Advanced Machine Learning Report

Assignment 3

Matteo Breganni 869549

Summary

Convolutional neural network

In the following report, I will explain my thought process step by step towards creating the best possible convolutional neural network to predict the digits in the MNIST dataset, with a total parameter cap of 7000.

Note: every model uses the same batch size (128), number of epochs (15), loss function (categorial_crossentropy), optimizer (adam) and validation split (0.2).

Data preparation

To prepare the data, I have used two different processing techniques for the training set.

The first one is what we've seen in class, and it simply scales pixel values to the range 0-1 by dividing by 255 (since every instance is a grayscale image).

The second method standardizes the data by subtracting the mean and dividing by the standard deviation, making sure that the data has a mean of 0 and a standard deviation of 1 to improve the convergence speed.

I will show in a later section how these two methods compare. Regardless, in both cases the labels are one-hot encoded.

Benchmark model

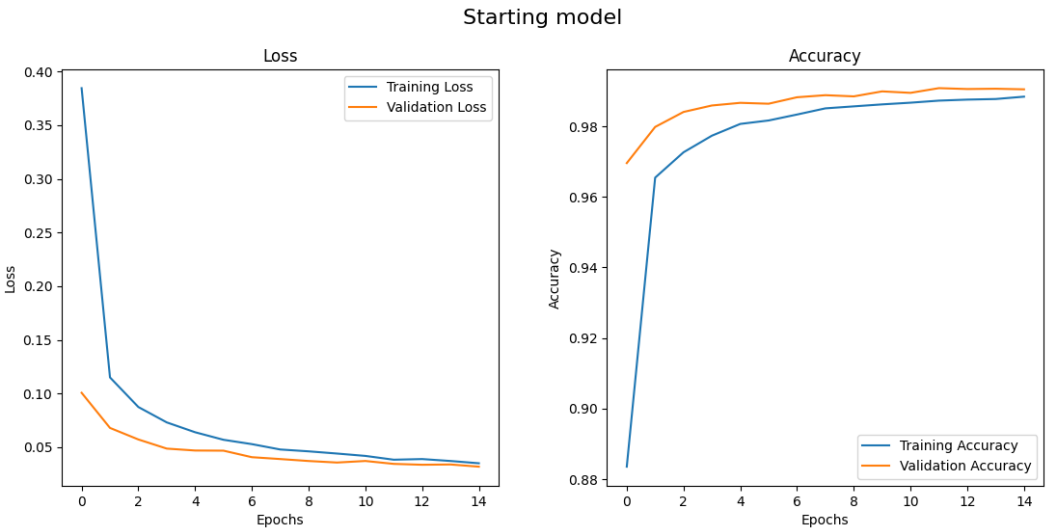
The first model is the one we created during the lab, which has 34'826 parameters. This is obviously way above the maximum allowed of 7000 parameters, but it serves the purpose of a benchmark, to compare the results of the "reduced" following models with.

```
keras.Input(shape=input_shape),
layers.Conv2D(32, kernel_size=(3,3), activation='relu'), # First convolutional layer (2D) with 32 filters, each with a 3x3 size.
                                                         # The activation is performed directly after the convolution
                                                         # By default the stride is 1
layers.MaxPooling2D(pool_size=(2,2)), # By default the stride is equal to the selected pool size
layers.Conv2D(64, kernel_size=(3,3), activation='relu'), # Second convolutional layer
layers.MaxPooling2D(pool_size=(2,2)), # Second pooling layer

layers.Flatten(), # We need to convert the output volume into a one dimensional array
layers.Dropout(0.5), # This will switch off randomly, at each iteration, 50% of the nodes. After the training they will be all turned back on
layers.Dense(num_classes, activation='softmax') # Fully connected layer that maps the size of the flatten layer into 10 neurons (num_classes),
                                                # with a softmax activation to have a probability distribution
```

Layer (type)	Output Shape	Param #
conv2d_74 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_74 (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_75 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_75 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten_29 (Flatten)	(None, 1600)	0
dropout_5 (Dropout)	(None, 1600)	0
dense_29 (Dense)	(None, 10)	16,010

Total params: 34,826 (136.04 KB)
Trainable params: 34,826 (136.04 KB)
Non-trainable params: 0 (0.00 B)



The starting model reaches this level of performance on the test set:

```
Test loss: 0.024789882823824883
Test accuracy: 0.9908999800682068
```

