# DAT405 Assignment 7 - Group 52

Hampus Jansson - (5 hrs)

Erik Johannesen - (5 hrs)

May 29, 2023

In [2]:
```
# imports
from __future__ import print_function
import keras
from keras import utils as np_utils
import tensorflow
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
from tensorflow.keras import regularizers
import tensorflow as tf
from matplotlib import pyplot as plt
import numpy as np
```

In [3]:
```
# Hyper-parameters data-loading and formatting

batch_size = 128
num_classes = 10
epochs = 10

img_rows, img_cols = 28, 28

(x_train, lbl_train), (x_test, lbl_test) = mnist.load_data()

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)
```

#### Preprocessing

First the data is converted to the type float32 which uses less memory than the standard float type which uses 64 bits. The reason for not using even fewer bits is that it would sacrifice precision. Pixel values might be represented by values from 0-255. By dividing all the values by 255, the values will range from 0 to 1 instead.

The last step transforms the Array with the labels into a binary class matrix. This is needed to match the expected format of the output layer.

In [4]:
```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

x_train /= 255
```

```
x_test /= 255

y_train = keras.utils.np_utils.to_categorical(lbl_train, num_classes)
y_test = keras.utils.np_utils.to_categorical(lbl_test, num_classes)
```

In [ ]:

```
**2.1**
<p>There are a total of 4 layers. The first layer is an input layer which flattens
<p>There are two hidden layers. These have 64 neurons each and use the activation
<br>ReLU has efficient computation but does not work well when there are negative
<p>The last layer is the output layer and uses softmax activation. Softmax activat

<p>The first layer does not do any computations so it does not have any parameters
The second hidden layer has (64*64)+64 = 4160 parameters. The output layer has (64
<p>This gves a total of 55050 parameters in the network.<p>

<p>The dimensions of the input layer are decided by the input data. The entire pic
<br>The output layer has 10 neurons because there are 10 classes to put the data i

**2.2**
<p>Categorical crossentropy is used. The function functional form is: <br>
L = - (sum from i=1 to output size)(yi*log(p_i))<br>
L = Loss<br>
y_i = true label for class i<br>
p_i = predicted probablity for class i<br>
The closer L is to 0 the better is the performance.<br>
This loss function is appriate for multi-class classification, other loss functi
Cross-entropy compares the predicted distribution of probabilities with the true l
<br>This makes the loss function appropriate for this task. <p>
```

