Report: Introduction to PyTorch

Computer Vision

Assignment 1

Ian SCHOLL
ischoll@student.ethz.ch

2 Simple 2D classifier (50 pts.)

2.1 Dataset (10 pts.)

My constructor uses the *split* input to accordingly load the corresponding *.npz* file in the 'data' folder. The 'samples' column with 1x2 vectors with the sample coordinates then gets saved into *self.samples*, and the 'annotations' column with the corresponding cluster labeling of each sample gets saved into *self.annotations*:

```
data_path = os.path.join('data', f'{split}.npz')
data = np.load(data_path)
self.samples = data['samples']
self.annotations = data['annotations']
```

My *__getitem__* method then simply goes and takes the sample coordinates and according label at index *idx* out of the variables created in the constructor:

```
def __getitem__(self, idx):
    # Returns the sample and annotation with index idx.
    sample = self.samples[idx]
    annotation = self.annotations[idx]
```

2.2 Linear classifier (10 pts.)

A single line is enough to define the linear layer taking the 1x2 coordinates as input and computing a single value as output:

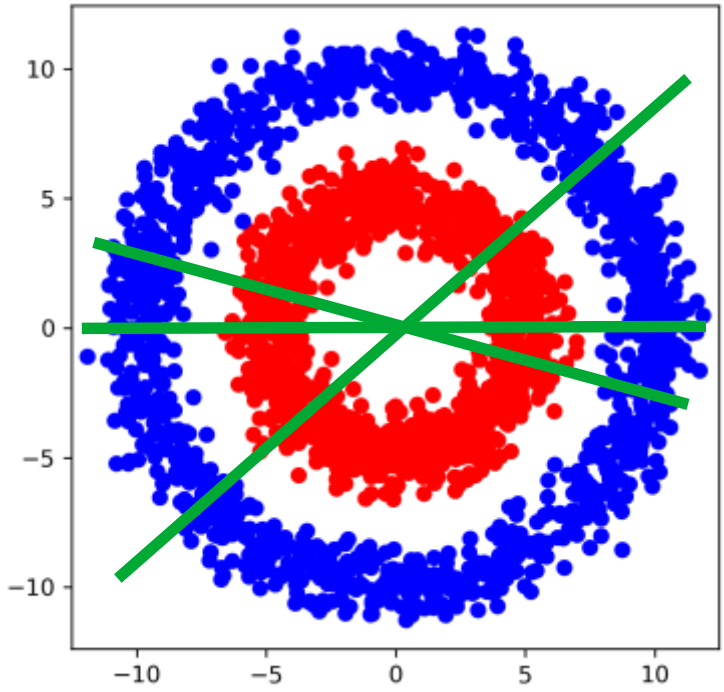
```
self.layers = nn.Sequential(
    nn.Linear(2, 1)
)
```

2.3 Training loop (15 pts.)

```
[Epoch 01] Loss: 1.1414
[Epoch 01] Acc.: 49.0079%
[Epoch 02] Loss: 0.7673
[Epoch 02] Acc.: 49.0079%
[Epoch 03] Loss: 0.6980
[Epoch 03] Acc.: 52.5794%
[Epoch 04] Loss: 0.6936
[Epoch 04] Acc.: 52.3810%
[Epoch 05] Loss: 0.6931
[Epoch 05] Acc.: 51.5873%
[Epoch 06] Loss: 0.6931
[Epoch 06] Acc.: 50.3968%
[Epoch 07] Loss: 0.6929
[Epoch 07] Acc.: 50.9921%
[Epoch 08] Loss: 0.6934
[Epoch 08] Acc.: 49.8016%
[Epoch 09] Loss: 0.6930
[Epoch 09] Acc.: 49.4048%
[Epoch 10] Loss: 0.6929
[Epoch 10] Acc.: 48.6111%
```

Once the training loop is implemented as instructed in the *run_training_epoch* function, the model can be trained and put to the test for the first time, revealing the following results on the figure on the right:

We can observe that the predictions and therefore the accuracy of the model don't improve over the epochs, and that the accuracy always lies somewhere around 50% (even when using significantly more epochs). This is indeed an expected result, as trying to approximate a radially distributed and divided cluster linearly is nearly impossible. This is illustrated in the following:



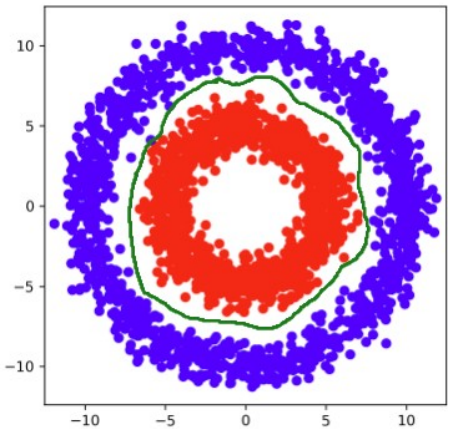
Using solely a linear layer can be visualized with the model trying to fit a straight line as the exemplary ones in green in the depiction above, in order to divide the red cluster from the blue one. However, this will obviously always fail. On average, about half of both the training and validation samples for both clusters will end up on either side of the 'division line', which is why we observe an accuracy of around 50% for all epochs.

2.4 Multi-layer perceptron (10 pts.)

```
[Epoch 01] Loss: 0.5881
[Epoch 01] Acc.: 94.8413%
[Epoch 02] Loss: 0.2547
[Epoch 02] Acc.: 99.8016%
[Epoch 03] Loss: 0.0850
[Epoch 03] Acc.: 99.8016%
[Epoch 04] Loss: 0.0394
[Epoch 04] Acc.: 100.0000%
[Epoch 05] Loss: 0.0228
[Epoch 05] Acc.: 100.0000%
[Epoch 06] Loss: 0.0155
[Epoch 06] Acc.: 100.0000%
[Epoch 07] Loss: 0.0114
[Epoch 07] Acc.: 100.0000%
[Epoch 08] Loss: 0.0094
[Epoch 08] Acc.: 99.8016%
[Epoch 09] Loss: 0.0074
[Epoch 09] Acc.: 100.0000%
[Epoch 10] Loss: 0.0056
[Epoch 10] Acc.: 100.0000%
```

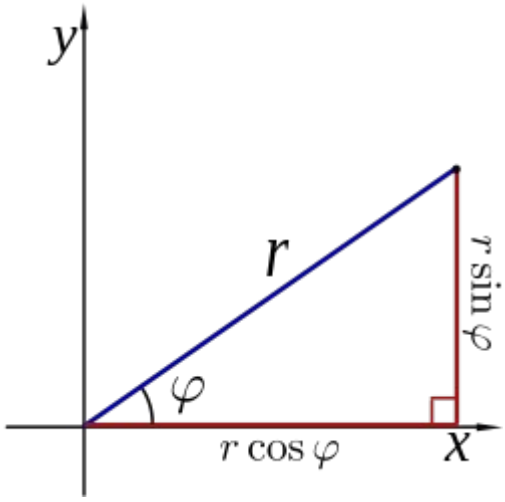
The multi-layer classifier shows a lot better results. We can observe accuracies of even 99.8 - 100% over almost all epochs, and losses are divided by a factor of 100 as shown in the figure on the right:

Now, apart from using **more** linear layers, the main difference is the introduction of ReLU **non-linearities** into the network. This will immediately allow for the model to tune more parameters to accurately attribute some new point to one of the clusters, as well as find a distinction that might not be a straight line anymore. Much rather, the delimitation found between the two clusters is now more 'flexible' and could be ideally visualized as the green line below:



2.5 Feature transform (5 pts.)

Inspired by the ideal delimitation of subchapter 2.4, one can immediately think of a different coordinate system, which are 2D polar coordinates as shown on the right:

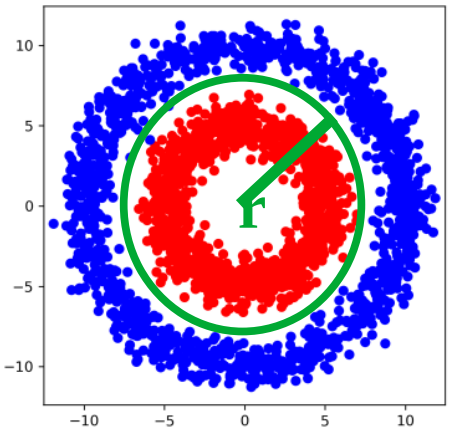


source: *https://en.wikipedia.org/wiki/Polar_coordinate_system*

Now, a sample point has coordinates given as a pair:

(r, φ)

Therefore, drawing a 'straight' line through all points with a certain coordinate r (of roughly 7 in our example) – which will translate to a circle of radius r in our original x-y coordinate system - should allow the model to again delimit the clusters as in the figure on the left:



Therefore, using a single linear layer now allows the model to make good predictions and achieve high accuracies of over 90% as can be seen in the results on the right:

```
[Epoch 01] Loss: 0.4799
[Epoch 01] Acc.: 77.3810%
[Epoch 02] Loss: 0.4514
[Epoch 02] Acc.: 78.7698%
[Epoch 03] Loss: 0.4316
[Epoch 03] Acc.: 80.5556%
[Epoch 04] Loss: 0.4129
[Epoch 04] Acc.: 81.7460%
[Epoch 05] Loss: 0.3966
[Epoch 05] Acc.: 82.9365%
[Epoch 06] Loss: 0.3804
[Epoch 06] Acc.: 84.3254%
[Epoch 07] Loss: 0.3663
[Epoch 07] Acc.: 84.9206%
[Epoch 08] Loss: 0.3504
[Epoch 08] Acc.: 86.5079%
[Epoch 09] Loss: 0.3370
[Epoch 09] Acc.: 88.4921%
[Epoch 10] Loss: 0.3238
[Epoch 10] Acc.: 90.0794%
```

If we use more epochs than just 10, we can even get the accuracy close to 100%. This is a clear improvement w.r.t. the single linear layer used on the x-y coordinates in 2.3 (cost also divided by 2), and the evolution over the epochs shows how the model actually gets optimized step by step and is not making predictions of pure '50-50-chance' anymore.

3 Digit classifier (50 pts.)

3.1 Data normalization (5 pts.)

A simple mathematical calculation allows to project values in the range {0,255} to values in the range [-1, 1]:

```
def normalize(sample):
    new_sample = ((sample/127.5) - 1)
    # print(new_sample)
    return new_sample
```

3.2 Training loop (5 pts.)

I followed the same steps as in subchapter 2.3, however this time I used the cross entropy loss as instructed:

```
# Compute the loss - cross entropy.
# Documentation https://pytorch.org/docs/stable/generated/torch.nn.functional.cross_entropy.html.
loss = F.cross_entropy(output, batch['annotation'])
```

3.3 Multi-layer perceptron (10 pts.)

Using just a single linear layer in order to implement a linear classifier, the *nn.Sequential* looks as follows:

```
# Define the network layers in order.
# Input is 28 * 28.
# Output is 10 values (one per class).
# Multiple linear layers each followed by a ReLU non-linearity (apart from the last).
self.layers = nn.Sequential(
    nn.Linear(28*28, 10)
    # nn.Linear(28*28, 32),
    # nn.ReLU(),
    # nn.Linear(32, 10)
)
```

This gives me the following results:

```
100%|| 3750/3750 [00:02<00:00, 1558.85it/s]
[Epoch 01] Loss: 0.4112
[Epoch 01] Acc.: 89.1400%
100%|| 3750/3750 [00:02<00:00, 1540.89it/s]
[Epoch 02] Loss: 0.3331
[Epoch 02] Acc.: 90.3700%
100%|| 3750/3750 [00:02<00:00, 1587.41it/s]
[Epoch 03] Loss: 0.3209
[Epoch 03] Acc.: 91.6600%
100%|| 3750/3750 [00:02<00:00, 1610.98it/s]
[Epoch 04] Loss: 0.3140
[Epoch 04] Acc.: 91.5300%
100%|| 3750/3750 [00:02<00:00, 1577.74it/s]
[Epoch 05] Loss: 0.3134
[Epoch 05] Acc.: 91.5000%
```

This is not a bad start, but definitely not yet fully satisfying, as we would like an accuracy closer to 100%.

