Neural network face recognition

To see more of the power of neural networks, we are going to program a more involved example.

Start by importing some packages we are going to use.

```
import matplotlib.pyplot as plt
import numpy as np
from random import randint
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
```

Data

We are going to load a dataset called Labeled Faces in the Wild (LFW) people, which can be loaded by scikit-learn. This is a set of, weirdly enough, early-2000s politicians. Each picture is centered on a single face and the data come with labels telling you the name of the person pictured.

Here we import the dataset and store it.

```
lfw_people = datasets.fetch_lfw_people(min_faces_per_person=70, resize=0.4)
data = lfw_people.data
labels = lfw_people.target
target_names = lfw_people.target_names
```

We stored the data in `data`. The labels, which are numbers assigned to each person, in `labels`, and the names of the people corresponding to each label in `target_names`. We will use `labels` to train a model, but use `target_names` when displaying output because it makes more sense to see `George W Bush` than it does `3`.

Splitting testing and training data

It is usual to split the data into a large set for training and a smaller set for testing, as we have done previously. scikit-learn provides a function `train_test_split` which will randomly allocate data into either the training or testing groups. We pass this the `data` and the `labels`, and use `test_size` to tell it what proportion of the data to allocate for testing - in this case 10%.

```
train_data, test_data, train_labels, test_labels = train_test_split(data, labe
```

Take a look at the data

Now we have a set of training data, `train_data`, and its associated labels `train_labels`.

We can take a look at what we've got using a matplotlib function `imshow`. The images are supposed to be 50 by 37 pixels greyscale, so we tell `imshow` this.

Here we use `subplot` to draw a 3 by 3 grid of images and their `target_names` randomly selected from the training data. `.xticks` and `.yticks` remove the axes, and we add some titles.

```
for i in range(9):
    n = randint(0,len(train_data)-1)
    plt.subplot(3, 3, i + 1)
    plt.imshow(train_data[n].reshape((50, 37)), cmap=plt.cm.gray)
    plt.title(target_names[train_labels[n]], size=10)
    plt.xticks(())
    plt.yticks(())
plt.suptitle("Training data (sample)")
plt.show()
```

Training the model

As before, we set up a `StandardScaler` based on the training data to scale it.

```
scaler = StandardScaler()
train_data_scaled = scaler.fit_transform(train_data)
```

Now we are ready to train the model. Here we use a `MLPClassifier` with two hidden layers of 100 neurons each.

```
model = MLPClassifier(hidden_layer_sizes=(100,100))
model.fit(train_data_scaled, train_labels)
```

Testing the model

Here we pull ten random entries from the testing data and display them in the same way as above. We scale the entry using the same scaler as we used for the training data, then use the model to predict the label. Finally, we use the label prediction to look up the name of the predicted person, and display this alongside the name the image is labelled with.

```
for i in range(10):
    n = randint(0,len(test_data)-1)
    plt.imshow(test_data[n].reshape((50, 37)), cmap=plt.cm.gray)

    test_data_scaled = scaler.transform([test_data[n]])

    prediction = model.predict(test_data_scaled)[0]

    plt.title(f"Prediction: {target_names[prediction]}\nLabel: {target_names[t
    plt.xticks(())
```

```
    plt.yticks(())
    plt.show()
```

How did your model do? Did it get most people right? We can perform more sophisticated tests than just sampling a few random items from the data. One thing we could do is loop through all the testing data and track which are right and wrong. Actually, there is a way of doing this from scikit-learn called a confusion matrix.

First we scale the whole array of testing data.

```
test_data_scaled = scaler.transform(test_data)
```

Now we ask the model to make predictions for every item of testing data.

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

The function `ConfusionMatrixDisplay` takes the correct labels for the testing data and the predictions made by the model and uses these to visualise where the model is right and wrong. We are also passing `display_labels`, the names of the people associated with each label, and two parameters to rotate the x-axis labels (because otherwise they overlap) and make the colours more sedate than the default.

```
from sklearn.metrics import ConfusionMatrixDisplay

ConfusionMatrixDisplay.from_predictions(test_labels, predictions, display_labe

plt.tight_layout()
plt.show()
```

This produces a grid of numbers. Each row represents a true label, and the columns give the numbers of predictions made for each label. The results will vary as there is randomness in the selection of the testing data and the model training, but for example, if I have a `1` on the `Tony Blair` row in the `George W Bush` column, that means one picture of Tony Blair was incorrectly predicted by the model to be a picture of George W Bush. The items on the main diagonal represent pictures that were labelled correctly. If the model was trained well, these should be the largest values by far.

If you adjust the number of hidden layers, their sizes, and the number of epochs (controlled by `max_iter`), you can see the effect on how well the model performs. For example, a version with no hidden layers and only 2 epochs such perform much worse.

```
model = MLPClassifier(hidden_layer_sizes=(),max_iter=2)
```

Note

Note that really for this sort of task you would use an approach called convolution neural networks, but these aren't available in scikit-learn. Try PyTorch or TensorFlow if you are interested.