**Handout - *Microcomputers in Technical Applications (part 2 Image Processing and Artificial Intelligence) – V1.6***

by Chris Hendriks (hendrikschris@yahoo.com)

1. **Introduction**

For the experiments in this workshop we will use the Raspberry Pi 3B+. Other platforms or models can be used. Generally, this will only affect the values for the processing time.

We will use Python as programming language. But within Python we will make use of Keras, an open-source neural network library written in Python. Next to the classes and functions Keras also contains a number of datasets. A dataset that we will use is the MNIST dataset which contains 60,000 training images and 10,000 validation images of handwritten digits (the numbers 0 up to 9).

Keras uses in its turn Tensorflow as an underlying code library (it can also use another library). To improve processing speed the core of Tensorflow is written in highly-optimized code (mainly C++).

For reasons of performance (especially when working on the Raspberry Pi Zero) we will work with a text editor ‘geany’ that behaves as a Python development environment (it has some features that make it especially suitable for writing Python programs). We will run the programs in the shell with the python3-command:

pi@raspberrypi:~ $ python3 program.py

Try some small python programs to get familiar with the text editor and the shell execution.

For making predictions using Neural Networks we go through the following steps:

1. Building a trained network:
2. Define network – how many layers of what type and what size.
3. Compile network – the model is prepared for the matrix operations so that it can be trained.
4. Train network – the weights, biases and other parameters are determined by offering inputs, calculating outputs, comparing the calculated output with the correct output and adjusting parameters (weights and biases) based on a particular backpropagation algorithm.
5. Evaluate network – with a testing set consisting of inputs and outputs the network is evaluated; we have to decide whether the quality of prediction is acceptable.
6. Use and/or store the trained model.
7. Make predictions:
8. Build or load the trained model.
9. Pre-process the data so that it is available in a form suitable for the network.
10. Offer the data to the model and obtain the prediction (knowing that there is a certain percentage of incorrect predictions). The only way to give a qualification to the prediction we get is to compare its percentage of probability with the percentages of the alternatives.

Building a trained network is usually done only once (or a limited number of times). It can be done on a different computer (usually a faster one). There is no limit to the number of times the trained model is used or the number of computers it is used on.

1. **Import modules**

We start importing some libraries and modules that we need:

from keras.datasets import mnist

from keras.models import Sequential

from keras.layers import Dense

from keras.utils import np\_utils

import matplotlib.pyplot as plt

import numpy as np

print (np.\_\_version\_\_)

1. As was mentioned before we use the mnist dataset of the keras datasets (the keras library has more of these ‘ready-made’ datasets).
2. Our models will be of the type ‘Sequential’. That means that the subsequent layers feed their output only in the next layer.
3. The first model we will build consists of one hidden layer of the type ‘Dense’ (that means fully connected to the previous and subsequent layer).
4. We will make use of np\_utils from keras.utils. This module consists of a collection of utilities (we will use, for instance, the matrix conversion utility later on).
5. For plotting images we will use the matplot library.
6. In our programs we will make use of NumPy. Numpy (short for NUMirical PYthon) is an extension of Python specifically designed for numerical operations. We will use it for its operations on matrices.

We can check whether the import has been processed successfully by asking the version of the imported module. Even when we don’t know what version we are supposed to be using, the execution of this statement shows that the module has been imported properly. However, only the library or module itself has an attribute ‘version’ – not the functions or classes in the module.

Import the various modules and check the version of numpy.

1. **Numpy arrays**

import numpy as np

a=np.array([1,2,3,]) # creates an np array with rank 1 and values 1, 2 and 3

print (a[2]) # prints ‘3’

print (a.shape) # prints ‘(3,)’

print(a[0]) # prints ‘1’

b=np.array([[1,2,3],[3,4,5]])

print (b[1,0]) # prints ‘3’

print(b.shape) # prints ‘(2,3)’

# we can create an array filled with zero:

d=np.zeros((2,2), dtype=int) # d is a 2 by 2 array with all elements being 0

print (d[0,1]) # prints ‘0’

# we can create an empty array or an array filled with a specified value

e=np.empty((2,3)) # an empty array with 2 rows and 3 columns

 # the data type ‘int’ is included; default is float

f= np.full((2,3),7) # an array filled with the value 7

b=np.array([[1,2,3],[3,4,5]])

b=b.reshape(6) # reshapes b into an array with rank 1

print(b) # prints ‘[1 2 3 3 4 5]’

b=b.reshape(3,2) # reshapes b into an array with rank 2 (

print(b) # prints ‘[[1 2]

 [3 3]

 [4 5]]’

a=np.array([[0,1,2],[3,4,5]])

print(type(a)) # prints

# Try the following:

a=np.array([[1,2,3,4,5],[11,12,13,14,15]])

b=a

for i in range (5):

 b[0,i]=a[0,(4-i)]

print(' array a: ','\n',a)

print(' array b: ','\n',b)

a=np.array([[1,2,3,4,5],[11,12,13,14,15]])

b=a.copy()

for i in range (5):

 b[0,i]=a[0,(4-i)]

print(' array a: ','\n',a)

print(' array b: ','\n',b)

# what is your conclusion?

In Keras, as in most AI software based on Python, we use numpy arrays. Numpy is a library for scientific computing in Python. A numpy array is a grid of values (in the computer memory simply a series of values stored in subsequent memory locations) with a tuple[[1]](#footnote-1) of non-negative integers associated to it indicating the shape (the shape can be printed – in that case the tuple is printed).

To indicate the n-dimensionality the name ndarray is sometimes used. The numpy array object has – apart from the array itself and the tuple – a number of functions associated to it. We will have a look at some of them

When referring to a single element of an np array the array-name is followed by a pair of square brackets with the position of the element in between.

When calling an ‘np.array’ function, it is always followed by a pair of round brackets – like all functions.

Within these brackets one can find square brackets – [ ] – to transfer content values as arguments of the function or round brackets for other parameters (the brackets are in that case part of the parameter).

So in <array\_name> followed by [number] ´number´ stands for the index;

Equally: [number1,number2] gives the indexes of a 2-dimensional array.

When creating an array object we can pass on its content by values in between []. This content is a parameter of the instantiation of the array class.

In all other situations we use parentheses or ´round brackckets´.

Try the examples given in box with some different values. Take care to understand the use of the various brackets.

We can reshape arrays as long as the total number of elements remains the same (see examples in the box).

With the ‘type’ function (this is a general python function that can be used for any variable) we can ask for the array type,

We can ‘slice’ a section from an array by indicating for every dimension from where to where the slice goes. The boundaries are separated by a colon. Whenever the ‘from’ is the beginning or the ‘to’ is the end of the array dimension, it can be omitted.

import numpy as np

original=np.array([[[0,1,2,3],[4,5,6,7],[8,9,0,1]],[[10,11,12,13],[14,15,16,17],[18,19,10,11]]])

print('The shape of the original array;')

print(original.shape)

print('The original array;')

print(original)

print('')

sliced1=original[0:1,:,:]

print('The shape of the first sliced array;')

print(sliced1.shape)

print('The first liced array;')

print(sliced1)

print('')

sliced2=original[:,:1,:2]

print('The shape of the second sliced array;')

print(sliced2.shape)

print('The second liced array;')

print(sliced2)

*Output:*

The shape of the original array;

(2, 3, 4)

The original array;

[[[ 0 1 2 3]

 [ 4 5 6 7]

 [ 8 9 0 1]]

 [[10 11 12 13]

 [14 15 16 17]

 [18 19 10 11]]]

The shape of the first sliced array;

(1, 3, 4)

The first liced array;

[[[0 1 2 3]

 [4 5 6 7]

 [8 9 0 1]]]

The shape of the second sliced array;

(2, 1, 2)

The second liced array;

[[[ 0 1]]

 [[10 11]]]

The box gives some examples.

Try some different values for the size and shape of the array. Try also three-dimensional and four-dimensional arrays.

Print some elements and some slices of the arrays.

Subsets of a numpy array can be specified by entering only the array value in one dimension (or a limited number of dimensions). Get some familiarity with this concept using the code in the box.

import numpy as np

a=np.array([[[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]],[[[51,52,53],[54,55,56]],[[57,58,59],[60,61,62]]]])

print(a.shape)

print(a)

print ('print(a[0])')

print(a[0].shape)

print(a[0])

print ('print(a[0,:,0])')

print(a[0,:,0].shape)

print(a[0,:,0])

print ('print(a[0,:,0,:])')

print(a[0,:,0,:].shape)

print(a[0,:,0,:])

When a slice is specified the subset is made into a new object.