Now I will use a *nn.Sequential* as instructed with 1 hidden layer of dimension 32 with ReLU and a final linear prediction layer:

```
# Define the network layers in order.
# Input is 28 * 28.
# Output is 10 values (one per class).
# Multiple linear layers each followed by a ReLU non-linearity (apart from the last).
self.layers = nn.Sequential(
    # nn.Linear(28*28, 10)
    nn.Linear(28*28, 32),
    nn.ReLU(),
    nn.Linear(32, 10)
)
```

Using several layers and non-linearities again clearly improves the results, the accuracy approaches the 100% more and losses are halved:

```
100%|                                                                                | 3750/3750 [00:03<00:00, 1107.06it/s]
[Epoch 01] Loss: 0.3973
[Epoch 01] Acc.: 91.0600%
100%|                                                                                | 3750/3750 [00:03<00:00, 1088.76it/s]
[Epoch 02] Loss: 0.2627
[Epoch 02] Acc.: 92.0800%
100%|                                                                                | 3750/3750 [00:03<00:00, 1022.66it/s]
[Epoch 03] Loss: 0.2164
[Epoch 03] Acc.: 93.8500%
100%|                                                                                | 3750/3750 [00:03<00:00, 1036.33it/s]
[Epoch 04] Loss: 0.1896
[Epoch 04] Acc.: 94.7700%
100%|                                                                                | 3750/3750 [00:03<00:00, 1021.50it/s]
[Epoch 05] Loss: 0.1734
[Epoch 05] Acc.: 95.1000%
```

This is both what we hoped for and expected. Introducing more layers on one hand raises the number of parameters that can be tuned in order to improve the model quickly and effectively. Introducing non-linearities through ReLU on the other hand also makes the model less ‘stiff’, as it does not have to completely stick to the limiting idea of full linearity anymore.

3.4 Convolutional network (10 pts.)

The convolutional network takes on a slightly more complicated form and can be seen in my implementation in the *nn.Sequential* as follows:

```
# Define the network layers in order.
# Input is 28x28, with one channel.
# Multiple Conv2d and MaxPool2d layers each followed by a ReLU non-linearity (apart from the last).
# Needs to end with AdaptiveMaxPool2d(1) to reduce everything to a 1x1 image.
self.layers = nn.Sequential(
    nn.Conv2d(1, 8, 3),
    nn.ReLU(),
    nn.MaxPool2d(2, stride=2),
    nn.Conv2d(8, 16, 3),
    nn.ReLU(),
    nn.MaxPool2d(2, stride=2),
    nn.Conv2d(16, 32, 3),
    nn.ReLU(),
    nn.AdaptiveMaxPool2d(1)
)
# Linear classification layer.
# Output is 10 values (one per class).
self.classifier = nn.Sequential(
    nn.Linear(32, 10)
)
```

The accuracy that I obtain with this model is even better, which can be seen in the results:

```
100%|                                                                                | 3750/3750 [00:07<00:00, 511.16it/s]
[Epoch 01] Loss: 0.2643
[Epoch 01] Acc.: 96.9700%
100%|                                                                                | 3750/3750 [00:07<00:00, 518.72it/s]
[Epoch 02] Loss: 0.0879
[Epoch 02] Acc.: 98.1000%
100%|                                                                                | 3750/3750 [00:07<00:00, 488.94it/s]
[Epoch 03] Loss: 0.0651
[Epoch 03] Acc.: 98.0300%
100%|                                                                                | 3750/3750 [00:07<00:00, 478.28it/s]
[Epoch 04] Loss: 0.0526
[Epoch 04] Acc.: 98.0300%
100%|                                                                                | 3750/3750 [00:08<00:00, 457.19it/s]
[Epoch 05] Loss: 0.0458
[Epoch 05] Acc.: 98.5000%
```

This means, using a full convolutional network instead of a Multi-Layer Perceptron can raise the accuracies and decrease the losses even more, as this should be a more suited technique especially for learning on images than using just linear layers and ReLU.

3.5 Comparison of number of parameters (10 pts.)

(The formulas used in this subchapter have been taken from the following source (20.10.2021): https://learnopencv.com/number-of-parameters-and-tensor-sizes-in-convolutional-neural-network/)

It can first be stated that ReLU(), AdaptiveMaxPool2D() and MaxPool2D() have no parameters associated with them, if anything they only use hyperparameters.