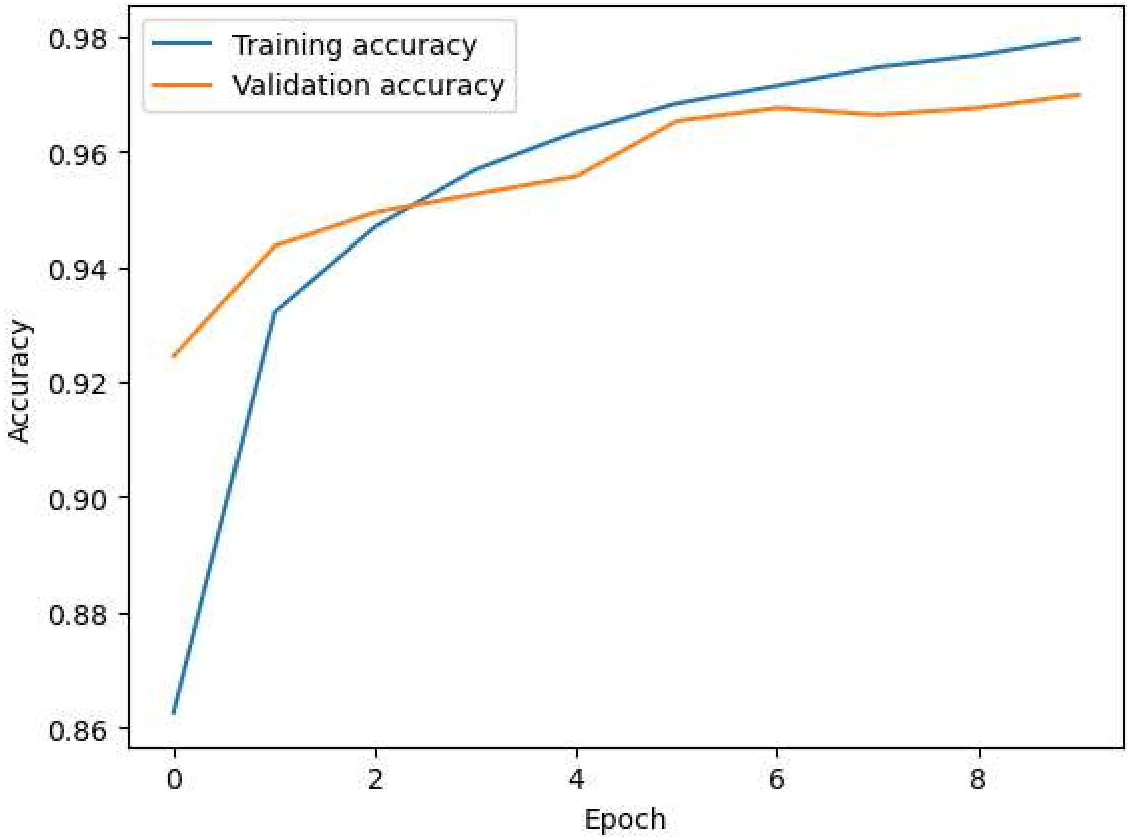
### 2.3

Here the model is created and trained for 10 epochs. Finally the training and valuation accuracy across the epochs is plotted.

In [7]:

```
## Define model ##
model = Sequential()

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


model.compile(loss=keras.losses.categorical_crossentropy,
               optimizer=tensorflow.keras.optimizers.SGD(learning_rate = 0.1),
        metrics=['accuracy'],)

fit_info = model.fit(x_train, y_train,
           batch_size=batch_size,
           epochs=epochs,
           verbose=1,
           validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss: {}, Test accuracy {}'.format(score[0], score[1]))

#plotting the history of the accuracy
plt.plot(fit_info.history['accuracy'])
plt.plot(fit_info.history['val_accuracy'])
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Training accuracy', 'Validation accuracy'], loc='upper left')
plt.show()
```

```
Epoch 1/10
469/469 [==============================] - 1s 2ms/step - loss: 0.4887 - accuracy:
0.8625 - val_loss: 0.2634 - val_accuracy: 0.9245
Epoch 2/10
469/469 [==============================] - 1s 2ms/step - loss: 0.2347 - accuracy:
0.9322 - val_loss: 0.1944 - val_accuracy: 0.9436
Epoch 3/10
469/469 [==============================] - 1s 2ms/step - loss: 0.1792 - accuracy:
0.9469 - val_loss: 0.1644 - val_accuracy: 0.9494
Epoch 4/10
469/469 [==============================] - 1s 2ms/step - loss: 0.1457 - accuracy:
0.9569 - val_loss: 0.1496 - val_accuracy: 0.9526
Epoch 5/10
469/469 [==============================] - 1s 2ms/step - loss: 0.1240 - accuracy:
0.9634 - val_loss: 0.1369 - val_accuracy: 0.9557
Epoch 6/10
469/469 [==============================] - 1s 2ms/step - loss: 0.1079 - accuracy:
0.9684 - val_loss: 0.1136 - val_accuracy: 0.9653
Epoch 7/10
469/469 [==============================] - 1s 2ms/step - loss: 0.0955 - accuracy:
0.9714 - val_loss: 0.1073 - val_accuracy: 0.9676
Epoch 8/10
469/469 [==============================] - 1s 2ms/step - loss: 0.0858 - accuracy:
0.9747 - val_loss: 0.1134 - val_accuracy: 0.9664
Epoch 9/10
469/469 [==============================] - 1s 1ms/step - loss: 0.0777 - accuracy:
0.9768 - val_loss: 0.1070 - val_accuracy: 0.9676
Epoch 10/10
469/469 [==============================] - 1s 2ms/step - loss: 0.0702 - accuracy:
0.9796 - val_loss: 0.0985 - val_accuracy: 0.9699
Test loss: 0.09854990243911743, Test accuracy 0.9699000120162964
```




**2.4**

The best result that was achieved when regularization factors were used was an accuracy of around 75%.

Other than the choice of regularization factors, it's also unknown how many layers Hinton used and how many neurons he used in each layer.

