Crash Course Deep Learning - Assignment 2

Instructor: Prof. Alexander Schönhuth
Submitted by: Jan Gerrit Hölting
Student ID: 6570518
Universiteit Utrecht

In this report, we will investigate the performance of convolutional neural networks in recognising handwritten digits. The report can be seen as an extension to [Höl19a], where basic neural networks (NN) have been applied to the same problem. The best accuracy any conventional neural network achieved was at 98% within approximately 20 minutes of training and can be found in Section 1b of [Höl19a]. In this report, we will therefore try to improve the performance and efficiency of handwritten digit recognition with the use of convolutional neural networks (CNN).

The test data set for the neural network will again be the MNIST database of handwritten digits (see [LCB19]) containing 60,000 training examples as well as 10,000 test examples. The experimental results have been obtained using the accompanying Python implementation which in turn uses Google's open-source machine learning platform TensorFlow 2 (see [Goo19]). All the code that has been used can be found in the GitHub repository [Höl19b].

General setup:

Some parameters of the neural network will remain constant throughout the following experiments. For more information on the interpretation of the parameters as well as the methods used, the reader is asked to consult Nielsen's *Neural Networks and Deep Learning* [Nie19] for an informal or Goodfellow et al.s' *Deep Learning* [GBC19] for a more formal introduction. We will use 50 epochs, a batch size of 100 while iterating through the example data and a standard gradient descent optimizer with a learning rate of 0.05. This means that we will deviate from [Höl19a] by choosing a much smaller learning rate (0.05 as compared to 0.5 in the earlier report). Furthermore, we will for each tested architecture only use one split of training, validation and test data and therefore refrain from cross-validation. As we will see, this improves efficiency while still achieving high accuracy of the chosen models. In the output layer we will use cross entropy as the loss function with softmax activation. All convolutional hidden layers will use ReLU as activation function while all fully connected hidden layers will use the sigmoid function as the activation function. Furthermore, the number 6570518 is used as a seed for all random variable initialisation.

1 Exercises

1.a

In this first part of the report, we will discuss a small but essential difference between conventional and convolutional neural networks, namely their differing approach in handling input data and data that is passed in-between layers. In the code for all of the following CNNs (see assignments/assignment2 in [Höl19b]) we will see that the network's input is reshaped as

```
x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
```

thus differing to the method

```
x_train = x_train.reshape((-1, img_rows*img_cols))
```

used in the earlier report [Höl19a].

Now the second command flattens the image, because fully connected NNs expect a vector as input (every layer's weights and biases are represented by a matrix), and all pixels are treated equally (there is no bundling of pixels based on their location in the vector). The second command returns a matrix representing the pixels which will be iterated through by the network automatically (in TensorFlow Keras). On a high level (not taking into account the actual memory structure and access), this makes the handling much easier and more intuitive than having to handle neighbourhood pixels with their indices in a flat vector. This is important because in CNNs, local environments of a pixel are taking into account. The 1 at the end of the first command indicates that we are handling a one-dimensional image in the sense that our input is a greyscale image instead of one that is RGB encoded. For an RGB encoded image, the 1 would be replaced by a 3. The application of a filter to an input image represented by a matrix of (grey-scale) values can be seen in Figure 1.

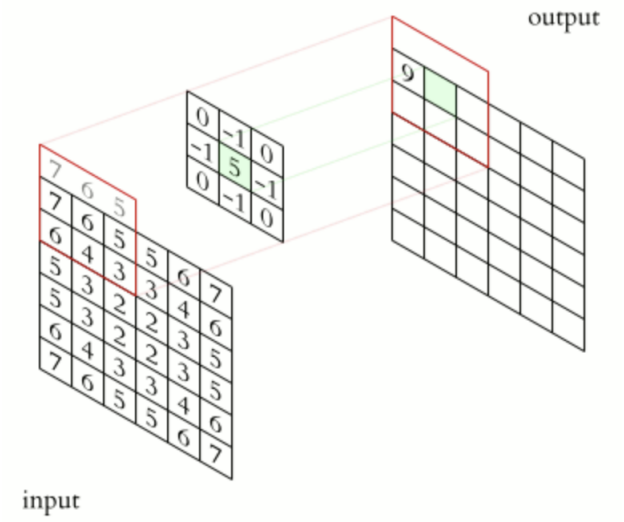

Figure 1: The working of a filter applied to an input image represented as a matrix of (grey-scale) values. Screenshot taken from Michael Plotke's animation on the German entry for CNNs on wikipedia, see [wik19].

