COMP6685_Lab5b_Keras_MLP_MNIST

March 2, 2025

1 MLP for Complex Problems: The MNIST dataset

In this exercise we will first learn to use the simple perceptron network (input-output layers only) and a Multi-Layer Perceptron (MLP, with one or more hidden layers). To make the task more interesting than the XOR problem, we will be using a more complex training set. This will be the MNIST dataset, a well known neural network problem for the recognition of the 10 handwritten characters from 0 to 9 (MNIST).

This exercise is based on the Gulli & Pal (2017) 'Deep Learning with Keras' textbook, with some additional code to help us understand and test the programme.

Importing the libraries and defining the main training parameters

The initial code is necessary to prepare the data and the simulation (hyper)parameters. We first import numpy. In our case we will use it to create and pre-process the array of the training data sets. We then import a few functions from Keras (we used some of these in our previous XOR exercise). The matplotlib library will be used for visualising some MNIST images and the plot of the training results.

The code also defines the variables for some of the main parameters used throughout this program. The random seed definition is also important to be able to repeat the same parameter configuration.

[1]:
```
# import of numpy and keras libraries
from __future__ import print_function
import numpy as np
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.optimizers import SGD
from tensorflow.keras import utils
import matplotlib.pyplot as plt

# variables for network and training
N_EPOCH = 200 # initially set at 200 ; you can change this later
BATCH_SIZE = 128
VERBOSE = 1
N_CLASSES = 10   # number of classes/categories of digits from 0 to 9, i.e.␣
 ↪number of output units
OPTIMIZER = SGD(learning_rate=0.1) # Stochastic gradient descent optimiser
N_HIDDEN = 128   # number of hidden units
```

```
VALIDATION_SPLIT=0.2 # proportion of the dataset used for validation, with␣
 ↪remaining .8 for training

#each 2D image consists of 28x28 values/pixels, which needs to be reshaped in a␣
 ↪vector of 784 pixels
RESHAPED = 784

# random seed number to be used for reproducibility
np.random.seed(1671)
```

```
2025-02-03 12:44:38.268032: I tensorflow/core/platform/cpu_feature_guard.cc:210]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other
operations, rebuild TensorFlow with the appropriate compiler flags.
```

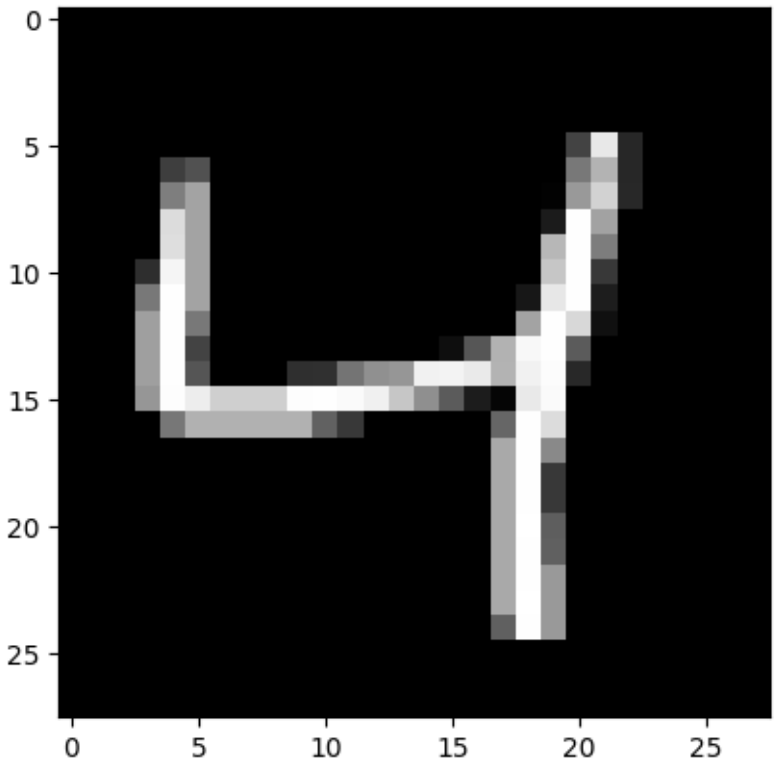
Preparing the MNIST dataset and visualising the input images

This part of the code prepares the input and output training set, and the corresponding test sets. It also visualises a sample image. The MNIST dataset is included in the Keras program and we do not need to use and external file.