1. **Images and matrices**

An image is in fact a matrix of pixels. The matrix is of rank 2 (rows and columns) if it is a greyscale image. If it is a color image there is a third dimension for Red, Green and Blue.

import matplotlib.pyplot as plt

import numpy as np

im=np.full((28,28),100)

plt.imshow(im,cmap='gray',vmin=0,vmax=255)

plt.show()

We can plot an image using the Pyplot function of ‘matplotlib’.

Try the example in the box (the program halts with the display of the image; terminating the display makes the program continue). The parameter *cmap='gray'* tells the plotfunction to plot in grey-scale. When leaving out this parameter colors are being used for the various shades of grey.

The parameter *vmin=0,vmax=255* forces a specific range being used – in this case 0 is for black and 255 for white. If not specified the minimum value in the matrix will be used for black and the maximum value for white.

Try the following images:

plt.subplot(221)

plt.imshow(im1,cmap='gray',vmin=0,vmax=255)

text='exercise 4c'

plt.title(text)

plt.subplot(222)

plt.imshow(im2,cmap='gray',vmin=0,vmax=255)

text='exercise 4d'

plt.title(text)

plt.show()

1. An all-white image and an all-black image
2. An all-white image using a matrix with dimensions 28 x 20 (28 rows and 20 columns); as you will notice the rows in the matrix correspond to the vertical axis in the plotted image and the columns in the matrix correspond to the horizontal axis in the plotted image
3. An all-white image (28x20) with a black spot in the top righthand corner

im1 and im2 are images that have been loaded or composed

1. An all-gray image (28x28) (try light and dark gray)
2. An all-white image (28x28) with a dark-gray vertical narrow line at position 5 (horizontal) and a fat light-gray line at position 19 (vertical).

Text can be added to the image. We can also plot a number of images in the same diagram (see the code in the box).

1. **Data for training and evaluation**

We will train the model to recognize handwritten digits. For this purpose, we load the following numpy arrays:

* The input training data X\_train (60,000 images)
* The actual digits shown by the training data (y\_train; the digits shown on the 60,000 images)
* The input testing data X\_test (10,000 images)
* The actual digits shown by the test data (y\_test; the digits shown on the 10,000 images)

Let’s have a look at an image. With the plot function from the matplot library (matplotlib.pyplot has been imported as plt) we can get the image on the screen. The images are grayscale.

(X\_train,y\_train),(X\_test,y\_test)=mnist.load\_data()

plt.imshow(X\_train[0])

plt.show()

Plot some of the input images. Take images of the train and the test dataset.

(X\_train,y\_train),(X\_test,y\_test)=mnist.load\_data()

plt.imshow(X\_test[234],cmap='gray',vmin=0,vmax=255)

text='The image shows the digit '+str(y\_test[234])

plt.title(text)

plt.show()

The outputs corresponding to the images are scalars (y\_train and y\_test are arrays of scalars).

Print the digits corresponding to the images.

The data we load with mnist.load\_data() need some adjustments before they can be fed into the NN model:

1. The input matrix has to be reshaped from rank 3 (images, pixels x-axis, pixels y-axis) to a rank 2 (images, number of pixels).

num\_pixels=X\_train.shape[1]\*X\_train.shape[2]

X\_train=X\_train.reshape(X\_train.shape[0],num\_pixels)

X\_train=X\_train.astype('float32')

X\_train/=255

y\_train\_c=np\_utils.to\_categorical(y\_train)

X\_train.shape[0] is the number of images (60,000);

X\_train.shape[1] is the number of pixels in the x direction;

X\_train.shape[2] is the number of pixels in the y direction.

1. X\_train consists of integers in the range 0 – 255. We convert them into float(32) in the range 0 to 1.0.
2. y\_train has to be converted into ‘one-hot’ arrays. Since we want to detect decimal integers, we define a so called one-hot output layer; if the training data is given as numbers 0 up to 9, these scalars have to be converted in one-hot vectors: 0 will be converted into 1000000000, 1 into 0100000000, etcetera. The example in the box will give the following output:

import numpy as np

from keras.utils import np\_utils

a=np.array([5,2,3,7])

a=a.reshape(2,2)

a=np\_utils.to\_categorical(a,num\_classes=10)

print (a)

[[[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]

 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]]

 [[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]

 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]]]

The adjustments made to X\_train and y\_train also have to be made to X\_test and y\_test. In later experiment we need the original array after we have modified it. In those cases we can make a copy of the original y\_test by using y\_test\_original=y\_test.copy ().

Exercise: a is an np.array; write a small program to find out the difference between b=a and b=a.copy()

1. **Building the basic NN model (define and compile network)**

The basic neural network (NN) consists of an input layer, a hidden layer and an output layer. The input layer consists in this workshop of images of 28 by 28 pixels. Since we are detecting digits, the output layer consists of the numbers 0 up to 9. The number of neurons in the hidden layer can have any value – although very small values don’t make sense.

In the basic NN the hidden layer and the output layer are fully connected to the respective previous layers. That means that every neuron has an array of weights and a bias related to it. During forward propagation the values of every next layer – array – is calculated by multiplying the previous layer with the weights and adding the (negative) bias. Most of the matrix processing is done inside Keras objects but some operations we have to initiate ourselves.

We have to take care of the following steps:

1. Specify the input layer in the proper format, that is the format required by the first hidden layer. Since the first hidden layer is a fully connected layer – it will be called a *Dense* layer, it expects a one-dimensional array (as if it was connected to a previous dense layer. In our case the number of inputs is equal to the number of pixels. We usually have to reshape the input of the two-dimensional image into the one-dimensional array (in our examples a 28x28 image is reshaped into an array with one row of 784 elements).
2. Define the hidden layers with respect to the number of neurons.
3. Since we want to detect decimal integers, we define the output as a so called one-hot layer.

The layers in a NN model are ordered sequentially (the output of one layer is the input of the next layer). We start specifying this model shape: model=Sequential(). This is an empty model so we add layers one-by-one.

The first layer has the images as input – one image after the other is offered to the model. In the case of a fully connected NN the layers have the type ‘Dense’. The Dense layers have the following parameters:

* Number of neurons – we can choose any number; larger numbers offer more accuracy but take also more processor time.

model=Sequential()

model.add(Dense(64,input\_dim=num\_pixels,activation='relu'))

model.add(Dense(10,activation='softmax'))

model.compile(loss='categorical\_crossentropy',optimizer='adam',metrics=['accuracy'])

model.summary()

* The first layer has to be told how many inputs it is connected to; the subsequent layers deduce the number of inputs from the previous layer. The input dimension for the first layers is equal to the number of pixels of the images, which is equal to 28\*28 or X\_train.shape[1]\*X\_train.shape[2] (these two variables give the x and y co-ordinate of the original images).
* The activation function tells when a neuron is activated. The most common activation function is ‘relu’: rectified linear unit: the output of the function is 0 when the input is negative and equal to the input when the input is positive.
* Softmax is another activation function: the outputs sum up to 1 so can be interpreted as probabilities. The model can make a prediction based on which option has a higher probability.

Now that the model has been built, we have to compile it, so that it is translated into a convenient form to be trained, i.e. to calculate the weights and biases.

For compilation the following parameters are required:

* loss=’categorical\_crossentropy’: the loss is a measure for the difference between the output that the model produces with a specific input image and the actual output corresponding to that image. In other words: when an image of the number 5 is offered at the input the model generates the chance that it is a ‘0’, a ‘1’ and so on up to a ‘9’. It is classified into the 10 categories corresponding to the 10 digits. The correct value is a ‘0’ for all the outputs except for the output of the digit ‘5’ – that one should be 1. There are various ways to calculate the loss. Crossentropy is a logarithmic formula that results in a rapidly larger contribution to the loss if the deviation gets larger.
* Optimizer=’adam’. The optimizer is the method of calculating the (adjustments of the) weights and biases based on the calculated loss. ‘Adam’ is a popular method to do this calculation. There are in fact two phases in a complete cycle: the forward propagation (with an input the subsequent neuron values are calculated based on the actual weights and biases and resulting in the value for the loss) and the backwards propagation (from the loss the new values for weights and biases are calculated from the last layer back to the first). ‘Adam’ is one of the algorithms for the calculations in the backwards propagation.
* metrics=['accuracy']. A metric is a function that is used to judge the performance of the model. With metrics measurement based on ‘accuracy’ the total correct predictions are divided by the total number of predictions. In other words, it shows the fraction that was predicted correctly.