1.b

As a first exercise in understanding the structure of CNNs, we will consider the following network:

1. Input layer
2. Convolution layer with a 5×5 filter and 6 output channels, stride 1, padding, followed by activation function ReLU
3. Max pooling layer with a 2×2 filter and stride 2×2, zero padding
4. Fully connected layer with 128 nodes, sigmoid as activation function
5. Output layer.

The Python implementation for this model can be found in `exercise1b.py` in assignments/assignment2 in [Höl19b].

1.c

Now the internal structure of the network (the output shapes of each layer) can be requested as a concise summary with the Keras command `model.summary()`. For the model proposed in 1.b, the output can be seen in Figure 2

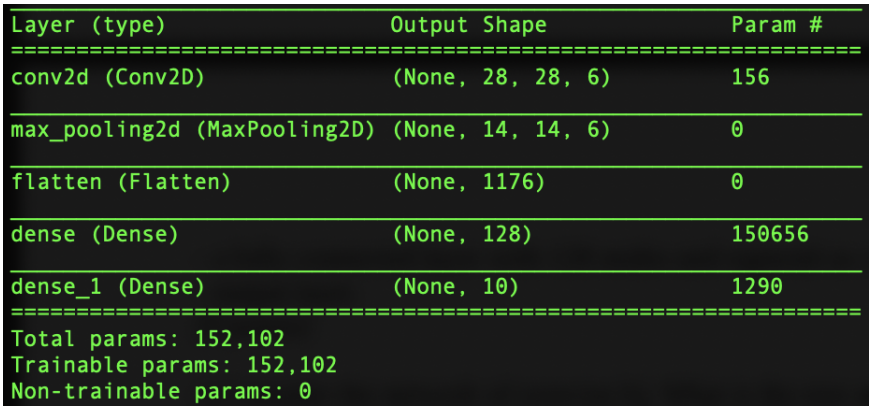

Figure 2: The internal data structure of the CNN proposed in Section 1.b, created by the Keras command `model.summary()`.

The input images of the MNIST data set are of dimension 28×28. Since in Section 1.b we decided to use padding in the (first,) convolutional layer with 6 output channels, we can observe that the output of that layer is again 28×28×6: the third dimension of 6 is a result of the choice of output channels, the 28×28 dimensions are kept because we use padding and a stride length of 1. Hence, each pixel will be considered as the center of a 5×5 local reference field and therefore contributes an output value, and we don't loose any dimensions in the output of this first layer. Something similiar (though with 3×3 filters) can be observed in Figure 1 where the padding of the image is chosen to be the closest pixel on the border of the image instead of 0.

The MaxPooling layer on the other hand significantly decreases the output dimensions: here we find dimensions of 14×14×6, the 6 again resulting from the use of 6 output channels. The dimensions fo 14×14 are a consequence of using no padding as well as a stride length of 2 in both directions: each entry of the output of one channel will only be used once for pooling and no padding is used; the stride and filter dimensions of 2×2 therefore halve each dimension.

1.d

We can observe the effect that padding has on the output dimensions of the convolutional layer by removing it from the first layer. The `model.summary()` for this CNN can be seen in Figure 3.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 24, 24, 6)         156
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 12, 12, 6)         0
_________________________________________________________________
flatten (Flatten)            (None, 864)               0
_________________________________________________________________
dense (Dense)                (None, 128)               110720
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 112,166
Trainable params: 112,166
Non-trainable params: 0
```

Figure 3: The internal data structure of the CNN proposed in Section 1.b without padding in the convolutional layer, created by the Keras command `model.summary()`.

As we can see, the output shape of the first layer in this example only is 24×24×6 as compared to 28×28×6 in 1.c. Since no padding is used in this case, only the pixels with coordinates $(i, j), i, j \in [3, 26]$ (counting from 1) occur as centres of a local reference field (LRF) of size 5×5. The output channel dimension of 6 remains the same.

1.e

If we now change the stride size of the first layer in the model of Section 1.b (thus keeping the padding), we can observe the following changes to the output dimensions of the first layer.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 14, 14, 6)         156
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 7, 7, 6)           0
_________________________________________________________________
flatten (Flatten)            (None, 294)               0
_________________________________________________________________
dense (Dense)                (None, 128)               37760
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 39,206
Trainable params: 39,206
Non-trainable params: 0
```

Figure 4: The internal data structure of the CNN proposed in Section 1.b with stride size 2 in each direction in the convolutional layer, created by the Keras command `model.summary()`.

In Figure 4 we can see that the output shape of the first layer has now changed to 14×14×6 even though we are still using padding. In this case, every pixel with coordinates $(i, j), i, j \in [1, 28]$ (counting from 1) where both i and j are uneven is considered as the centre of a LRF because of the stride size. We are therefore again halving the output dimensions of the first layer.