[ ]:
```
# data: shuffled and split between train and test sets, loading and using the␣
 ↪Keras mnist dataset
(input_X_train, output_Y_train), (input_X_test, output_Y_test) = mnist.
 ↪load_data()

# print the shapes of the input and output data
print("Training data input shape: " , input_X_train.shape)
print("Training data output shape: " , output_Y_train.shape)
print("Test data input shape: " , input_X_test.shape)
print("Test data ouput shape: " , output_Y_test.shape)

# visualisation of the numerical vector and plot of a selected image
Selected_Image = 2
image = input_X_train[Selected_Image]
print ("Sample input image: " + str(image))
plt.imshow(image, cmap='gray')
plt.show()
```

```
Training data input shape:  (60000, 28, 28)
Training data output shape:  (60000,)
Test data input shape:  (10000, 28, 28)
Test data ouput shape:  (10000,)
Sample input image: [[  0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

```
    0   0   0   0   0   0   0   0   0   0]
[   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
[   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]]
```



The input images now have to be reshaped as a linear vector. That is, we go from a 2D image of 28x28 pixels, to a linear vector of 784 (i.e. 28*28) pixels, to be passed as the 784 input units. Moreover, the initial pixel grey values given as type **int** in the range 0-255 will be normalised to the **float32** type in the range 0-1.

[ ]:
```
# use 60000 images for training, 10000 for validation test
input_X_train = input_X_train.reshape(60000, RESHAPED)
input_X_test = input_X_test.reshape(10000, RESHAPED)
input_X_train = input_X_train.astype('float32')
input_X_test = input_X_test.astype('float32')

# normalisation of the pixel values from 0-255 range to 0-1 range
input_X_train /= 255
input_X_test /= 255

print ("Input data ready")
```

```
Input data ready
```

Preparing the output labels

This code converts the output data into categorical (one-hot encoding) vectors of 0s and 1s. See example of the visualisation of the one-hot vector for the selected image.

[4]:
```
# convert class vectors to binary class matrices
output_Y_train = utils.to_categorical(output_Y_train, N_CLASSES)
output_Y_test = utils.to_categorical(output_Y_test, N_CLASSES)

# print the categorical, one-hot output vector for the sample image
label = output_Y_train[Selected_Image]
print ("One-hot-vector: " + str(label))
```

```
One-hot-vector: [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
```

2 Training the Simple Perceptron

Defining the network: Simple Perceptron

We will start by training a simple perceptron, i.e. a network with an input layer (the 784 input values/pixels) connected to the output layer (the 10 number classes)

[5]:
```
# Defaults sequential model
model = Sequential()

# Dense layer for all to all connections
# Define the output layer with 10 output units, and softmax activation as␣
 ↪categorical output
model.add(Dense(N_CLASSES, input_shape=(RESHAPED,)))
model.add(Activation('softmax'))

# Use categorical crossentropy for the loss evaluation, and the accuracy metrics
# we previously chose the SGD optimiser in the OPTIMIZER variable definition
model.compile(loss='categorical_crossentropy', optimizer=OPTIMIZER,␣
 ↪metrics=['accuracy'])

#show the model summary
model.summary()
```

```
/usr/local/anaconda3/envs/ml-algo/lib/python3.12/site-
packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Model: "sequential"
```

```
Layer (type)                    Output Shape                  Param #

dense (Dense)                   (None, 10)                    7,850

activation (Activation)         (None, 10)                    0


Total params: 7,850 (30.66 KB)

Trainable params: 7,850 (30.66 KB)

Non-trainable params: 0 (0.00 B)
```

Let's train the simple perceptron network

Let's now train (fit) the network with the above-defined batch size (128), and number of epochs (200). We save the training results into the history variable.

Here we use the previous **VALIDATION_SPLIT=0.2** definition to split the dataset into a 20% (0.2) validation set and the remaning 80% as training set.

[6]:
```
#train the network
history = model.fit(input_X_train, output_Y_train, batch_size=BATCH_SIZE,␣
 ↪epochs=N_EPOCH, verbose=VERBOSE, validation_split=VALIDATION_SPLIT)
```