Use model.summary() to display an overview of the model (‘none’ in ‘Output Shape’ means that ‘unspecified’ and that it can vary with the size of the input. Verify the number of parameters[[2]](#footnote-2)

1. **Training and evaluation of the model**

X\_train=X\_train[0:10000,:28,:28]

y\_train=y\_train[0:10000]

# reshape arrays

model.fit(X\_train,y\_train, epochs=5,batch\_size=100,verbose=2)

We will take a subset of the 60,000 images in order to reduce training time (of course we sacrifice model accuracy in this way).

The parameters when training the model (model.fit) are:

* training data (X\_train,y\_train)
* epochs: the number of complete training cycles with the full data set
* batch\_size = the number of forward propagations (=training images) before the start of the backward propagation (= calculation of weights and biases) based on the averaged loss.
* verbose=2 determines how the result is presented.

We can evaluate the model with the evaluate-function that returns an array with a value for the loss and a value for the accuracy (= correct digits / total number of digits).

score=model.evaluate(X\_test,y\_test,verbose=2)

print ('validation loss: ',round(score[0],4))

print ('validation acc: ',round(score[1],4))

We can obtain the predicted values with the function predict\_classes. By comparing them with the digits shown on the images offered at the input, we can get an idea of the errors made by the network.

pr=model.predict\_classes(X\_test)

print(pr[0:20])

print(y\_test[0:20])

j=0

while (pr[j]==y\_test\_original[j]):

 j=j+1

print(pr[j],y\_test\_original[j])

plt.imshow(X\_test[j].reshape(28,28),cmap='gray')

plt.show()

Display some incorrect predictions and see whether also a human might have made the same mistake[[3]](#footnote-3).

The model will always give a predicted digit. But how reliable is the prediction? An indication for the reliability of the prediction is given by the relative score the prediction gets. The code in the box gives a way to get this score printed. (model.predict(X\_test[8].reshape(1,784)) gives in an array the scores of all the possible outputs).

pr=model.predict(X\_test[8].reshape(1,784))

for i in range(10):

 print(i,round(100\*pr[0,i],4))

Summarizing the various ways to get an idea of the quality of predictions:

* ‘score’ gives a value for the overall quality of a test set;
* ‘predict\_classes’ gives a prediction for a set of test images; we can have a look at individual ‘missers’;
* ‘predict’ gives a prediction for an individual image; by comparing the percentages of the probabilities that were obtained we can get an idea of the quality of the prediction (if the predicted value has a high probability – say more than 99 % – then the result is likely to be correct; if the probabilities of some digit are more or less the same than one can rightfully have doubts.

So, the first two methods are for situations that the correct value(s) is/are known. The last one is for situations that this is not the case.

Take an image that results in the correct prediction. Measure the probability for the possible outputs. Now delete the pixels of any column by making them black and measure the probability again. Repeat this ‘degeneration’ of the original image until the model does not predict the correct digit anymore.

Summarizing: the ´lifetime´ of an AI network can be considered as consisting of the following phases: training, testing/validating and production.

During *training* the accuracy shows the fraction that was predicted correctly. Also, the average loss is a measure for the quality.

During t*esting* again the average loss and average accuracy are quality indicators. The evaluate function gives the overall score of the two indicators. We can obtain the incorrect predictions with the function predict\_classes. By comparing them with the digits shown on the images offered at the input, we can get an idea of the errors made by the network.

During p*roduction* comparing the percentages of the probabilities that were obtained for all possible outputs gives an indication of the quality of an individual.

1. **Convolutional NN**

Store a copy of the basic NN network (we will use the basic NN network later for comparison).

from keras.datasets import mnist

from keras.models import Sequential

from keras.layers import Dense,**Conv2D,MaxPooling2D,Flatten**

from keras.utils import np\_utils

(X\_train,y\_train),(X\_test,y\_test)=mnist.load\_data()

X\_train=X\_train[0:10000,:28,:28]

y\_train=y\_train[0:10000]

num\_classes=10

X\_train=**X\_train.reshape(X\_train.shape[0],28,28,1)**

X\_train=X\_train.astype('float32')

X\_train/=255

X\_test=**X\_test.reshape(X\_test.shape[0],28,28,1)**

X\_test=X\_test.astype('float32')

X\_test/=255

y\_test=np\_utils.to\_categorical(y\_test)

y\_train=np\_utils.to\_categorical(y\_train)

model=Sequential()

**model.add(Conv2D(16,(3,3),input\_shape=(28,28,1),activation='relu'))**

**model.add(MaxPooling2D(pool\_size=(2,2)))**

**model.add(Flatten())**

model.add(Dense(64,activation='relu'))

model.add(Dense(num\_classes,kernel\_initializer='normal',activation='softmax'))

model.compile(loss='categorical\_crossentropy',optimizer='adam',metrics=['accuracy'])

model.summary()

model.fit(X\_train,y\_train, epochs=5,batch\_size=100,verbose=2)

score=model.evaluate(X\_test,y\_test,verbose=2)

print ('validation loss: ',round(score[0],4))

print ('validation acc: ',round(score[1],4))

We will convert the basic NN network into a convolutional network. The adjustments will be discussed and are printed bold in the program code in the box.

We can make our basic NN into a convolutional NN by adding one or more convolutional layers. A convolutional layer multiplies subsequent subsets of the initial image with a matrix consisting of weights (in our example code the matrix is 3x3). In that way certain shape elements can be detected independent of their position. The way the matrix is filled up is determined by the back-propagation software (initially it is filled with random values). A number of patterns can be detected in this way (in the code example there are 16 matrices like this).

With the MaxPooling layer the ‘highlights’ are detected in the resulting matrices and its output is made into an array by ‘Flatten’ so that it can be fed into the ‘Dense’ layer.

**Conv2D, MaxPooling2D and Flatten** are modules within Keras (keras.layers to be precise) – they have to be imported and added as layers to the sequential model.

Convolutional layers expect n-by-m matrices as their input. The images can be color but since our input images are greyscale there is only a single layer. Therefor the input images are reshaped to (number\_of\_images, **28,28,1**). As mentioned before, the dense layer expects a ‘flat’ one-dimensional array of input neurons. Therefor the layer ‘Flatten’ is included. As you can see: we always have to take care that we offer a properly shaped array to a layer. The input layer usually has to be reshaped and when going from convolutional/maxpooling to dense or from dense to convolutional reshaping has to be included.

Have a summary of the model printed and explain the various values (note: the number of parameters of a conv2D layer is equal to (number of elements of each filter + bias)\*(number of channels) = ((height of the filter)\*(width of the filter)+1)\*(number of channels).

Try the code for the resulting convolutional NN. The computation will take a bit more time.

Train the basic NN and the convolutional NN with a training set of 30000 images. Evaluate the results. Note that not every evaluation of the same network produces the same results – this is because the layers are initially filled with random values.

The trained model can be stored and loaded later again (in the box the code for storing and loading the model).

from keras.models import load\_model

model.save('model\_conv\_nn.h5')

new\_model=load\_model('model\_conv\_nn.h5')

Store the model you have obtained and load it in a new program for evaluation. The function ‘load\_model’ is included in ‘keras.models’ – it has to be imported. The name ‘model’ is the name that was used when defining the network – it could have been any name; also the names ‘'model\_conv\_nn’ and ‘new\_model’ are arbitrarily chosen. The extension ‘h5’ refers to the HDF5 format, the format used to store the weights and biases of the model. Evaluate ‘new\_model’.

**Loading a pre-trained network**

from keras.utils import plot\_model

from keras.models import load\_model

model1=load\_model('pretrained\_model1.h5')

plot\_model(model1, to\_file='model1.png')

model1.summary()

In the previous paragraph we have stored a trained model. This model – with all the weights, biases and network structure in it – can be loaded again. In a similar way a model trained by others (from internet or elsewhere) can be loaded.

Two pre-trained models are stored on the Raspberry (model1 and model2). Load model1 and store a plot of the image of the model – when the program in the box is run the plot of the model is stored in the working directory of the pi. Have a look at the model and link it to the print of the summary.

Evaluate the model with the test data from the MNIST dataset (see code in the box; do not forget to import the necessary modules and adjust the format of the test data).

model1=load\_model('model1.h5')

eval\_data=model1.evaluate(X\_test,y\_test,verbose=2)

print('error is: ',round(100- eval\_data [1]\*100,2),' %')

The evaluation gives the percentage of incorrect predictions in relation to the total number of predictions.

Compare the result with the accuracy of the other pretrained model as well as the models you compiled yourself.

