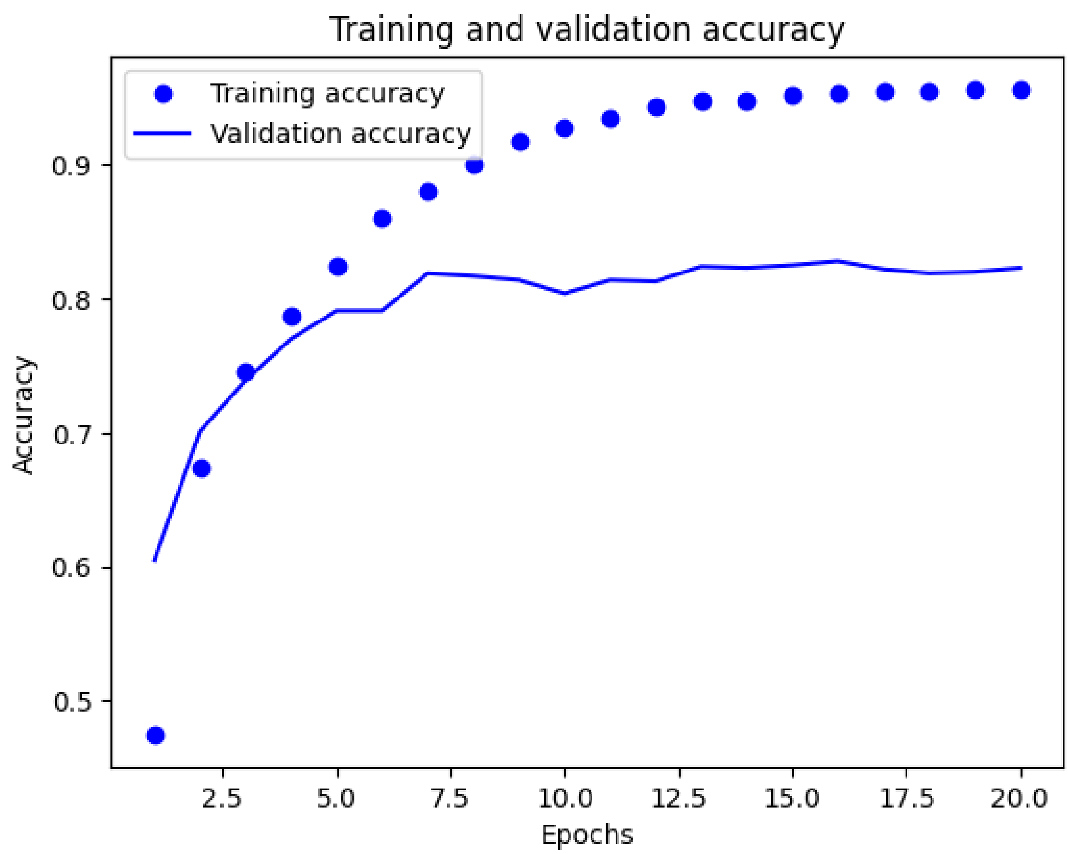
```
plt.ylabel("Loss")
plt.legend()
plt.show()
```



Plotting the training and validation accuracy

[ ]:

```
plt.clf()
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()
```



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)
```

```
Epoch 1/9
18/18 [==============================] - 1s 50ms/step - loss: 2.6560 - accuracy:
0.5217
Epoch 2/9
```

```
18/18 [==============================] - 1s 44ms/step - loss: 1.4866 - accuracy:
0.6894
Epoch 3/9
18/18 [==============================] - 1s 48ms/step - loss: 1.1312 - accuracy:
0.7572
Epoch 4/9
18/18 [==============================] - 1s 45ms/step - loss: 0.9166 - accuracy:
0.8066
Epoch 5/9
18/18 [==============================] - 1s 44ms/step - loss: 0.7474 - accuracy:
0.8413
Epoch 6/9
18/18 [==============================] - 1s 47ms/step - loss: 0.6189 - accuracy:
0.8702
Epoch 7/9
18/18 [==============================] - 1s 46ms/step - loss: 0.5157 - accuracy:
0.8902
Epoch 8/9
18/18 [==============================] - 1s 44ms/step - loss: 0.4334 - accuracy:
0.9076
Epoch 9/9
18/18 [==============================] - 1s 42ms/step - loss: 0.3660 - accuracy:
0.9205
71/71 [==============================] - 0s 4ms/step - loss: 0.9366 - accuracy:
0.7921
```

[ ]:
```
results
```

[ ]:
```
[0.9366254806518555, 0.7920747995376587]
```

1.2.5 Generating predictions on new data

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

```
71/71 [==============================] - 0s 4ms/step
```

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

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

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

[ ]:
```
0.9999999
```

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

[ ]:
```
3
```