```
Epoch 1/200
375/375             2s 5ms/step -
accuracy: 0.7631 - loss: 0.9328 - val_accuracy: 0.8954 - val_loss: 0.3969
Epoch 2/200
375/375             2s 5ms/step -
accuracy: 0.8880 - loss: 0.4115 - val_accuracy: 0.9059 - val_loss: 0.3481
Epoch 3/200
375/375             1s 4ms/step -
accuracy: 0.9015 - loss: 0.3570 - val_accuracy: 0.9096 - val_loss: 0.3279
Epoch 4/200
375/375             1s 3ms/step -
accuracy: 0.9026 - loss: 0.3494 - val_accuracy: 0.9127 - val_loss: 0.3157
Epoch 5/200
375/375             1s 3ms/step -
accuracy: 0.9083 - loss: 0.3304 - val_accuracy: 0.9149 - val_loss: 0.3084
Epoch 6/200
375/375             1s 3ms/step -
accuracy: 0.9112 - loss: 0.3183 - val_accuracy: 0.9150 - val_loss: 0.3005
Epoch 7/200
375/375             1s 3ms/step -
accuracy: 0.9124 - loss: 0.3143 - val_accuracy: 0.9178 - val_loss: 0.2967
```

```
Epoch 200/200
375/375             3s 9ms/step -
accuracy: 0.9339 - loss: 0.2388 - val_accuracy: 0.9284 - val_loss: 0.2674
```

Looking at the results of the trained network

Let's evaluate the model to see how well it has learned (or not) the MNIST problem.

[7]:
```
#test the network using the generalisation test dataset
score = model.evaluate(input_X_test, output_Y_test, verbose=VERBOSE)
print("\nTest score/loss:", score[0])
print("Test accuracy:", score[1])
```

```
313/313             2s 8ms/step -
accuracy: 0.9166 - loss: 0.3055

Test score/loss: 0.2687970995903015
Test accuracy: 0.9257000088691711
```

3 Training the Multi-Layer Perceptron

Defining the network: Multi-Layer Perceptron

We will now create a multi-layer perceptron with 784 input units, two hidden layers with 128 hidden units each, and an output layer with the 10 units.

[13]:
```
N_EPOCH = 20 # we need fewer epoch than before, as the multi-layer percetpron␣
 ↪can learn faster.
N_HIDDEN = 128

model = Sequential()

# Hidden layer 1 with 128 hidden units and ReLu activation function
model.add(Dense(N_HIDDEN, input_shape=(RESHAPED,)))
model.add(Activation('relu'))
# Hidden layer 2 with 128 hidden units and ReLu activation function
model.add(Dense(N_HIDDEN))
model.add(Activation('relu'))

# output layer with 10 units and softmax activation
model.add(Dense(N_CLASSES))
model.add(Activation('softmax'))

# Summary of the whole model
model.summary()

OPTIMIZER = SGD(learning_rate=0.1) # Stochastic gradient descent optimiser

# model compilation
```

```
model.compile(loss='categorical_crossentropy', optimizer=OPTIMIZER,␣
  ↪metrics=['accuracy'], run_eagerly=True)
```

```
/usr/local/anaconda3/envs/ml-algo/lib/python3.12/site-
packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
dense_7 (Dense)	(None, 128)	100,480
activation_7 (Activation)	(None, 128)	0
dense_8 (Dense)	(None, 128)	16,512
activation_8 (Activation)	(None, 128)	0
dense_9 (Dense)	(None, 10)	1,290
activation_9 (Activation)	(None, 10)	0

Total params: 118,282 (462.04 KB)

Trainable params: 118,282 (462.04 KB)

Non-trainable params: 0 (0.00 B)

Let's train the multi-layer perceptron network

Let's now train (fit) the network with the above-defined batch size (128), and number of epochs (20).

[14]:
```
#train the network
history = model.fit(input_X_train, output_Y_train, batch_size=BATCH_SIZE,␣
  ↪epochs=N_EPOCH, verbose=VERBOSE, validation_split=VALIDATION_SPLIT)
```

```
Epoch 1/20
375/375            37s 98ms/step -
accuracy: 0.7702 - loss: 0.8302 - val_accuracy: 0.9183 - val_loss: 0.2760
Epoch 2/20
```

```
375/375             15s 39ms/step -
accuracy: 0.9937 - loss: 0.0253 - val_accuracy: 0.9728 - val_loss: 0.0898
Epoch 19/20
375/375             16s 43ms/step -
accuracy: 0.9943 - loss: 0.0239 - val_accuracy: 0.9747 - val_loss: 0.0883
Epoch 20/20
375/375             12s 32ms/step -
accuracy: 0.9951 - loss: 0.0215 - val_accuracy: 0.9749 - val_loss: 0.0910
```