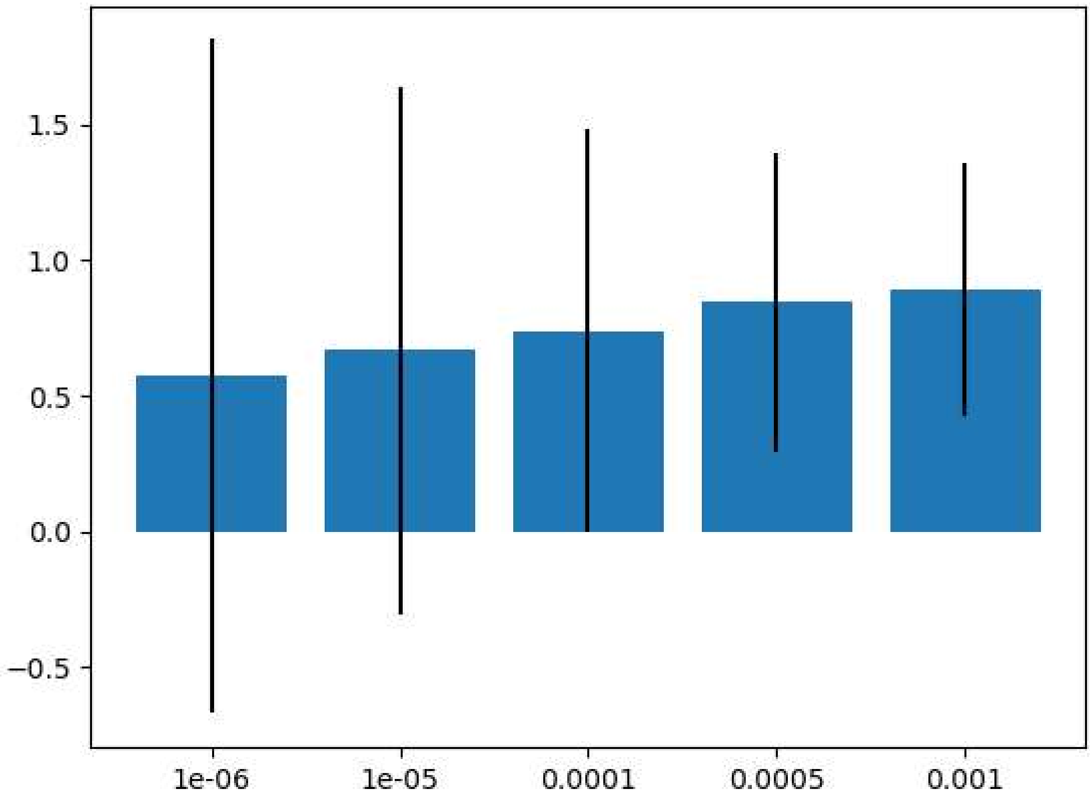
In [58]:

```
epochs=40
model = Sequential()
regularization_factors = [0.000001, 0.00001, 0.0001, 0.0005, 0.001]
scores=[]
losses=[]

#creating models for each choice of regularization factor. creating, fitting and co
for x in regularization_factors:

    model.add(Flatten())
    model.add(Dense(300, activation = 'relu',kernel_regularizer=regularizers.L2(x)
    model.add(Dense(500, activation = 'relu',kernel_regularizer=regularizers.L2(x)
    model.add(Dense(num_classes, activation='softmax'))

    model.compile(loss=keras.losses.categorical_crossentropy,
                  optimizer=tensorflow.keras.optimizers.SGD(learning_rate = 0.1),
            metrics=['accuracy'],)

    fit_info = model.fit(x_train, y_train,
               batch_size=batch_size,
               epochs=epochs,
               verbose=0,
               validation_data=(x_test, y_test))
    score = model.evaluate(x_test, y_test, verbose=0)
    losses.append(score[0])
    scores.append(score[1])
    print('Regularization factor: {},Test loss: {}, Test accuracy {}'.format(x,scor

#plotting the results of the different models
plt.bar(np.arange(len(regularization_factors)), scores, yerr=losses,tick_label=regu
```

```
Regularization factor: 1e-06,Test loss: 1.2407552003860474, Test accuracy 0.569999
9928474426
Regularization factor: 1e-05,Test loss: 0.9703974723815918, Test accuracy 0.666199
9821662903
Regularization factor: 0.0001,Test loss: 0.7427606582641602, Test accuracy 0.73919
99959945679
Regularization factor: 0.0005,Test loss: 0.5481944680213928, Test accuracy 0.84409
99984741211
Regularization factor: 0.001,Test loss: 0.4655928313732147, Test accuracy 0.891200
0060081482
```

Out[58]:

```
<BarContainer object of 5 artists>
```




**3.1**

Using a model which included a convolutional layer resulted at best with an accuracy of around 90%.
After the convolutional layer, a pooling layer was added to reduce the amount of computation needed.
Different optimizers were tried and 'adam' gave the best result. SGD which was used in the previous models had very poor performance in all attempts when a convolutional layer was involved.

In [ ]:

```
#settings
epochs = 10
model = Sequential()

#creating the layers
keras.Input(shape=(28,28,1))
model.add(Conv2D(32, activation = 'relu',kernel_size=(3,3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(num_classes, activation='softmax'))

#compiling and fitting model, choosing settings for model
model.compile(loss=keras.losses.categorical_crossentropy,
               optimizer='adam',
        metrics=['accuracy'],)

fit_info = model.fit(x_train, y_train,
           batch_size=batch_size,
           epochs=epochs,
           verbose=1,
           validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss: {}, Test accuracy {}'.format(score[0], score[1]))
```

### 3.2

Convolutional layers can be good for processing spatially structured data, (like images). It also means less computational complexity, which is important when working on big datasets like this. It also uses parameter sharing, which means that weights is shared by all neurons. This leads to less paramters in the system as a whole. Another important factor with convoltional layers is that they are able to develop a different representation of an image, which can give better insights that when flattening an image.

In [ ]:

```
```