MLP

The number of parameters for the MLP can be computed using the following formulas:

W_{ff} = Number of weights of a FC Layer which is connected to an FC Layer.
B_{ff} = Number of biases of a FC Layer which is connected to an FC Layer.
P_{ff} = Number of parameters of a FC Layer which is connected to an FC Layer.
F = Number of neurons in the FC Layer.
F_{-1} = Number of neurons in the previous FC Layer.

$$W_{ff} = F_{-1} \times F$$
$$B_{ff} = F$$
$$P_{ff} = W_{ff} + B_{ff}$$

Hidden Layer: W_ff = 784 * 32 = 25088; B_ff = 32; → P_ff = 25088 + 32 = 25120

Final Layer: W_ff = 32 * 10 = 320; B_ff = 10; → P_ff = 320 + 10 = 330

Total number of Parameters: P_tot = 25120 + 330 = **25450**

This can be back-checked with some lines of code, which indeed reveals the same number of parameters (multiplying all shown array sizes and adding them up):

```
# Create the network.
net = MLPClassifier()
# net = ConvClassifier()

print(net)
for name, param in net.named_parameters():
    if param.requires_grad:
        print(name, param.data.shape)
```

```
MLPClassifier(
  (layers): Sequential(
    (0): Linear(in_features=784, out_features=32, bias=True)
    (1): ReLU()
    (2): Linear(in_features=32, out_features=10, bias=True)
  )
)
layers.0.weight torch.Size([32, 784])
layers.0.bias torch.Size([32])
layers.2.weight torch.Size([10, 32])
layers.2.bias torch.Size([10])
```

CNN

W_c = Number of weights of the Conv Layer.
B_c = Number of biases of the Conv Layer.
P_c = Number of parameters of the Conv Layer.
K = Size (width) of kernels used in the Conv Layer.
N = Number of kernels.
C = Number of channels of the input image.

$$W_c = K^2 \times C \times N$$
$$B_c = N$$
$$P_c = W_c + B_c$$

Convolutional Layer 1 (2D Convolution): W_c = 3² * 1 * 8 = 72; B_c = 8; → P_c = 80

Convolutional Layer 2 (2D Convolution): W_c = 3² * 8 * 16 = 1152; B_c = 16; → P_c = 1168

Convolutional Layer 3 (2D Convolution): W_c = 3² * 16 * 32 = 4608; B_c = 32; → P_c = 4640

W_{cf} = Number of weights of a FC Layer which is connected to a Conv Layer.
B_{cf} = Number of biases of a FC Layer which is connected to a Conv Layer.
O = Size (width) of the output image of the previous Conv Layer.
N = Number of kernels in the previous Conv Layer.
F = Number of neurons in the FC Layer.

$$W_{cf} = O^2 \times N \times F$$
$$B_{cf} = F$$
$$P_{cf} = W_{cf} + B_{cf}$$

Final Classifier Layer (Linear): W_cf = 1² * 32 * 10 = 320; B_cf = 10; → P_cf = 320 + 10 = 330

Total number of Parameters: P_tot = 80 + 1168 + 4640 + 330 = **6218**

Again, this can be back-checked with some small code, which reveals the same number of total parameters (multiplying all shown array sizes and adding them up):

```
# Create the network.
# net = MLPClassifier()
net = ConvClassifier()

print(net)
for name, param in net.named_parameters():
    if param.requires_grad:
        print(name, param.data.shape)
```

```
ConvClassifier(
  (layers): Sequential(
    (0): Conv2d(1, 8, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
    (7): ReLU()
    (8): AdaptiveMaxPool2d(output_size=1)
  )
  (classifier): Sequential(
    (0): Linear(in_features=32, out_features=10, bias=True)
  )
)
layers.0.weight torch.Size([8, 1, 3, 3])
layers.0.bias torch.Size([8])
layers.3.weight torch.Size([16, 8, 3, 3])
layers.3.bias torch.Size([16])
layers.6.weight torch.Size([32, 16, 3, 3])
layers.6.bias torch.Size([32])
classifier.0.weight torch.Size([10, 32])
classifier.0.bias torch.Size([10])
```

Conclusion: Therefore, we see that using a neural network with convolutions doesn't only improve our model and predictions, but it at the same time also decreases the number of parameters to be calculated and stored. Therefore we get better performance for less memory use, showing the power of convolutional neural networks!