Looking at the results of the trained network

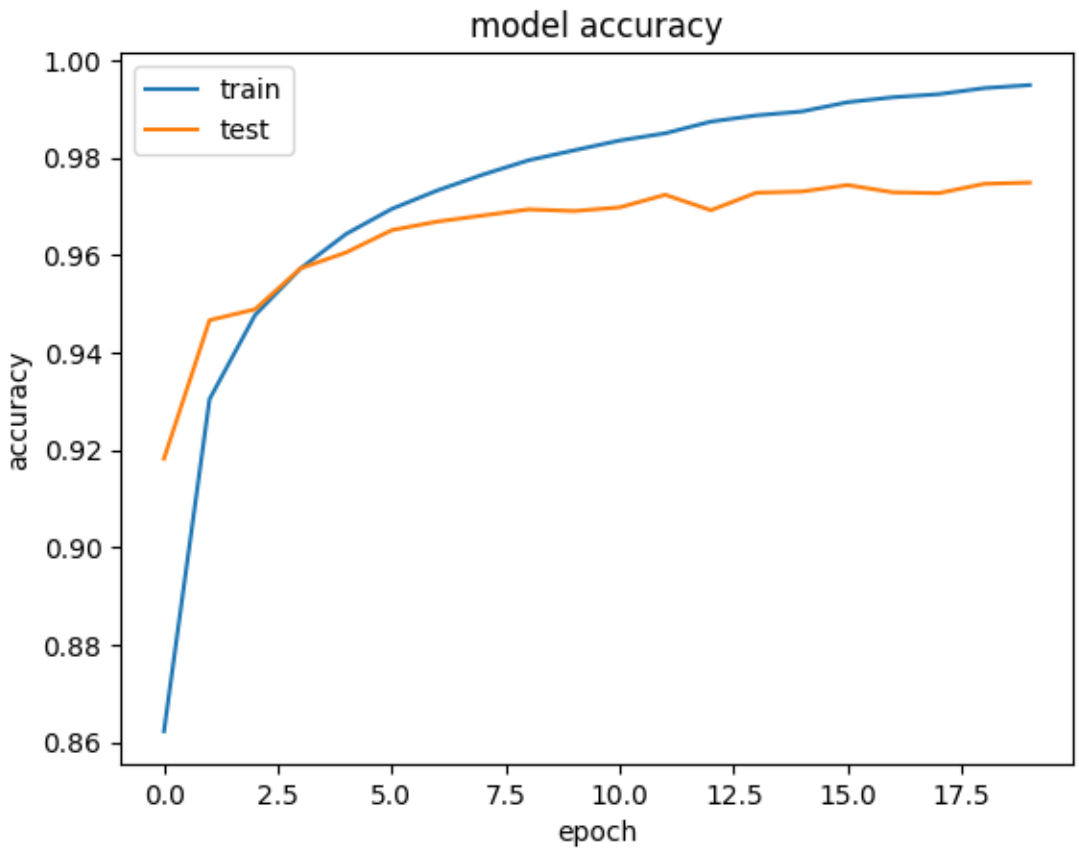
Let's explotre the results both for the score and accuracy values, as well as to visualise the plots of these values during the training.

[15]:
```
#test the network
score = model.evaluate(input_X_test, output_Y_test, verbose=VERBOSE)
print("\nTest score/loss:", score[0])
print("Test accuracy:", score[1])
```