2 Evaluating different CNN architectures

2.a

Similar to [Höl19a], we will use this second part of the report to evaluate the performance of different architectures of CNNs. All of the architectures tested in the following will have at least two convolutional layers, two max pooling layers and one fully connected layer. The three architectures can be viewed in detail in Table 1.

Model 1	**Model 2**	**Model 3**
1. Input layer 2. Convolution layer, 5×5 filter, 16 output channels, stride 1, padding, activation function ReLU 3. Max pooling layer, 2×2 filter, stride 2, zero padding 4. Convolution layer, 3×3 filter, 32 output channels, stride 1, padding, activation function ReLU 5. Max pooling layer, 2×2 filter, stride 2, zero padding 6. Fully connected layer, 128 nodes, sigmoid as activation function 7. Output layer	1. Input layer 2. Convolution layer, 5×5 filter, 16 output channels, stride 1, padding, activation function ReLU 3. Max pooling layer, 2×2 filter, stride 2, zero padding 4. Convolution layer, 3×3 filter, 32 output channels, stride 1, padding, activation function ReLU 5. Max pooling layer, 2×2 filter, stride 2, zero padding 6. Fully connected layer, 256 nodes, sigmoid as activation function 7. Fully connected layer, 128 nodes, sigmoid as activation function 8. Output layer	1. Input layer 2. Convolution layer, 7×7 filter, 16 output channels, stride 1, padding, activation function ReLU 3. Max pooling layer, 2×2 filter, stride 2, zero padding 4. Convolution layer, 5×5 filter, 32 output channels, stride 1, padding, activation function ReLU 5. Max pooling layer, 2×2 filter, stride 2, zero padding 6. Fully connected layer, 256 nodes, sigmoid as activation function 7. Fully connected layer, 128 nodes, sigmoid as activation function 8. Output layer

Table 1: Three different architectures of CNNs.

The implementation of these models can be found in `exercise2a.py`. The resulting performance values of the three different architectures can be seen in Table 2. Again, the validation loss and efficiency has been used to determine the best performing model. Additionally, the loss over epochs for all three models can be observed in Figure 5.

Model	**Validation loss**	**Total parameters**	**Time (min:sec)**
1	0.04368	207,178	13:38
3	0.04466	440,906	14:35
2	0.04151	449,482	19:53

Table 2: Comparing different NN architectures with regard to average validation loss, accuracy on the test data and time taken for training (in minutes).

From Table 2 we observe that Model 3 is the best-performing, with regard to its loss value on the

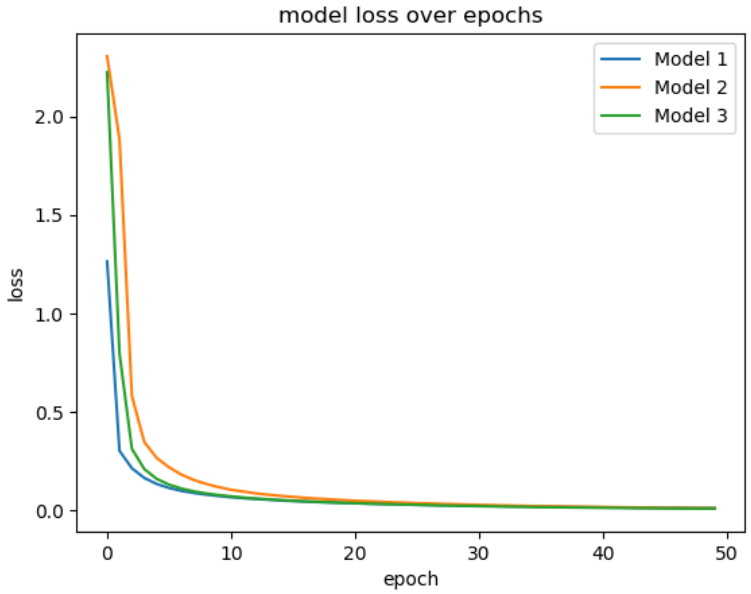

Figure 5: Plot of the evolution of loss over 50 epochs of training the three CNN architectures proposed in Table 1.

validation data but not with regard to its timing; models 1 and 2 both achieve similar validation loss and timing results. The timing values also correspond to the number of trainable parameters, of which Model 3 has the most (these can conveniently be requested through `model.summary()`). Overall, however, we can observe in Figure 5 that despite initial differences in training loss, all three models converge to similar values after approximately 25 epochs of training. Testing Model 3 on the separate test set (remember that the MNIST data base is split into 60,000 training images and 10,000 test images) leads to an accuracy of 98.96%. We can therefore conclude that this model outperforms the best NN in [Höl19a] (98% accuracy after approximately 20 minutes of training) with respect to accuracy at a similar training time.