3.6 Confusion matrix (10 pts.)

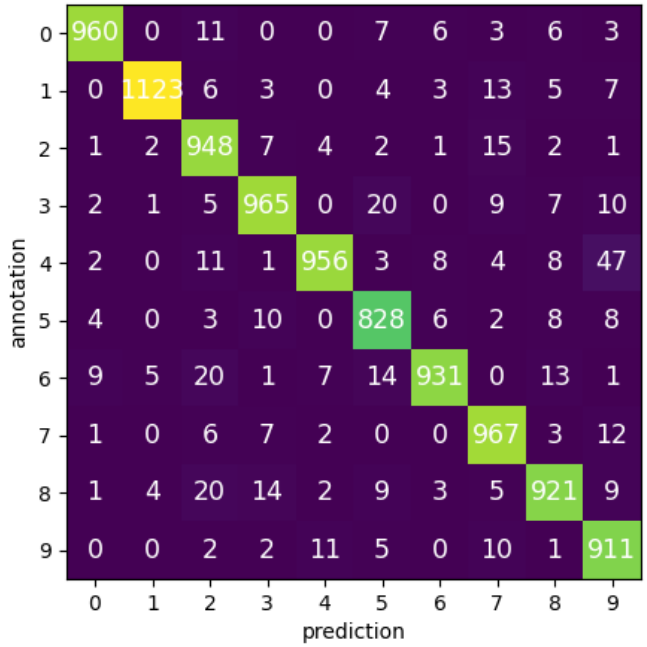
Some code of the *run_validation_epoch* function combined with new lines of code can be used to calculate the confusion matrix' elements as follows:

```python
# Based on run_validation_epoch, write code for computing the 10x10 confusion matrix.
confusion_matrix = np.zeros([10, 10])
for batch in valid_dataloader:
    # Forward pass only.
    data_out = net(batch['input'])
    ground_truth = batch['annotation']
    data_shape = data_out.shape

    # Compute the accuracy using compute_accuracy.
    # content of compute_accuracy
    for k in range(data_shape[0]):
        i = torch.argmax(data_out[k])
        j = ground_truth[k]
        confusion_matrix[i][j] += 1
```

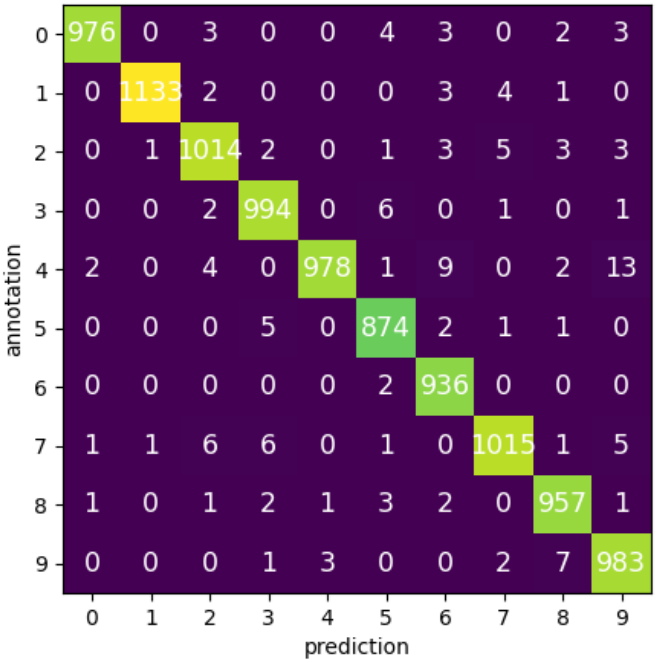
After implementing the suggested code change on Moodle (*L61 to plt.text(i, j, '%d' % (confusion_matrix[j, i]), ha='center', va='center', color='w', fontsize=12.5);* https://moodle-app2.let.ethz.ch/mod/forum/discuss.php?d=87233#p186459*),* running this code can be used to compute the final confusion matrix.

In the case of the **MLP** the confusion matrix looks as follows:



Indeed we can observe an almost diagonal matrix, with a few outliers in the non-diagonal elements, where the highest wrong-prediction number reaches 47 (a number 4 predicted as a number 9, which is understandable as handwritten they look rather similar). This shows that the model's predictions are very good and also visually tells us, that the accuracy indeed must be rather close to 100%.

In the case of the **Convolutional Network** the confusion matrix looks as follows:



Again we can observe this time an almost perfectly diagonal matrix with many 0-elements and a few outliers in the non-diagonal elements, where the highest wrong-prediction number now reaches only 13 (again the case of a number 4 predicted as a number 9, which is understandable as handwritten they look rather similar). This shows that the model's predictions are very good and also visually tells us, that the accuracy indeed now must be *very* close to 100% indeed.