CNN models

Model #1

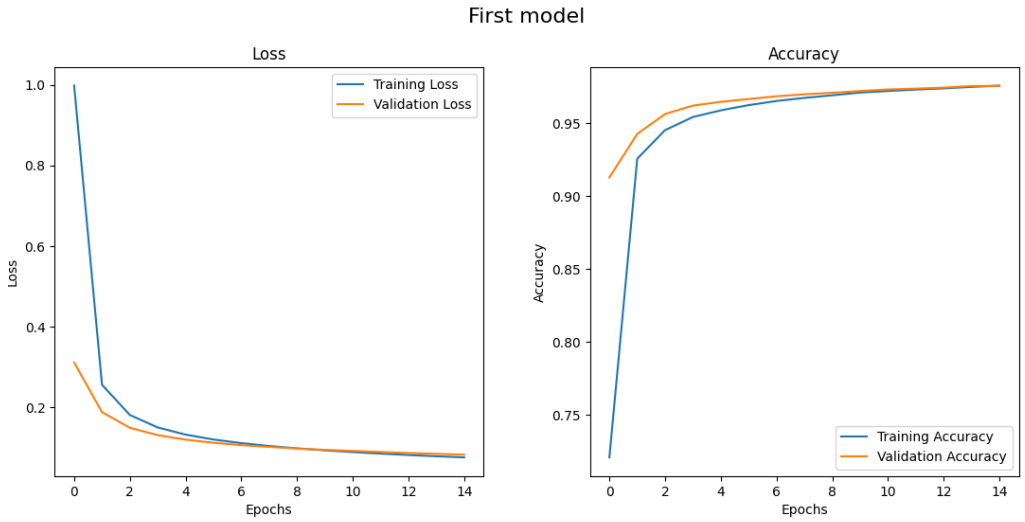
The following model is simply the first model I created with total number of parameters below 7000, this is the starting point that I will be improving on. This model uses the simple data preparation I explained earlier.

```
keras.Input(shape=input_shape),
layers.Conv2D(32, kernel_size=(3,3), activation='relu'),
layers.MaxPooling2D(pool_size=(5,5)),
layers.Conv2D(16, kernel_size=(2,2), activation='relu'),
layers.MaxPooling2D(pool_size=(2,2)),

layers.Flatten(),
layers.Dense(num_classes, activation='softmax')
```

Layer (type)	Output Shape	Param #
conv2d_76 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_76 (MaxPooling2D)	(None, 5, 5, 32)	0
conv2d_77 (Conv2D)	(None, 4, 4, 16)	2,064
max_pooling2d_77 (MaxPooling2D)	(None, 2, 2, 16)	0
flatten_30 (Flatten)	(None, 64)	0
dense_30 (Dense)	(None, 10)	650

Total params: 3,034 (11.85 KB)
Trainable params: 3,034 (11.85 KB)
Non-trainable params: 0 (0.00 B)



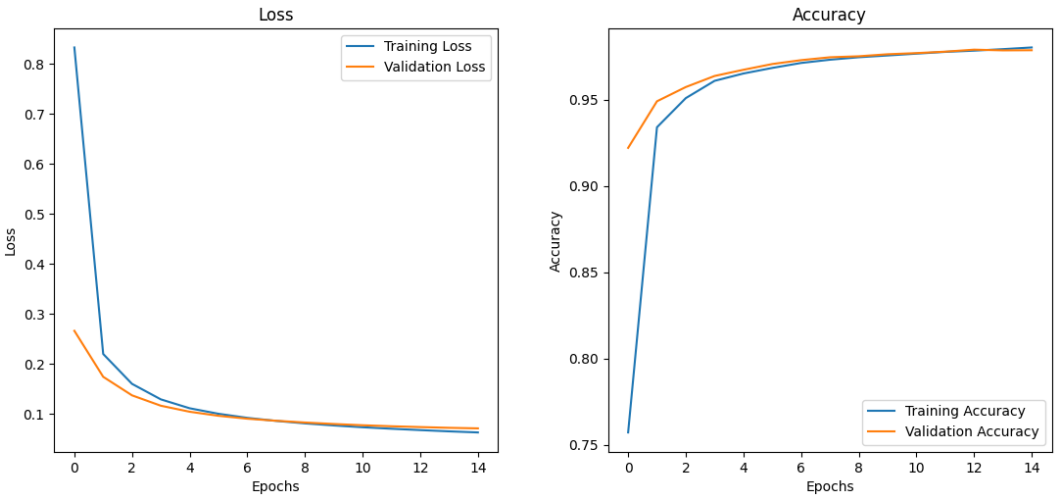
The starting model reaches this level of performance on the test set:

```
Test loss: 0.07468102872371674
Test accuracy: 0.9779000282287598
```

This level of accuracy is already quite good, but there is still room for improvement.