:

y\_test\_original=y\_test

y\_test=np\_utils.to\_categorical(y\_test)

:

model1=load\_model('model1.h5')

pr=model1.predict\_classes(X\_test)

i=0

list\_incorrect=[]

for i in range (10000):

 if (pr[i]!=y\_test\_original[i]):

 list\_incorrect.append(i)

print('total number of incorrect predictions = ',len(list\_incorrect))

for i in range (5):

 plt.imshow(X\_test[list\_incorrect[i]].reshape(28,28),cmap='gray')

 text=('image shown '+str(y\_test\_original[list\_incorrect[i]])+'; predicted '+str(pr[list\_incorrect[i]]))

 plt.title(text)

 plt.show()

We can also make a list of all predicted digits and compare those with actual digits offered. The box gives an example.

Notes:

- before we convert y\_test to categorical we make a copy of the original array of digits for later use;

- we got back an object ‘model1’ as an incarnation of the class load\_module;

- the function ‘predict\_class’ gives us an array of all predicted digits;

- we enter the incorrect predictions in a list;

- the function len() gives the length of a list;

- the first five incorrect digits are shown with text.

Have a look at some of the incorrect predictions (you can have a look at more incorrect predictions by taking a longer or different range)

1. **Visualizing the accuracy of a model during training**

When training a model, a data object is returned that contains information about the accuracy and loss of the training result

history=model.fit(X\_train,y\_train, validation\_split=0.2,epochs=10,batch\_size=16,verbose=2)

plt.plot(history.history['acc'])

plt.plot(history.history['val\_acc'])

plt.title('Model accuracy')

plt.ylabel('accuracy')

plt.xlabel('epoch')

plt.legend(['Train','Test'],loc='upper left')

plt.show()

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()

In the box the code is given for obtaining the data object and plotting the accuracy and loss during the subsequent training steps. The parameter ‘validation\_split=0.2’ tells the training object to take 20 % of the input data for evaluation of the model at the end of each epoch (the last 20 % is being taken; any fraction can be specified). Alternatively we could specify a separate dataset for testing with validation\_data=(X\_test,y\_test). In both ways we have ‘training’ and ‘evaluate’ in a single run – we could have used ‘model.fit’ and ‘model.evaluate’ separately as well.

Include the code in the box in the first model you compiled (the model with two dense layers). Run the program and explain the results.

Build the same network twice. Train and test both networks network and plot the accuracy results in a single diagram (the code in the box might be of help). Why is the resulting accuracy not exactly the same?[[4]](#footnote-4) Why do we have to build and compile the network twice – can the experiment also be performed by running the model.fit function two times on the same network?

model=Sequential()

model.add(Dense(64,input\_dim=num\_pixels,activation='relu'))

model.add(Dense(10,activation='softmax'))

model.compile(loss='categorical\_crossentropy',optimizer='adam',metrics=['accuracy'])

history=model.fit(X\_train,y\_train,validation\_split=0.2,epochs=10 ,batch\_size=100,verbose=2)

model1=Sequential()

model1.add(Dense(64,input\_dim=num\_pixels,activation='relu'))

model1.add(Dense(10,activation='softmax'))

model1.compile(loss='categorical\_crossentropy',optimizer='adam',metrics=['accuracy'])

history1=model1.fit(X\_train,y\_train, validation\_split=0.2,epochs=10,batch\_size=100,verbose=2)

plt.plot(history.history['acc'])

plt.plot(history.history['val\_acc'])

plt.plot(history1.history['acc'])

plt.plot(history1.history['val\_acc'])

plt.title('Model accuracy')

plt.ylabel('accuracy')

plt.xlabel('epoch')

plt.legend(['Train1','Test1','Train2','Test2'],loc='upper left')

plt.show()

Build a fully connected NN model with three dense layers (two relu and the softmax layer. Choose suitable values for the training set and the various parameters. Run the ‘fit’-function and consider the resulting plot for the accuracy (or the loss). What strikes you?

The difference in accuracy during training and testing is caused by the fact that this simple network adapts its parameters to the specific characteristics of the training set. This effect is called overfitting. Overfitting can be reduced by including a *dropout layer* – model.add(Dropout(0.3)) – after the first or the second dense layer. The dropout layer sets randomly (with the specified probability) the input to the next layer to zero. In that way the network is forced to make calculations with fewer neurons and in that way less dependent on the characteristics of the input data. It has the effect of simulating a large number of networks with different network structures and, in that way, making nodes in the network more robust to the inputs.

Include the dropout layer and plot the results of the two networks (with and without dropout layer) in the same diagram. Does the inclusion of the dropout layer reduce overfitting in this case[[5]](#footnote-5)? Try also training the network with a lager number of epochs (e.g. 20).

In general, the effect of including a dropout layer to reduce overfitting becomes more visible when the training set is larger and when the network is more complex (including a convolutional layer). In appendix 1 the effect of including a dropout layer is shown[[6]](#footnote-6). The code shows the two networks the evaluation is based on. Try to explain the various characteristics[[7]](#footnote-7).

1. **Visualizing filters and feature maps of a trained convolutional network**

We can get all kind of information about a trained (convolutional) network. In this paragraph we will have a look at the filters and feature maps of a convolutional layer of a network. We will use predefined\_model1 for this exercise.

import matplotlib.pyplot as plt

from keras.models import load\_model

from keras.utils import plot\_model

model1=load\_model('pretrained\_model1.h5')

layer=model1.layers[0]

weights=layer.get\_weights()

print(layer)

print(weights)

im1= weights[0][:,:,:1,0:1]

im1=im1.reshape(5,5)

plt.subplot(221)

plt.imshow(im1)

im2= weights[0][:,:,:1,1:2]

im2=im2.reshape(5,5)

plt.subplot(222)

plt.imshow(im2)

im3= weights[0][:,:,:1,2:3]

im3=im3.reshape(5,5)

plt.subplot(223)

plt.imshow(im3)

im4= weights[0][:,:,:1,3:4]

im4=im4.reshape(5,5)

plt.subplot(224)

plt.imshow(im4)

plt.show()

We have seen already before how we can get the plot of the network as well as the summary of the layer characteristics. The weights can be printed or plotted (see the code in the box at the right). Note that ‘weights’ is a list consisting of 2 elements: weights[0] contains the actual weights and weights[1] contains the biases. Each weight-array consists of 30 5x5 matrices of 1 color (grayscale) – so the dimension is: [5,5,1,30][[8]](#footnote-8).

To visualize the first convolutional layer for a specific image (the first activation map or feature map corresponding to that image) we are going to define a new network with the same weight matrices and biases as the original one but with the output of the first layer as our model output. For this purpose we import ’Model’ from ‘keras.models’. ‘Model’ allows us to define a model equal to another model but with a different output (see the code in the box below). Using any image from the mnist data set we can get a plot of the feature set of the convolutional layer. When running the code in the box below, the original image and two images of the feature map are being shown.

* Extend the program so that it shows 6 images of the first feature map
* Extend the program so that it shows 6 images of the feature map of the second convolutional layer. What do you notice?
* Run m1.summary() to display the structure of the new model. What do you notice?

import matplotlib.pyplot as plt

from keras.datasets import mnist

from keras.models import load\_model,Model

(X\_train,y\_train),(X\_test,y\_test)=mnist.load\_data()

im=X\_test[10].copy()

im=im.reshape(1,28,28,1); im=im.astype('float32'); im/=255

model=load\_model('pretrained\_model1.h5')

m1=Model(inputs=model.inputs,outputs=model.layers[0].output)

fm=m1.predict(im)

im1=im.reshape(28,28)

plt.subplot(221); plt.imshow(im1,cmap='gray')

plt.subplot(223); plt.imshow(fm[0,:,:,0],cmap='gray')

plt.subplot(224); plt.imshow(fm[0,:,:,1],cmap='gray')

plt.show()

1. **Sensitivity analysis**

In order to get some idea of the sensitivity of the model for abnormalities in the image we will do some tests. Select a single image and evaluate it using any of the trained models. Be aware that there will always be a prediction when running the predict function and that a correct prediction is not a prove that the model can handle the defected image. Check with the predict function whether the prediction is sufficiently reliable (accept a prediction only when the related class has a probability percentage of – for instance – 98 %).

Do the following experiments:

1. Greyscale inversion: invert the image in its gray scale; white becomes black, etc. and evaluate it. What is your conclusion?

for i in range (28):

 for j in range (28):

 array\_image[i,j]=1.0-array\_image[i,j ]

1. Grayscale range: reduce the range of the greyscale by multiplying the image array with a certain factor and evaluate it. What is your conclusion?

factor = 0.5

array\_image \*=factor

1. Rotation: turn the image 90, 180 and 270 degrees and evaluate it. What is your conclusion?

array\_rotated=np.empty((28,28))

for i in range (28):

 for j in range (28):

 array\_rotated[i,j]=array\_image[(j),(27-i)]

plt.imshow(array\_rotated,cmap='gray',vmin=0,vmax=1.0)

plt.show()

How could you train a model so that it recognizes numbers in all positions?

