Assignment 7

by

- Christoffer Wikner (931012)
- Erik Rosvall (960523)

1

In this code block:
you set the data type in the matrix as a float32. A float who is a 32-bit point number.

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
```

Dividing x_train and x_test with 255

```
x_train /= 255
x_test /= 255
```

Last rows in the pre-processing part:

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

to_categorical: this function converts a vector of type class to a binary matrix.
lbl_train: represents the class vector to be converted to the matrix it self.
num_classes: number of classes that are passed in

2

A)

This Neural Network has on input layer, two hidden layers and one output layer.

Number of neurons

Each picture are 28x28 pixels

$$28 * 28 = 784$$

Layers:

- The input layer: 784 (as a single vector)
- Hidden layer 1: 64
- Hidden layer 2: 64
- output layer: 10

The output layer have 10 because the input is an image with the numbers 0-9. The same applies to the output layer where the output are between 0-9.

B)

They are using *categorical_crossentropy* as a loss function. In other words, the values **mean**.

it can be found in this code block

```
loss=keras.losses.categorical_crossentropy
```

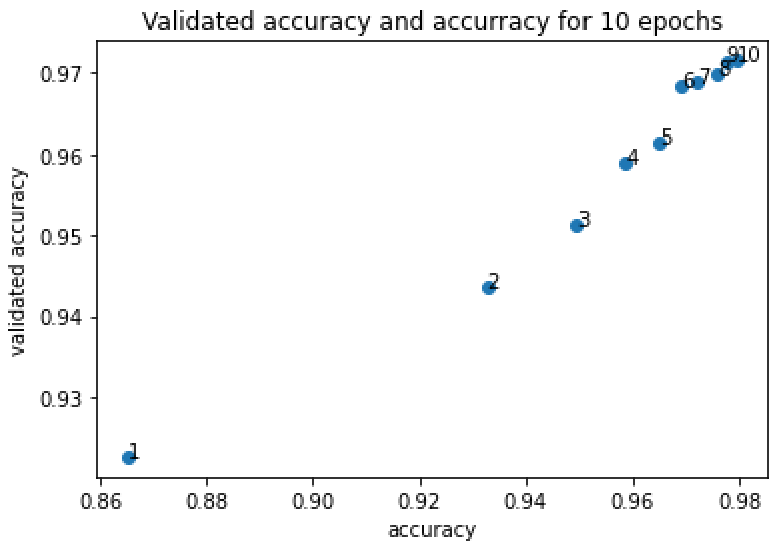
Mathematical description:

$$Loss = -\sum_{i=1}^{outputsize} y_i * \hat{y}_i$$

This loss measure the probability distribution between the two values. The minus ensures that the loss gets smaller if/when the distribution gets closer to each other.

Categorical crossentropy is a good technique for classifications problems, like the MNIST problem. categorical crossentropy is constructed to give a high probability for the correct digit, hence a lower probability for the non-correct digit.

C)



D)

PART 1

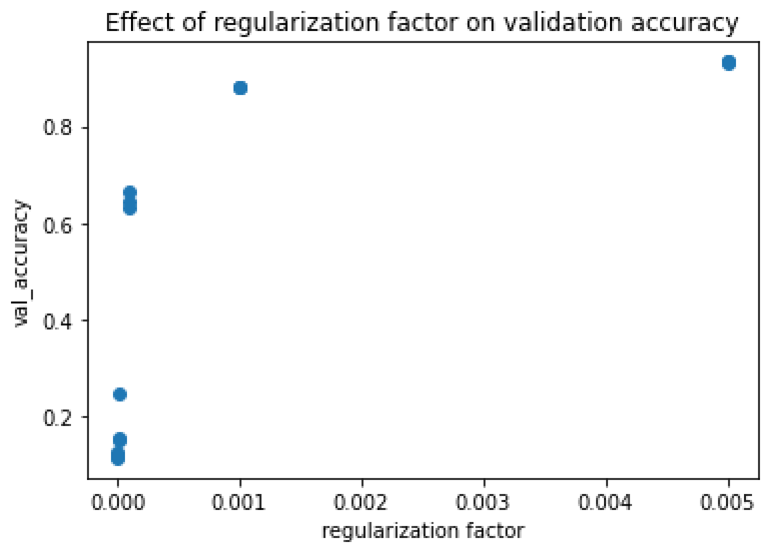
Running a 3 layer neural net with 500 and 300 hidden layers units respectively trained for 40 epochs we reached a validated accuracy of 0.9825999736785889.

PART 2

Scores from the run (mean value with different regularization factor):

- 0.001:
  - run 1: 0.8827949985861778
  - run 2: 0.8796149969100953
  - run 3: 0.8805699959397316
- 0.005
  - run 1: 0.9322899937629738
  - run 2: 0.9343074962496758

    - run 3: 0.9335399985313415
- 0.0001
    - run 1: 0.6656724993139506
    - run 2: 0.6439999975264072
    - run 3: 0.6331549979746341
- 0.00001
    - run 1: 0.1552674988284707
    - run 2: 0.2472900003194809
    - run 3: 0.1531550010666251
- 0.000001
    - run 1: 0.1170499999076128
    - run 2: 0.12772999927401543
    - run 3: 0.11470750011503697



> Because the dots in the graph are to close, we did not plot the number for each dot to make it more easy to read

PART 3

The closest we got was with a regularization factor of 0.1 for 40m epochs, with 2 hidden layer (500 and 300). The result was 0.9825999736785889

With our tests of 5 regularization factors between 0.001 and 0.000001 the highest mean we reached was 0.9343074962496758 when regularization factor was 0.005. This was somewhat surprising as the plot indicated before this a linear relation between higher regularization factor and validated accuracy. However as shown by the plot and results 0.005 outperformed 0.001.

Things that could account for the lower validated accuracy compared to Hinton's result of 0.9847. The computer hardware is one factor. Thoe there is a higher propability that Hinton has optimized the batch sizes, epochs, amount of layers and the amount of units in each layer. Hinton could also have used momentum (initial momentum, final momentum) instead of gradient desent, there is also a possibility that Hinton have used a static weight to the Neural Network.