Model #2

The second model has the same structure as the first model, but it uses the normalized (0-centered, std 1) training set. I made this model to verify that this strategy does improve the model's performance. All the following models also use the 0-centered normalized dataset.



This confirms the accuracy does improve (possibly simply because it converges faster).

```
Test loss: 0.06037464737892151
Test accuracy: 0.980400025844574
```

The accuracy is already very close to the benchmark's.

Model #3

In the third model, the strategy is to modify the previous model by adding more parameters and getting as close to 7000 as possible.

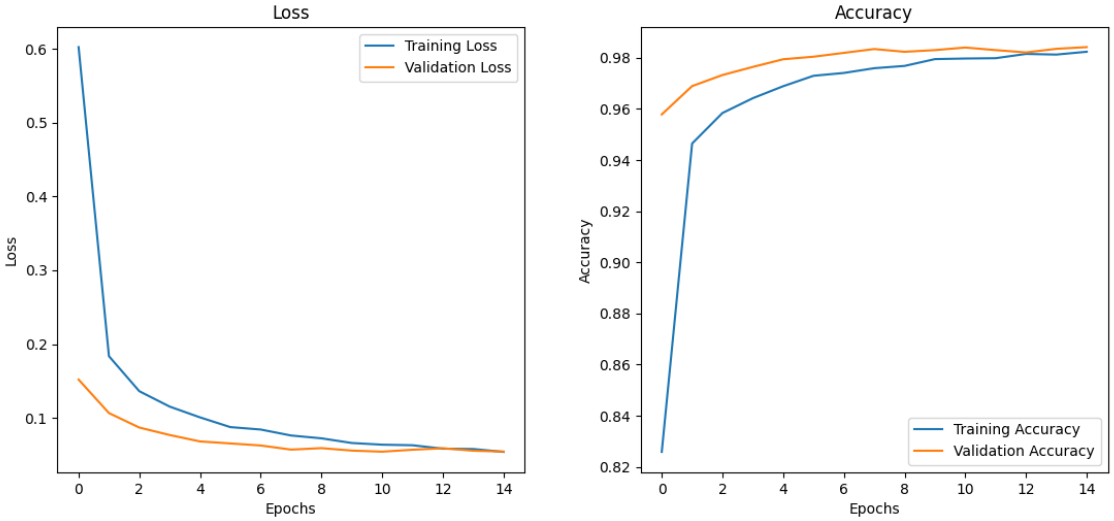
I also had to add a bit of dropout to prevent overfitting the model.

```
keras.Input(shape=input_shape),
layers.Conv2D(32, kernel_size=(9,9), activation='relu'),
layers.MaxPooling2D(pool_size=(6,6)),
layers.Conv2D(31, kernel_size=(2,2), activation='relu'),
layers.MaxPooling2D(pool_size=(2,2)),

layers.Flatten(),
layers.Dropout(0.1),
layers.Dense(num_classes, activation='softmax')
```

Layer (type)	Output Shape	Param #
conv2d_196 (Conv2D)	(None, 20, 20, 32)	2,624
max_pooling2d_195 (MaxPooling2D)	(None, 3, 3, 32)	0
conv2d_197 (Conv2D)	(None, 2, 2, 31)	3,999
max_pooling2d_196 (MaxPooling2D)	(None, 1, 1, 31)	0
flatten_90 (Flatten)	(None, 31)	0
dropout_9 (Dropout)	(None, 31)	0
dense_90 (Dense)	(None, 10)	320

Total params: 6,943 (27.12 KB)
Trainable params: 6,943 (27.12 KB)
Non-trainable params: 0 (0.00 B)



By maximizing the parameters, the accuracy got even closer to the benchmark's.

```
Test loss: 0.04872306063771248
Test accuracy: 0.9853000044822693
```

Model #4

In the fourth model, the strategy is to modify the previous model by using smaller kernels and fewer filters in each layer, while adding more layers.