1. Noise: add noise to the picture using a random generator to add a number of grey pixels to the image (in the box a sample code is shown; to run this code add ‘import random’).

number\_of\_pixels=20

for k in range (number\_of\_pixels):

 i=random.randint(0,27)

 j=random.randint(0,27)

 array\_image[i,j]=random.random()

1. Horizontal and vertical position of the digit in the image. Take any digit from the test set and shift the image of the digit to the left (see code in the box; n is the number of pixels that the digit shifts to the left). In the same way the image can be shifted to the right or upwards or downwards.

ar1=np.zeros((28,28))

for i in range (28):

 for j in range(28-n):

 ar1[i,j]=ar[(i),(j+n)]

1. **Processing the camera image**

We are going to sort products based on a number that is written on a sticker that is stuck on the product. By taking a picture of the number and detecting what number it is, a robot can do the sorting.

We start making the picture and processing the image. But first of all, with the Pi switched off, you’ll need to connect the Camera Module to the Raspberry Pi’s camera port, then start up the Pi and ensure the software is enabled (sudo raspi-config; Interfacing Options; Camera). Rather than using the Raspberry Pi [camera module](https://www.raspberrypi.org/documentation/usage/camera/README.md), you can use a standard USB webcam to take pictures and video on the Raspberry Pi - see Appendix 5

For getting an image use the code in the box:

from picamera import PiCamera # import the class PiCamera from the picamera library

import time

camera = PiCamera() # an object ‘camera’ is created

camera.start\_preview() # the lens is opened

time.sleep(2) # 2 sec are given for adjusting parameters

camera.capture('/home/pi/image.jpg') # a picture is taken; you can use any folder

camera.stop\_preview() # the lens is closed

Set up the camera and the object with the number. Make a picture of the number (in due course the presence of an object triggers the camera automatically but initially we will do it manually). The image is stored on the pi. Have a look at the resulting image and adjust the position if necessary. The image has a size of 720x480 pixels.

We take the distance between camera and object to be constant. The same applies to the size of the digit, the position of the camera with respect to the digit (horizontal, vertical or upside down) and the approximate light intensity. The related parameters have to be tuned once in this process.

For detecting the image we use the following method:

* the image is written on a white sticker, stuck on the product

 4.

* in the bottom right-hand corner of the section that contains the digit we put a small dot (see image)
* we first detect the dot
* when having the location of the dot we can crop the section with the image of the digit
* finally, we offer the cropped section to a trained AI network.

We start defining some constants.

factor=12

dim\_digit=factor\*28

stride=4

white = 80

black=40 axes of plotted image

size\_dot=4 (0,0) j (720)

dim\_dot=factor\*size\_dot

filter=np.zeros((dim\_dot,dim\_dot))

for i in range (1,dim\_dot-1):

 for j in range (1,dim\_dot-1): i

 filter[i,j]=255 (480)

model=load\_model('pretrained\_model1.h5')

img=Image.open('image.jpg')

im=img.convert('L')

ar=np.array(im)

found=False

i\_ref=0

while ((not found) and (i\_ref<480-dim\_digit)):

 j\_ref=0

 while ((not found) and (j\_ref<720-dim\_digit)):

 ar1=ar[i\_ref:(i\_ref+dim\_dot),j\_ref:(j\_ref+dim\_dot)]

 conv\_matrix= filter+ar1

 found=np.all(conv\_matrix>white) and np.any(ar1<black)

 j\_ref=j\_ref+stride

 i\_ref=i\_ref+stride

i\_ref=i\_ref-stride

j\_ref=j\_ref-stride

plt.imshow(ar,cmap='gray',vmin=0,vmax=255)

plt.gca().add\_patch(patch.Rectangle((j\_ref,i\_ref),dim\_dot,dim\_dot,linewidth=1,edgecolor='r',facecolor='none'))

plt.show()

* Because the image we offer to the AI network must have a size of 28x28 we will crop a section out of the camera image with dimensions that are multiples of 28. The constant ‘factor’ is the multiplication factor. Hence the dimensions of the section to be cropped are factor\*28.
* We can choose the value for the stride – with a smaller value for the stride the result is more accurate but it takes a longer time to do the processing.
* We choose the minimum value for what we consider as white and the maximum value for what we consider as black.
* Finally we assume a maximum value for the dot we included in the image – if we take 4x4 (the parameter size\_dot is equal to 4) pixels in the image of the digit, then at the camera image the dimensions are factor\*size\_dot.

After importing the necessary modules and specifying the constants, we define a filter for detecting the dot. We define a filter array with dimensions dim\_dot filled with white (255) and with the outer elements black (0). We load the model and the image that was stored after taking a picture with the camera. The image is converted into a grey-scale numpy array.

The dot is detected by adding the subsequent image sections to the filter array. If the elements of the sum are white (we specified the minimum white level as a constant) and the image section has at least 1 element that is black (the maximum black level was specified as a constant) then we ‘captured’ the dot. Since the dot was put in the corner of the image of the digit we can now crop this image.

We subsequently have to process the array of the image with the following actions:

* crop the image and convert it to a 28x28 array (it is good to include a plot after each step initially; in that way you can check the result; the code for plotting can be removed later)

ar2=np.empty((28,28))

for i in range (28):

 for j in range(28):

 ar2[i,j]=ar[(i\_ref+i\*factor),(j\_ref+j\*factor)]

plt.imshow(ar2,cmap='gray',vmin=0,vmax=255)

plt.show()

* invert the image from dark in a light background to light in a dark background (hint: for a single element the inversion can be achieved with: ar3[i,j]=255-ar2[i,j]
* rotate the image if necessary (hint: a rotation over 180 degrees can be achieved by applying ar4[i,j]=ar3[27-i,27-j] to each pixel
* stretch the range between ‘black’ and ‘white’ to the full range and convert this range to ‘float32’ between 0 and 1 (see code in the box)

min=255

max=0

for i in range (28):

 for j in range (28):

 if ar4[i,j]<min: min=ar4[i,j]

 if ar4[i,j]>max: max=ar4[i,j]

print('min',min,'max',max)

ar4-=min

ar4=ar4.astype('float32')

ar4/=(max-min)

* make all pixels below a certain value (e.g. 0.5) equal to zero (black); this conforms the images of the MNIST data set
* remove the dot from the image by making the pixels in that section equal to zero.

The result can be used as input for the trained AI network.

ar4=ar4.reshape(1,28,28,1)

pr1=model.predict(ar4)

for r in range(10):

 if ((pr1[0,r]>0.9)):

 digit\_shown=r

 reliability=round(100\*pr1[0,r],4)

 print('digit shown is ',digit\_shown,' ; reliability is ',reliability)

Exercises:

1. Suppose the white area is not all-white – there are some dark pixels. How could the software for detecting the area with the dot be modified in order to accommodate these dark pixels and make the software more robust.
2. What modifications are needed to have the value of ‘factor’ being determined automatically by the software.
3. A single pixel is taken from the original image to determine the grey level of the pixel of the 28x28 image. Mathematically correct would be to take the average of a *factor* by *factor* square to be converted to a pixel of the 28x28 image (why?). This could be achieved by the code line ar2[i,j]=np.mean(ar[(i\_ref+i\*factor):(i\_ref+(i+1)\*factor),(j\_ref+j\*factor):(j\_ref+(j+1)\*factor)]) in which the numpy function mean() calculates the average of the elements of an array. Interpret this code line and check whether this modification results in a significant improvement. Explain your observation.
4. **Physical set-up of robot with camera**

The physical set-up is as follows:

- a camera takes a picture of an object that is transported on an imaginary conveyor-belt

- the AI software in the Raspberry Pi analyzes the image and determines the number that is written on the object

- the robot sorts the objects arriving by picking them up and transporting them to a position that corresponds to the number on the object.



We will first have a closer look at the robot and the programming of it. The robot we use is controlled by servo motors with a position control based on Pulse Width Modulation (PWM).

PWM is used in many systems. It means that a periodic signal is high for a certain part of the cycle and low for the rest of it. The percentage of time that the signal is high is called the duty cycle.

PWM is available on the raspberry through the pigpio library. The pigpio library generates an accurate pwm signal (we could also

generate a pwm signal in software but that is less accurate). **Pigpio uses BCM numbering**.