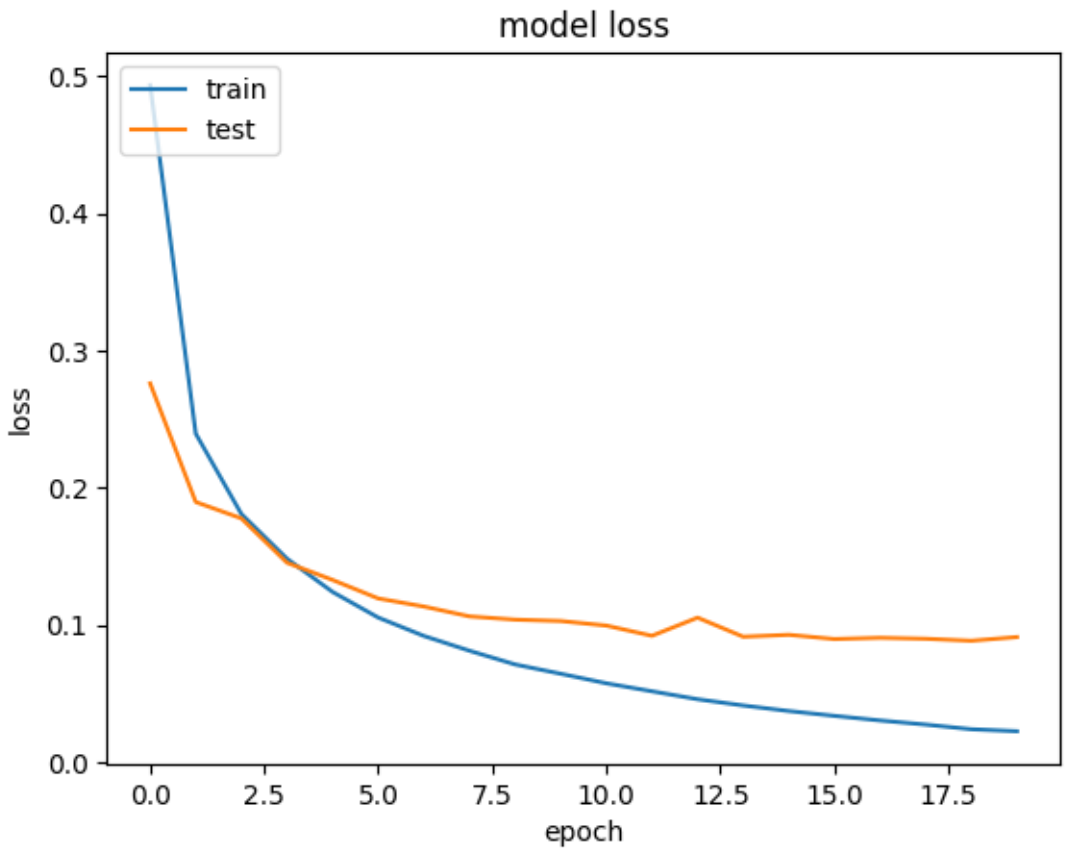
```
313/313             10s 31ms/step -
accuracy: 0.9737 - loss: 0.0880

Test score/loss: 0.07635730504989624
Test accuracy: 0.9765999913215637
```

[16]:
```
# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

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



[17]:
```
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



4 Adding weight droputs

An efficient way to reduce the number of parameters (weights) to be trained, as well as to increase generalisation capabilities, is to randomly remove (i.e. **dropout**) a certain proportion of the nodes at random in each epoch.

[20]:
```
# import the dropout layer type
from tensorflow.keras.layers import Dropout

# Probability of weights dropout
P_DROPOUT = 0.3

# We can increse this parameter afterwards
N_EPOCH = 20

model = Sequential()
model.add(Dense(N_HIDDEN, input_shape=(RESHAPED,)))
model.add(Activation('relu'))
model.add(Dropout(P_DROPOUT))
```

```
model.add(Dense(N_HIDDEN))
model.add(Activation('relu'))
model.add(Dropout(P_DROPOUT))
model.add(Dense(N_CLASSES))
model.add(Activation('softmax'))

# model compilation
model.summary()

OPTIMIZER = SGD(learning_rate=0.1) # Stochastic gradient descent optimiser

# model compilation
model.compile(loss='categorical_crossentropy', optimizer=OPTIMIZER,␣
 ↪metrics=['accuracy'], run_eagerly=True)
```

```
/usr/local/anaconda3/envs/ml-algo/lib/python3.12/site-
packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_5"

Layer (type)	Output Shape	Param #
dense_13 (Dense)	(None, 128)	100,480
activation_13 (Activation)	(None, 128)	0
dropout_2 (Dropout)	(None, 128)	0
dense_14 (Dense)	(None, 128)	16,512
activation_14 (Activation)	(None, 128)	0
dropout_3 (Dropout)	(None, 128)	0
dense_15 (Dense)	(None, 10)	1,290
activation_15 (Activation)	(None, 10)	0

Total params: 118,282 (462.04 KB)

Trainable params: 118,282 (462.04 KB)