I took inspiration from AlexNet and VGG, using filters that decrease in size as the model gets deeper, while the number of filters (depth) grows. This is important, because while the spatial information gets destroyed, it gets "encoded" into the growing depth of the filters.

This method seems to be what worked best, I have tried several different configurations, but from what I found it's very important to find a good balance between the number of parameters in the convolutional side of the network and the fully connected side. If too few parameters were to be used in the fully connected side (for instance by reducing its input to a 1x1 matrix, before flattening it), that would create a model with a significantly lower accuracy. Likewise, growing the parameters of the fully connected part too much would leave too few parameters for the convolutional side, and that would make the performance worse.

The number of filters is also very important, if left too small (especially in the last layers, because they need a higher level of abstraction) the performance would not be great.

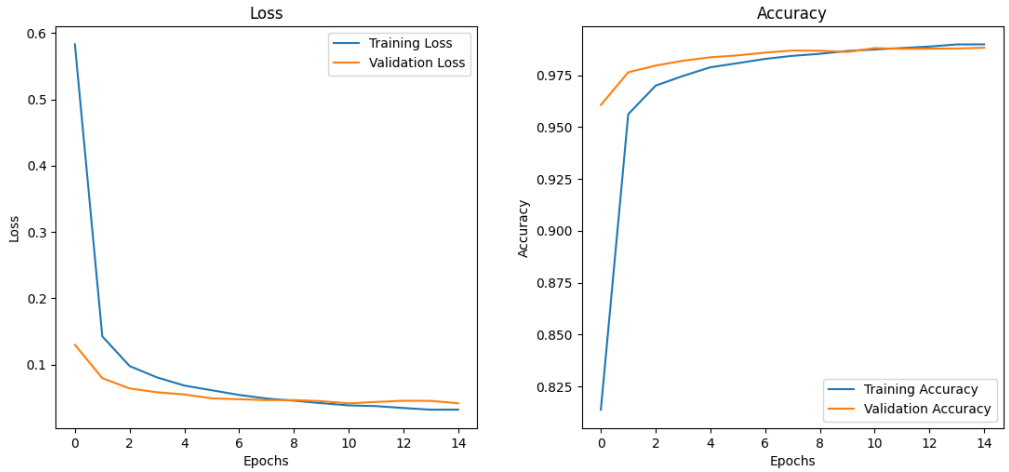
I also tried using AveragePooling2D instead of MaxPooling2D but I found that the performance was pretty much identical (slight variations are normal, since I'm using dropout).

```
keras.Input(shape=input_shape),
layers.Conv2D(8, kernel_size=(5,5), activation='relu'),
layers.Conv2D(16, kernel_size=(3,3), activation='relu'),
layers.MaxPooling2D(pool_size=(3,3)),
layers.Conv2D(16, kernel_size=(3,3), activation='relu'),
layers.Conv2D(31, kernel_size=(2,2), activation='relu'),
layers.MaxPooling2D(pool_size=(2,2)),

layers.Flatten(),
layers.Dropout(0.1),
layers.Dense(num_classes, activation='softmax')
```

Layer (type)	Output Shape	Param #
conv2d_241 (Conv2D)	(None, 24, 24, 8)	208
conv2d_242 (Conv2D)	(None, 22, 22, 16)	1,168
max_pooling2d_223 (MaxPooling2D)	(None, 7, 7, 16)	0
conv2d_243 (Conv2D)	(None, 5, 5, 16)	2,320
conv2d_244 (Conv2D)	(None, 4, 4, 31)	2,015
max_pooling2d_224 (MaxPooling2D)	(None, 2, 2, 31)	0
flatten_104 (Flatten)	(None, 124)	0
dropout_22 (Dropout)	(None, 124)	0
dense_104 (Dense)	(None, 10)	1,250

Total params: 6,961 (27.19 KB)
Trainable params: 6,961 (27.19 KB)
Non-trainable params: 0 (0.00 B)



This result is extremely close to the benchmark's.

```
Test loss: 0.034649841487407684
Test accuracy: 0.9901999831199646
```

Model #5 (extra)