Since pigpio affects the hardware directly we have to start a specific program at the highest command level. For this purpose, open the terminal and enter the shell command: sudo pigpiod (‘sudo’ stands for ‘superuser do’; pigpiod is the daemon for pigpio). At the end of our program we have to terminate the daemon with the shell command: sudo killall pigpiod.

The position of the servo motor is set by a PWM signal based on a frequency of 50 Hz. With a duty cycle of about 5% the servo angle will be at its minimum (-90 degrees); if the duty cycle is 7.5 % the servo will be at its center position (0 degrees) and if it is about 10 % it will be at its maximum (+90 degrees).

Since the range of the duty cycle in the PWM method in pigpio is 0 – 255: 7.5% is about 18, 5% is about 12 and 10% is about 24.

Connect the bottom servo (red: positive of external power supply; black: ground of external power supply and Raspberry; yellow: PWM input signal). We will use port GPIO21 in BCM numbering (in BOARD numbering pin 40) of the Raspberry for the PWM signal.

Enter and run the program in the box (make sure you know the meaning of each of the statements). In the program we use the keyboard keys (in this case ‘ a’ and ‘ s’ but you can take any other combination) to rotate the servo.

start the pigpiod deamon from the terminal

import pigpio

import time

pin = 21

d=18

pwm=pigpio.pi()

pwm.set\_mode(pin,pigpio.OUTPUT)

pwm.set\_PWM\_frequency(pin,50)

pwm.set\_PWM\_dutycycle(pin,d)

c=' '

while (c!='q'):

 if c=='a': d=d-1

 if c=='s': d=d+1

 if d<8:

 d=8

 print ("lower limit reached")

 if d>28:

 d=28

 print ("upper limit reached")

 pwm.set\_PWM\_dutycycle(pin,d)

 print (“The present position is: “,d)

 c=input("enter next move : ")

pwm.set\_mode(pin,pigpio.INPUT)

pwm.stop()

Note: you will notice that the servo has a hysteresis (that means that when it changes direction, it takes an extra step); this hysteresis comes from the internal gearbox.

Connect the other two servos. Choose pins to supply the PWM signal.

Write a program that allows a user to make the three servos turn in either direction.

The movement to any position is controlled by repetitive pressing of keys on the keyboard (choose keys for each of the movements). Obviously three different objects are needed, each with its own unique name. Test the program by making the robot arm pick up an object and place it at a predefined position.

Extend the program in such a way that all keystrokes are added to a list. Print the list at the end.

Extend the program in such a way that after pressing the ‘q’ key the program asks whether you want to repeat the movement of the arm. If ‘y’ is hit the servos move to the start position and make the movement stored in the list. Hint: include a small delay between the steps read from the list; otherwise the movement goes too fast. Also, if time allows: various optimizations are possible like making the delay dependent on the kind of movement – continuous or near reaching a stop position – and deleting unnecessary movements from the list.

Modify the program in such a way that an object is picked up and is placed in a position that depends on a variable ‘number’. The value of ‘number’ is at this stage entered through the keyboard – in the final setup it will be supplied by the IA program. Assuming 3 different values for ‘number’ there will be 3 lists to make the robot-arm bring the object in the correct position (it is also possible to make a list for the common part of the movement and three additional lists for the last sections). The lists have to be made one time and can be used afterwards in the final program. This is in fact a basic form of machine learning: the robot is trained once and the resulting ‘knowledge’ – in this case in the form of lists – can be used for later operations.

Assemble the various programs in such a way that when an object is placed in front of the camera a picture is taken, the digit on the object is determined and the object is placed by the robot at the right place. The process of taking a picture is started manually.

Extend the program in such a way that the process is started automatically. The camera takes pictures at regular intervals and detects the presence of an object by detecting the dot – the chance of detecting a dot when there is no object is very small (more robust software can bring this chance to an acceptable minimum).

With the last method we can also build software for detecting the arrival of an object when objects arrive on a constantly moving conveyor belt (the speed has to be such that at least one of the pictures taken contains the full digit – the conveyor belt is stopped in front of the robot when an object is detected).

For testing the software move the object alongside the camera with the proper speed. A light, or any other signal, indicates that the conveyer belt has to stop.

1. **Image rotation**

So far, we have assumed the image to be in an upright position. If the image can be in any position, we can follow either of the following approaches:

1. Train the model with images of digits in any rotational position.
2. Underline the digits, detect the position of the line[[9]](#footnote-9) and rotate the image accordingly to get it in the right position.
3. If the range of rotation is limited a few images, mutually rotated, can be offered to the model. If one of the results is more than 99 % that image is most probably in the correct position and the detected number can be taken as correct.

The software for rotation is based on the following relationship:

We have an image ‘im’ and rotate it through an angle ‘ang’ giving a new image ‘im\_rot’. Then the values of image pixels have the following relationship:

xn=round(cos(ang)\*x+sin(ang)\*y)

yn=round(-sin(ang)\*x+cos(ang)\*y)

whereby x and y are the co-ordinates of the original image related to the point of rotation, ang is the angle of rotation (clockwise) and xn and yn are the co-ordinates of the rotated image related to the same point of rotation.

Alternatively, we can use the rotate function from the Image module in the Pillow library (the library has the name PIL).

In each of the boxes the code for one of the alternative methods for rotation is shown.

Exercises:

1. Check what the impact is of turning an MNIST image first in one direction over a number of degrees and then back over the same number of degrees. How does the rotation affect the reliability of the prediction?
2. For how many degrees can an image (take any image of de MNIST dataset) be turned before the detection fails. Will it be the same for all images?
3. Implement any of the previously mentioned methods for position independency.

from keras.datasets import mnist

import numpy as np

import matplotlib.pyplot as plt

import math

(X\_train,y\_train),(X\_test,y\_test)=mnist.load\_data()

im=X\_test[234]

im\_rot=np.zeros((28,28))

ang=0.3\*math.pi

for x in range (28):

 for y in range (28):

 xa=x-14; ya=y-14

 xn=round(math.cos(ang)\*xa+math.sin(ang)\*ya)

 yn=round(-math.sin(ang)\*xa+math.cos(ang)\*ya)

 xn=xn+14; yn=yn+14

 if (xn<0): xn=0

 if (xn>27): xn=27

 if (yn<0): yn=0

 if (yn>27): yn=27

 im\_rot[xn,yn]=im[x,y]

plt.subplot(221); plt.imshow(im,cmap='gray')

plt.subplot(222); plt.imshow(im\_rot,cmap='gray')

plt.show()

from keras.datasets import mnist

from PIL import Image

import matplotlib.pyplot as plt

import matplotlib.image as mim

(X\_train,y\_train),(X\_test,y\_test)=mnist.load\_data()

im=X\_test[234]

img=Image.fromarray(im)

ang=-45

im\_rot=img.rotate(ang)

plt.subplot(221); plt.imshow(im,cmap='gray')

plt.subplot(222); plt.imshow(im\_rot,cmap='gray')

plt.show()

im\_rot=im\_rot.astype('uint8')

im=Image.fromarray(im\_rot)

im.save('im\_rot.jpg')

# or

mim.imsave('im\_rot.jpg',im\_rot)

**Appendix 1 NN with and without dropout layer**

import matplotlib.pyplot as plt

import numpy as np

from keras.utils import plot\_model

from keras.utils import np\_utils

from keras.datasets import mnist

from keras.models import Sequential

from keras.layers import Dense,Conv2D

from keras.layers import MaxPooling2D,Flatten

from keras.layers import Dropout

(X\_train,y\_train),(X\_test,y\_test)=mnist.load\_data()

np.random.seed(1)

X\_train=X\_train.reshape(60000,28,28,1)

X\_train=X\_train.astype('float32')

X\_train/=255

y\_train=np\_utils.to\_categorical(y\_train)

X\_test=X\_test.reshape(10000,28,28,1)

X\_test=X\_test.astype('float32')