2.b

The previous section suggested that having convolutional rather than fully connected layers in a neural network is more efficient both with respect to accuracy and training time. In this section, we will therefore test whether a deeper but lighter network, i.e. one with one more convolutional layer but less trainable parameters, can achieve similar performance results while requiring the same or less training time. The accompanying Python code can be found in `exercise2b.py` . We will leave the common parameters described in the introduction the same in order to achieve some comparability of the hyperparemeters. The chosen model has the following architecture:

1. Input layer
2. Convolution layer with a 4×4 filter and 32 output channels, stride 2, padding, followed by activation function ReLU
3. Max pooling layer with a 2×2 filter and stride 2×2, zero padding
4. Convolution layer with a 4×4 filter and 16 output channels, stride 1, padding, followed by activation function ReLU
5. Max pooling layer with a 2×2 filter and stride 2×2, zero padding

6. Convolution layer with a 2×2 filter and 9 output channels, stride 1, padding, followed by activation function ReLU
7. Max pooling layer with a 2×2 filter and stride 2×2, zero padding
8. Fully connected layer with 128 nodes, sigmoid as activation function
9. Output layer.

The `model.summary()` can be found in Figure 6; as we can see, the number of trainable parameters is at only 2.5% of those of Model 3 of Section 2.a.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 14, 14, 32)        544
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 7, 7, 32)          0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 7, 7, 16)          8208
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 3, 3, 16)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 3, 3, 9)           585
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 1, 1, 9)           0
_________________________________________________________________
flatten (Flatten)            (None, 9)                 0
_________________________________________________________________
dense (Dense)                (None, 128)               1280
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 11,907
Trainable params: 11,907
Non-trainable params: 0
```

Figure 6: The internal data structure of the CNN proposed in Section 1.b, created by the Keras command `model.summary()`.

Now to evaluate this model's performance, we will again consider the final accuracy on the test data. In general, it is advised to use as much training data as possible to fit the model and hence to refrain from using a training, validation and test dataset (split into 48,000, 12,000 and 10,000 images) and instead use a simply training and test data split and use the additional 12,000 validation images for training. Since we are not interested in further tuning the hyperparameters and we aborting the cross-validation after just one fold, it is not necessary to use a separate validation data set. Using only training and test data results in 14 minutes of training and an accuracy 98.57%. Even though the model has significantly less parameters to train, it does not take proportionally less time for training (roughly 14 minutes) as compared to Model 3 from Section 2.a and achieves slightly worse accuracy on the test data. If storage space is of any concern, then going for this deep and light architecture might be of advantage; however, on modern computers, training parameters of roughly 400,000 should be no reason for concern. Altogether, Model 3 therefore remains the best tested architecture of this report with a test accuracy of 98.96% after 14 minutes of training.

3 Conclusion

To conclude this report on the use of (C)NNs for handwritten digit recognition of the MNIST data set, we note that the use of CNNs significantly outperformed fully connected neural networks. We have seen that with comparatively little training time and a simple CNN architecture with just five layers, an accuracy of 98.96% correctly recognised digits could be achieved.

References

[GBC19] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. `http://www.deeplearningbook.org/`, 2019. – [Online; accessed 12-November-2019]

[Goo19] Google: *TensorFlow*. `https://www.tensorflow.org/`, 2019. – [Online; accessed 12-November-2019]

[Höl19a] Hölting, Jan G.: *Crash Course Deep Learning - Assignment 1*. `https://github.com/jangerrith/deep_learning/blob/master/assignments/assignment1/dl_assignment1_jangerrithoelting.pdf`, 2019. – [Online; accessed 21-November-2019]

[Höl19b] Hölting, Jan G.: *deep_learning, GitHub Repository*. `https://github.com/jangerrith/deep_learning`, 2019. – [Online; accessed 21-November-2019]

[LCB19] LeCun, Yann ; Cortes, Corinna ; Burges, Christopher: *THE MNIST DATABASE*. `http://yann.lecun.com/exdb/mnist/`, 2019. – [Online; accessed 12-November-2019]

[Nie19] Nielsen, Michael: *Neural Networks and Deep Learning*. `http://neuralnetworksanddeeplearning.com/`, 2019. – [Online; accessed 12-November-2019]

[wik19] wikipedia: *Convolutional Neural Network*. `https://de.wikipedia.org/wiki/Convolutional_Neural_Network`, 2019. – [Online; accessed 21-November-2019]