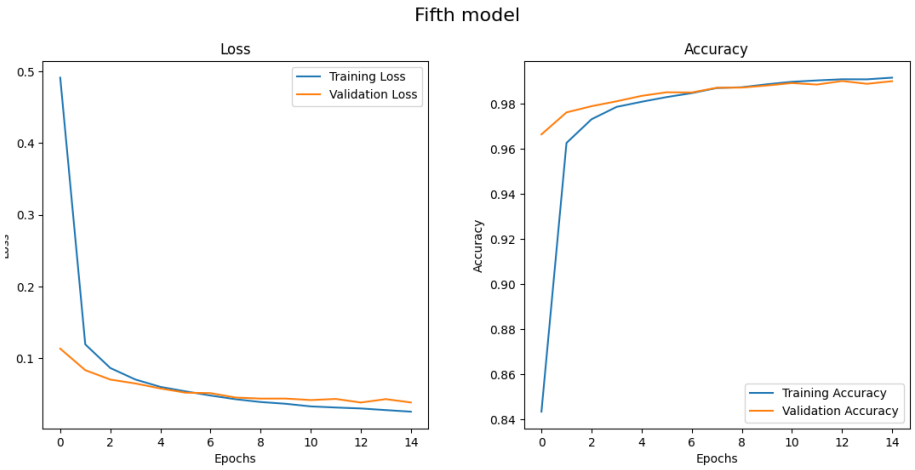
The one modification of the last model that is worth noting (because it works), is using the "*padding='same'*" option in the convolutional layers. This setting applies the filter differently, using a padding of zeros around the edges. However, the network can't be kept identical because this technique adds more parameters (so they would become more than 7000).

```
keras.Input(shape=input_shape),
layers.Conv2D(6, kernel_size=(5,5), activation='relu', padding='same'),
layers.Conv2D(11, kernel_size=(3,3), activation='relu', padding='same'),
layers.MaxPooling2D(pool_size=(3,3)),
layers.Conv2D(12, kernel_size=(3,3), activation='relu', padding='same'),
layers.Conv2D(24, kernel_size=(2,2), activation='relu', padding='same'),
layers.MaxPooling2D(pool_size=(2,2)),

layers.Flatten(),
layers.Dropout(0.10),
layers.Dense(num_classes, activation='softmax')
```

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 28, 28, 6)	156
conv2d_3 (Conv2D)	(None, 28, 28, 11)	605
max_pooling2d_2 (MaxPooling2D)	(None, 9, 9, 11)	0
conv2d_4 (Conv2D)	(None, 9, 9, 12)	1,200
conv2d_5 (Conv2D)	(None, 9, 9, 24)	1,176
max_pooling2d_3 (MaxPooling2D)	(None, 4, 4, 24)	0
flatten_1 (Flatten)	(None, 384)	0
dropout_1 (Dropout)	(None, 384)	0
dense_1 (Dense)	(None, 10)	3,850

Total params: 6,987 (27.29 KB)
Trainable params: 6,987 (27.29 KB)
Non-trainable params: 0 (0.00 B)




```
Test loss: 0.03073572926223278
Test accuracy: 0.989799976348877
```

The accuracy is very similar to the last model, usually slightly worse (depending on the randomness of the dropouts).

Results recap

	Total Parameters	Preprocessing method	Test accuracy	Notes
Model #1	3'034	Scaling (0–1)	97,79%	Baseline with simple preprocessing
Model #2	3'034	0-centered normalization	98,04%	Same structure, with normalization
Model #3	6'943	0-centered normalization	98,53%	More parameters, added dropout
Model #4	6'961	0-centered normalization	99,01%	Optimized architecture
Model #5	6,987	0-centered normalization	98,97%	Alternative model with padding
Benchmark	34'826	0-centered normalization	99,08%	/

Conclusions

In conclusion, the performance achieved by the fourth model (in particular) is extremely similar to the benchmark model's, while using around 27'000 parameters less. Of course, improvements could be made by trying different loss functions, optimizers, or simply training for more epochs. Data augmentation could help, just like using a custom stride value for the filters (to reduce the number of parameters that could be used somewhere else), but I haven't tried them since we haven't seen them in the labs yet and it seemed outside of the scope of the assignment.

Although it can be assumed that by further improving the fourth model (or the benchmark model), higher levels of accuracy could be reached, I find this result (99% accuracy) extremely satisfying in relation to this assignment's objective. It is quite clear that with a limited amount of parameters, balancing the size of the network's two parts is extremely important, as is having an architecture that lowers the size of the images while growing the depth of the filters.