```
Non-trainable params: 0 (0.00 B)
```

5 Let's train the multi-layer perceptron network with DROPOUT

Let's now train (fit) the above dropout network with the above-defined batch size (128), and number of epochs (250).

[21]:
```
#train the network
history = model.fit(input_X_train, output_Y_train, batch_size=BATCH_SIZE,␣
↪epochs=N_EPOCH, verbose=VERBOSE, validation_split=VALIDATION_SPLIT)
```

```
Epoch 1/20
375/375             30s 80ms/step -
accuracy: 0.6558 - loss: 1.0794 - val_accuracy: 0.9244 - val_loss: 0.2628
Epoch 2/20
375/375             29s 77ms/step -
accuracy: 0.8909 - loss: 0.3697 - val_accuracy: 0.9434 - val_loss: 0.2006
Epoch 3/20
375/375             27s 73ms/step -
accuracy: 0.9153 - loss: 0.2858 - val_accuracy: 0.9528 - val_loss: 0.1659
Epoch 4/20
375/375             35s 93ms/step -
accuracy: 0.9264 - loss: 0.2466 - val_accuracy: 0.9565 - val_loss: 0.1461
Epoch 5/20
375/375             28s 75ms/step -
accuracy: 0.9357 - loss: 0.2186 - val_accuracy: 0.9609 - val_loss: 0.1340
Epoch 6/20
375/375             28s 73ms/step -
accuracy: 0.9406 - loss: 0.1942 - val_accuracy: 0.9650 - val_loss: 0.1234
Epoch 7/20
375/375             41s 72ms/step -
accuracy: 0.9486 - loss: 0.1768 - val_accuracy: 0.9661 - val_loss: 0.1164
Epoch 8/20
375/375             28s 75ms/step -
accuracy: 0.9490 - loss: 0.1713 - val_accuracy: 0.9684 - val_loss: 0.1107
Epoch 9/20
375/375             29s 78ms/step -
accuracy: 0.9512 - loss: 0.1625 - val_accuracy: 0.9682 - val_loss: 0.1085
Epoch 10/20
375/375             30s 79ms/step -
accuracy: 0.9562 - loss: 0.1493 - val_accuracy: 0.9703 - val_loss: 0.1022
Epoch 11/20
375/375             27s 73ms/step -
accuracy: 0.9597 - loss: 0.1356 - val_accuracy: 0.9710 - val_loss: 0.0980
Epoch 12/20
```