Train on 60000 samples, val. on 10000 samples

Epoch 1/25 - val\_acc: 0.9817

Epoch 6/25 - val\_acc: 0.9871

Epoch 11/25 - val\_acc: 0.9865

Epoch 12/25 - val\_acc: 0.9861

Epoch 13/25 - val\_acc: 0.9887

Epoch 14/25 - val\_acc: 0.9880

Epoch 15/25 - val\_acc: 0.9861

Epoch 16/25 - val\_acc: 0.9892

Epoch 17/25 - val\_acc: 0.9876

Epoch 18/25 - val\_acc: 0.9880

Epoch 19/25 - val\_acc: 0.9878

Epoch 20/25 - val\_acc: 0.9862

Epoch 21/25 - val\_acc: 0.9878

Epoch 22/25 - val\_acc: 0.9867

Epoch 23/25 - val\_acc: 0.9869

Epoch 24/25 - val\_acc: 0.9881

Epoch 25/25 - val\_acc: 0.9863

Train on 60000 samples, val. on 10000 samples

Epoch 1/25 - val\_acc: 0.9783

Epoch 6/25 - val\_acc: 0.9861

Epoch 11/25 - val\_acc: 0.9889

Epoch 12/25 - val\_acc: 0.9872

Epoch 13/25 - val\_acc: 0.9870

Epoch 14/25 - val\_acc: 0.9865

Epoch 15/25 - val\_acc: 0.9866

Epoch 16/25 - val\_acc: 0.9887

Epoch 17/25 - val\_acc: 0.9889

Epoch 18/25 - val\_acc: 0.9887

Epoch 19/25 - val\_acc: 0.9887

Epoch 20/25 - val\_acc: 0.9877

Epoch 21/25 - val\_acc: 0.9885

Epoch 22/25 - val\_acc: 0.9893

Epoch 23/25 - val\_acc: 0.9885

Epoch 24/25 - val\_acc: 0.9889

Epoch 25/25 - val\_acc: 0.9876

X\_test/=255

y\_test=np\_utils.to\_categorical(y\_test)

num\_classes=10

model1=Sequential()

model1.add(Conv2D(24,(4,4),input\_shape=(28,28,1),activation='relu'))

model1.add(MaxPooling2D(pool\_size=(2,2)))

model1.add(Flatten())

model1.add(Dense(64,activation='relu'))

model1.add(Dense(64,activation='relu'))

model1.add(Dense(num\_classes,kernel\_initializer='normal',activation='softmax'))

model1.compile(loss='categorical\_crossentropy',optimizer='adam',metrics=['accuracy'])

history1=model1.fit(X\_train,y\_train, validation\_data=(X\_test,y\_test),epochs=25,batch\_size=16,verbose=2)

model2=Sequential()

model2.add(Conv2D(24,(4,4),input\_shape=(28,28,1),activation='relu'))

model2.add(MaxPooling2D(pool\_size=(2,2)))

model2.add(Flatten())

model2.add(Dense(64,activation='relu'))

model2.add(Dropout(0.35))

model2.add(Dense(64,activation='relu'))

model2.add(Dense(num\_classes,kernel\_initializer='normal',activation='softmax'))

model2.compile(loss='categorical\_crossentropy',optimizer='adam',metrics=['accuracy'])

history2=model2.fit(X\_train,y\_train, validation\_data=(X\_test,y\_test),epochs=25,batch\_size=16,verbose=2)

plt.plot(history1.history['acc'])

plt.plot(history1.history['val\_acc'])

plt.plot(history2.history['acc'])

plt.plot(history2.history['val\_acc'])

plt.title('Model accuracy 1+2')

plt.ylabel('accuracy')

plt.xlabel('epoch')

plt.legend(['Train 1','Test 1','Train 2','Test 2'],loc='upper left')

plt.show()

**Appendix 2 Available networks**



Model: pretrained\_model1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d\_1 (Conv2D) (None, 24, 24, 30) 780

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_1 (MaxPooling2D) (None, 12, 12, 30) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_2 (Conv2D) (None, 10, 10, 15) 4065

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_2 (MaxPooling2D) (None, 5, 5, 15) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_1 (Dropout) (None, 5, 5, 15) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten\_1 (Flatten) (None, 375) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 128) 48128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2 (Dense) (None, 50) 6450

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_3 (Dense) (None, 10) 510

=================================================================

Total params: 59,933

Trainable params: 59,933

Non-trainable params: 0

Model: model2

Layer (type) Output Shape Param #

=================================================================

conv2d\_1 (Conv2D) (None, 24, 24, 16) 416

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_1 (MaxPooling2D) (None, 12, 12, 16) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_2 (Conv2D) (None, 10, 10, 16) 2320

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_2 (MaxPooling2D) (None, 5, 5, 16) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_1 (Dropout) (None, 5, 5, 16) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten\_1 (Flatten) (None, 400) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 128) 51328

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2 (Dense) (None, 10) 1290

=================================================================

Total params: 55,354

Trainable params: 55,354

Non-trainable params: 0

**Appendix 3 Reliability of predictions**

from keras.datasets import mnist

from keras.models import Sequential

from keras.layers import Dense,Conv2D,MaxPooling2D,Flatten

from keras.utils import np\_utils

(X\_train,y\_train),(X\_test,y\_test)=mnist.load\_data()

X\_train=X\_train[0:20000,:28,:28]

y\_train=y\_train[0:20000]

num\_classes=10

X\_train=X\_train.reshape(X\_train.shape[0],28,28,1)

X\_train=X\_train.astype('float32')

X\_train/=255

X\_test=X\_test.reshape(X\_test.shape[0],28,28,1)

X\_test=X\_test.astype('float32')

X\_test/=255

y\_test=np\_utils.to\_categorical(y\_test)

y\_train=np\_utils.to\_categorical(y\_train)

model\_reliability1=Sequential()

model\_reliability1.add(Conv2D(16,(3,3),input\_shape=(28,28,1),activation='relu'))

model\_reliability1.add(MaxPooling2D(pool\_size=(2,2)))

model\_reliability1.add(Flatten())

model\_reliability1.add(Dense(64,activation='relu'))

model\_reliability1.add(Dense(num\_classes,kernel\_initializer='normal',activation='softmax'))

model\_reliability1.compile(loss='categorical\_crossentropy',optimizer='adam',metrics=['accuracy'])

model\_reliability1.fit(X\_train,y\_train, epochs=5,batch\_size=100,verbose=2)

score=model\_reliability1.evaluate(X\_test,y\_test,verbose=2)

print ('validation acc: ',round(score[1],4))

model\_reliability1.save('model\_reliability1.h5')

model\_reliability2=Sequential()

model\_reliability2.add(Conv2D(16,(3,3),input\_shape=(28,28,1),activation='relu'))

model\_reliability2.add(MaxPooling2D(pool\_size=(2,2)))

model\_reliability2.add(Flatten())

model\_reliability2.add(Dense(64,activation='relu'))

model\_reliability2.add(Dense(num\_classes,kernel\_initializer='normal',activation='linear'))

model\_reliability2.compile(loss='mean\_squared\_error',optimizer='adam',metrics=['accuracy'])

model\_reliability2.fit(X\_train,y\_train, epochs=5,batch\_size=100,verbose=2)

score=model\_reliability2.evaluate(X\_test,y\_test,verbose=2)

print ('validation acc: ',round(score[1],4))

model\_reliability2.save('model\_reliability2.h5')

A problem with determining the reliability – the degree of certainty – of a prediction is that when using softmax activation in the last layer the chances for the various digits add up to 1. Hence, there is a good chance that one of the digits has a chance of more than 0.9 even if the input image was a collection of random pixels. To overcome this problem we can use ´linear´ activation for the last layer in combination with loss=

'mean\_ squared\_error'

at compilation. The program in the box shows how to generate the two models. In the other box the code is given for predicting the image composed of random digits.

In practical situations it can be decided which one or a combination of the two is to be used. Alternatively, computer vision software can be used to determine whether the image shows a digit or not.

from keras.datasets import mnist

from keras.utils import np\_utils

from keras.models import load\_model

import matplotlib.pyplot as plt

import random

import numpy as np

model\_reliability1=load\_model('model\_reliability1.h5')

model\_reliability2=load\_model('model\_reliability2.h5')

im=np.empty((28,28))

for i in range(28):

 for j in range(28): im[i,j]=random.random()

pr1=model\_reliability1.predict(im.reshape(1,28,28,1))

for i in range(10): print(i,round(100\*pr1[0,i],8))

pr2=model\_reliability2.predict(im.reshape(1,28,28,1))

for i in range(10): print(i,round(100\*pr2[0,i],8))

**Appendix 4 Image processing with OpenCV**

import cv2

*# Create instance of cascade classifier for detecting faces*

faceCascade = cv2.CascadeClassifier("haarcascade\_frontalface\_default.xml")

image = cv2.imread('obama.jpeg')

cv2.imshow('',image)

cv2.waitKey(0)

cv2.destroyAllWindows()

*# Convert into grayscale*

gray = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

cv2.imshow('',gray)

cv2.waitKey(0)

cv2.destroyAllWindows()

*# Look for faces in the image using the loaded cascade file*

faces = faceCascade.detectMultiScale(gray,scaleFactor=1.1 ,minNeighbors=5)

*# For each face*

for (x, y, w, h) in faces:

 *# Draw rectangle around the face*

 *# (´255´ stands for the color and´3´ for the thickness of the lines)*

 cv2.rectangle(gray, (x, y), (x+w, y+h), (255, 255, 255), 3)

cv2.imshow('',gray)

cv2.waitKey(0)

cv2.destroyAllWindows()

OpenCV is a library for real-time image processing. The advantage of using this library, as compared to writing the code ourselves in Python, is that the various modules are written in a fast executing languages (C++) and optimized for speed.