```
375/375             27s 71ms/step -
accuracy: 0.9601 - loss: 0.1296 - val_accuracy: 0.9725 - val_loss: 0.0964
Epoch 13/20
375/375             29s 79ms/step -
accuracy: 0.9626 - loss: 0.1246 - val_accuracy: 0.9729 - val_loss: 0.0926
Epoch 14/20
375/375             30s 80ms/step -
accuracy: 0.9639 - loss: 0.1178 - val_accuracy: 0.9737 - val_loss: 0.0903
Epoch 15/20
375/375             29s 77ms/step -
accuracy: 0.9662 - loss: 0.1122 - val_accuracy: 0.9742 - val_loss: 0.0906
Epoch 16/20
375/375             42s 79ms/step -
accuracy: 0.9663 - loss: 0.1124 - val_accuracy: 0.9743 - val_loss: 0.0888
Epoch 17/20
375/375             28s 74ms/step -
accuracy: 0.9709 - loss: 0.1004 - val_accuracy: 0.9741 - val_loss: 0.0869
Epoch 18/20
375/375             29s 77ms/step -
accuracy: 0.9678 - loss: 0.1012 - val_accuracy: 0.9749 - val_loss: 0.0860
Epoch 19/20
375/375             29s 78ms/step -
accuracy: 0.9693 - loss: 0.0999 - val_accuracy: 0.9749 - val_loss: 0.0854
Epoch 20/20
375/375             29s 78ms/step -
accuracy: 0.9694 - loss: 0.0994 - val_accuracy: 0.9762 - val_loss: 0.0840
```

6 Looking at the results of the trained dropout network

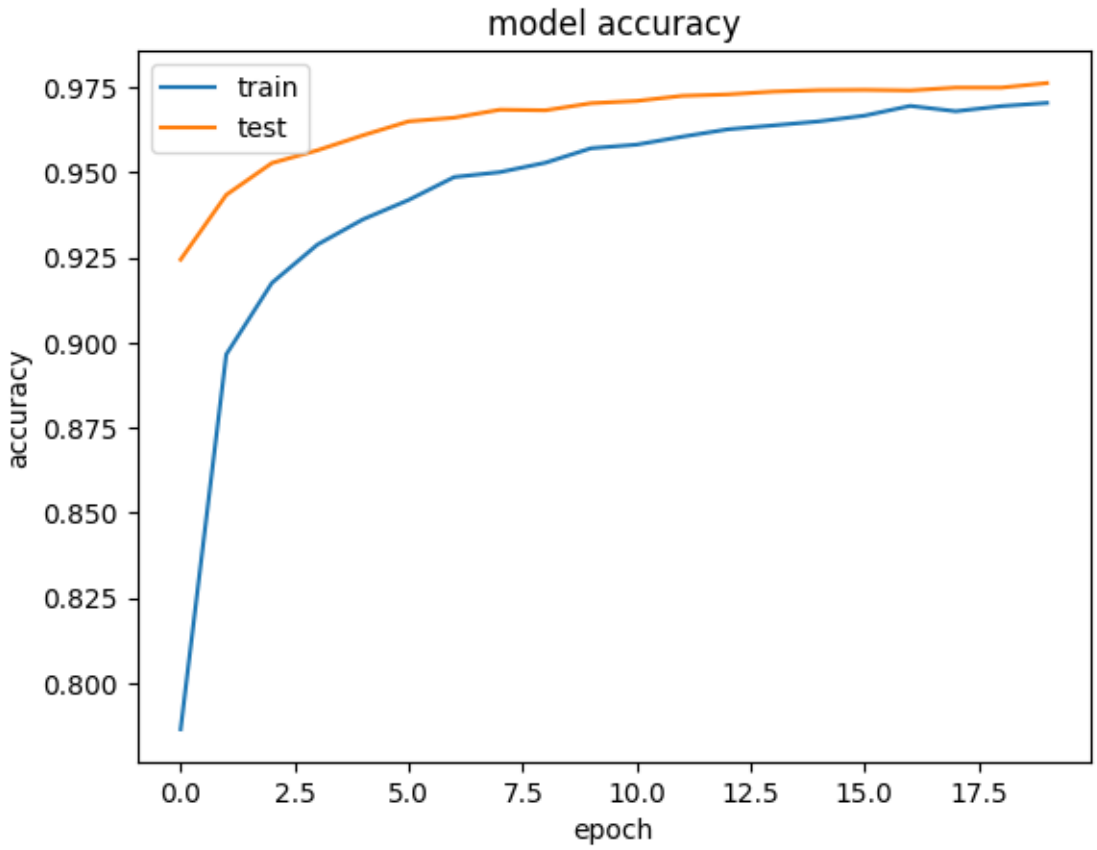
Let's explore the effects of adding the weight dropout on the network performance.

You can check if the dropout has further improved our results (it depends on the dataset and number of hidden neurons).

[22]:
```
#test the network
score = model.evaluate(input_X_test, output_Y_test, verbose=VERBOSE)
print("\nTest score:", score[0])
print("Test accuracy:", score[1])

# list all data in history
print(history.history.keys())

# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
```

```
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

```
313/313             9s 30ms/step -
accuracy: 0.9730 - loss: 0.0929

Test score: 0.08034870028495789
Test accuracy: 0.9771000146865845
dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss'])
```



6.1 Exploring Training Hyperparameters

Homework: You can explore the role of various hyperparameters to see how you can further improve the MLP model's performance on the MNIST dataset.

For example, if you increase the number of epochs for the dropout network to 250, you will see that the test and train accuracy errors will converge (accuracy closer to 97% for both training and test), which means that we have achieved the best tradeoff between training and testing.

You can carry out many additional simulations on hypeparameter exploration where you can try for example:

- different number of epochs
- different learning rate
- different number of hidden nodes
- different proportion of dropout rates
- different optimisers in addition to SGD (e.g. RMSprop, Adam)
- different batch size

6.2 Conclusions

With this tutorial we have practiced the training of both a Simple Perceptron, and a Multi-Layer Perceptron, with a benchmark dataset containing images of handwritten numbers. This helped us understand how to load the datase, visualise it, and visualise the training history and the effects of adding hidden layers and then adding weight dropout.