With OpenCV we can manipulate images and detect objects[[10]](#footnote-10).

In the example in the box we detect faces (frontal).

Some comments:

The word “cascade” in the classifier name means that the resultant classifier consists of several simpler classifiers (stages). A ´Haar cascade´ is basically a classifier that is used to detect the object for which it has been trained – in our example frontal faces. It tests on a number of Haar features. A Haar-feature is just like a kernel in CNN, except that in a CNN, the values of the kernel are determined by training, while a Haar-feature is manually determined. The diagram shows some Haar-features. The first two are “edge features”, used to detect edges.  The third is a “line feature”, while the fourth is a “four rectangle feature”, most likely used to detected a slanted line.

Caution: if the file that you need is not in the working directory you should enter the complete path of the file[[11]](#footnote-11)[[12]](#footnote-12).

The code lines ´cv2.waitKey(0)´ plus´cv2.destroyAllWindows()´ allow you to terminate the displayed image and continue with the program[[13]](#footnote-13).

The parameter ´scaleFactor´: detecting the faces occurs with a fixed size window but the image is resized stepwise; the scale factor determines the step at which the scaling takes place.

The parameter ´minNeighbors´: the idea behind this parameter is that the detector will run on a **multiple scale style** and at the same time following **sliding window strategy**. Doing so, it will give you multiple responses even for a single face region. The parameter sets a lower-bound threshold for the number of hits it gets in this way for a particular face, i.e. it will only be counted as a valid face if the number of responses for this face is higher than minNeighbors.

The function ´detectMultiScale´ returns a list of tuples: ´faces´ is a list of tuples consisting of position parameters x and y and the width and height of the square.

Try to identify faces on images made by the camera connected to the Raspberry Pi[[14]](#footnote-14). Be aware that only frontal faces are detected and that parameters might have to be adjusted.

**Appendix 5 Using the usb camera for imagesng a standard USB webcam**

If the camera module has not been installed yet: install the fswebcam package: sudo apt-get install fswebcam

Enter the command fswebcam followed by a filename and a picture will be taken using the webcam, and saved to the filename specified: fswebcam image.jpg

This command will show the following information:

--- Opening /dev/video0...

Trying source module v4l2...

/dev/video0 opened.

No input was specified, using the first.

Adjusting resolution from 384x288 to 352x288.

--- Capturing frame...

Corrupt JPEG data: 2 extraneous bytes before marker 0xd4

Captured frame in 0.00 seconds.

--- Processing captured image...

Writing JPEG image to 'image.jpg'.

The webcam used in this example has a resolution of 1280 x 720. To specify the resolution that I want the image to be taken at, use the -r flag: fswebcam -r 1280x720 image.jpg

You can get pictures without banner at the bottom by adding the --no-banner flag:

fswebcam -r 1280x720 --no-banner image.jpg

You can write a Python program which takes a picture with the webcam. By importing os we can have any terminal command being executed in a program.

import os

os.system('fswebcam -r 1280x720 --no-banner image.jpg')

With the program in the box the image is taken and displayed.

import os

import matplotlib.pyplot as plt

import matplotlib.image as mpimg

import numpy as np

os.system('fswebcam -r 1280x720 --no-banner image.jpg')

im=mpimg.imread('image.jpg')

plt.imshow(im,cmap='gray',vmin=0,vmax=255)

plt.show()

**Appendix 6 Using the usb camera for video**

OpenCV (we use cv2) is very suitable for making and manipulating images. The program in the box shows the video from the webcam on the screen and every time the space-bar is hit an image is stored and shown on the screen.

import cv2

cam = cv2.VideoCapture(0)

cv2.namedWindow("test")

img\_counter = 0

while True:

 ret, frame = cam.read()

 cv2.imshow("test", frame)

 k = cv2.waitKey(1)

 if k%256 == 27:

 # ESC pressed

 print("Escape hit, closing...")

 break

 elif k%256 == 32:

 # SPACE pressed

 img\_name = "opencv\_frame\_{}.png".format(img\_counter)

 cv2.imwrite(img\_name, frame)

 print("{} written!".format(img\_name))

 cv2.imshow(img\_name,frame)

 img\_counter += 1

cam.release()

A useful tutorial on image processing in as well images as video´s is found here: <https://pythonprogramming.net/loading-video-python-opencv-tutorial/>

**Appendix7 Quality indicators in summary**

The ´lifetime´ of an AI network can be considered as consisting of the following phases:

Training

Testing/validating

Production

We came across various quality indicators for each of these phases.

*Training*

The accuracy of a training sequence is calculated by dividing the number of correct predictions by the total number of predictions; in other words, the accuracy shows the fraction that was predicted correctly.

Also, the average loss is a measure for the quality. The loss is the difference between the output that the model produces with a specific input image and the actual output corresponding to that image. The accuracy and the average loss can be calculated over an epoch or a complete training cycle.

(chapter 6)

*Testing*

During validation again the average loss and average accuracy are quality indicators. The evaluate function gives the overall score of the two indicators.

We can obtain the incorrect predictions with the function predict\_classes. By comparing them with the digits shown on the images offered at the input, we can get an idea of the errors made by the network.

 (chapter 7)

*Production*

The ‘predict’ function gives a prediction for an individual image; by comparing the percentages of the probabilities that were obtained for all possible outputs. In that way we can get an idea of the quality of an individual prediction (if the predicted value has a high probability – say more than 99 % – then the result is likely to be correct; if the probabilities of some digit are more or less the same than one can rightfully have doubts.

(chapter 7)

1. A tuple is a sequence of immutable Python objects. The difference between a list and a tuple is that tuples cannot be changed. Apart from that lists use square brackets and tuples use parentheses (round brackets). [↑](#footnote-ref-1)
2. The number of parameters in a dense layer is equal to # weights (= # inputs \* # neurons) + #biases (= # neurons)

 [↑](#footnote-ref-2)
3. You can check what you register yourself when you seen an image for a short interval. Use the following code:

plt.show(block=False)

time.sleep(0.03)

plt.close() [↑](#footnote-ref-3)
4. The model starts with random values for the weights and biases. Therefor different training sessions result in different trained models. We could get the same result every time by seeding the random generator that is being used.

Use the following code:

from numpy.random import seed

seed(1) # the value of the parameter is not important

In this way the sequence of ‘randomly’ generated numbers is the same every time. The random generator is in fact a pseudo-random generator. [↑](#footnote-ref-4)
5. In this network the inclusion of a dropout layer does not have any significant effect. A larger training set and a more complex network would have shown a clearer effect. [↑](#footnote-ref-5)
6. Running the program is possible on the Raspberry Pi but it will take about 2 hours. [↑](#footnote-ref-6)
7. The most remarkable characteristics are the lower accuracy during training and the higher accuracy during testing when using a dropout layer. Both effects result from the fact that the network with a dropout layer adapts itself less to the characteristics of the training set. [↑](#footnote-ref-7)
8. The print of the array depends on the order of the axes. If you want to get the output in a different order of the axes – resulting in a different layout – you can use name\_array.transpose(3,0,1,2) whereby the parameters give the desired order of the axes. [↑](#footnote-ref-8)
9. Alternatively we can use two dots at the two bottom edges to determine the ‘line’. [↑](#footnote-ref-9)
10. <https://www.analyticsvidhya.com/blog/2019/03/opencv-functions-computer-vision-python/> gives an overview of the main functions of OpenCV with examples in Python code. [↑](#footnote-ref-10)
11. E.g. /home/pi/opencv/data/haarcascades/haarcascade\_frontalface\_default.xml and /home/pi/opencv/samples/data/obama.jpeg [↑](#footnote-ref-11)
12. You can download the haarcascade files from Github; download the raw version of the file (so without the markup code for the browser). [↑](#footnote-ref-12)
13. In case you delete the image in another way and the Shell (terminal window) doesn´t react anymore, just close it and open a new one. [↑](#footnote-ref-13)
14. [↑](#footnote-ref-14)