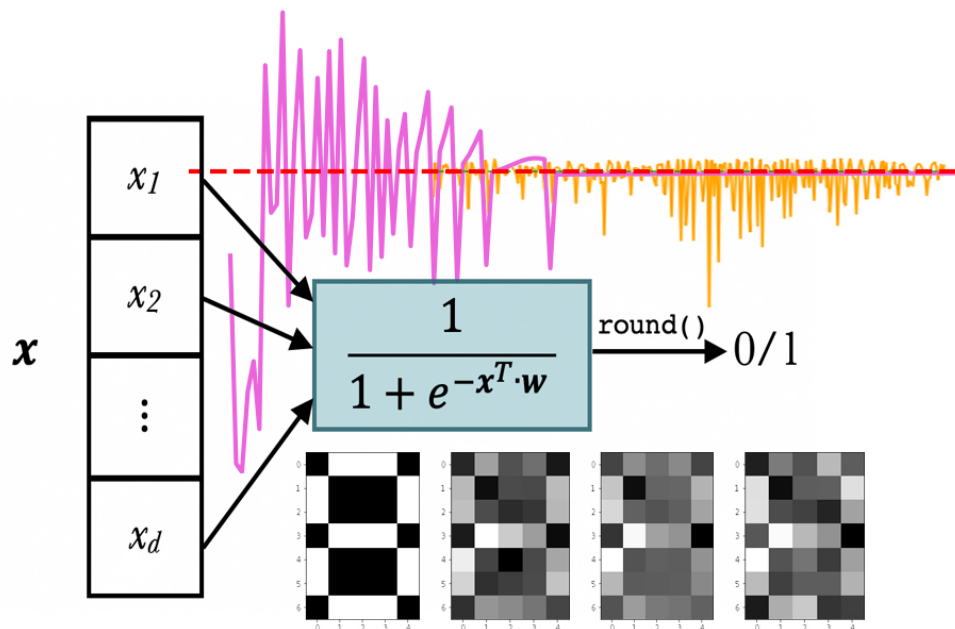


# PATTERN RECOGNITION

## WITH SINGLE LAYER NEURAL NETWORKS

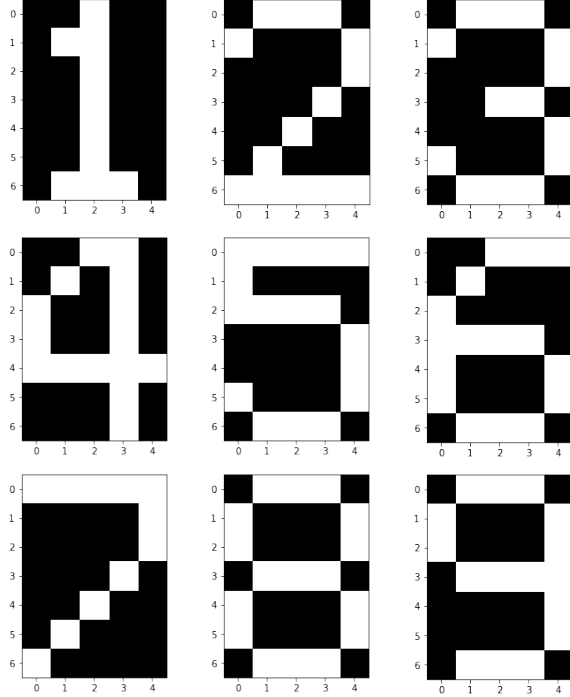


Margarita Geleta – Ibrar Malik

Mathematical Optimization | Data Science

## About the project

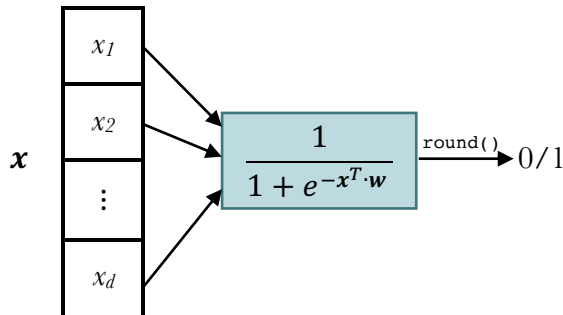
The aim of the project is to build a *Single Layer Neural Network* (abbreviated as *SLNN*) from zero in *Python* that would be capable of recognizing a set of target numbers. The numbers from the  $[0,9]$  interval will be used as targets. Each number has a  $7 \times 5$  pixel matrix representation, with values 0 and 1 (figure 1).



**Figure 1:** Pixel matrix representation of  $[1,9]$  numbers.

## Network architecture

Our neural network is a *perceptron*<sup>[2][4]</sup>, a single neuron model with  $d$  entries, 35 in our case – all the pixels of a number stored in the input vector  $\mathbf{x}$ . Then,  $\mathbf{x}$  is combined with the vector of weights  $\mathbf{w}$  and finally, the *sigmoid function* is applied to obtain a binary output 0/1.



**Figure 2:** Architecture of a perceptron (SLNN) with a sigmoidal activation function.

So, our neural network ( $y$ ) is defined as:

$$y(\mathbf{x}, \mathbf{w}) := \frac{1}{1 + e^{-\sum_{i=1}^n w_i \sigma(x_i)}}$$

We define it in *Python* as follows:

```
def sigmoid(x):
    return 1. / (1. + np.exp(-x))
def y(X, w):
    return sigmoid((sigmoid(X) @
                      w.T))
```

## Network training

Now the big question is: how do we get our neural network to learn? It is going to learn from errors. Thus, we need to define an “*error/loss function*”<sup>[2]</sup>. Before formulating that function, note that we are going to use a set of numbers to train our network and a separate test for testing the predictions. These sets are called *training* ( $\mathbf{X}^{TR}, \mathbf{y}^{TR}$ ) and *test* ( $\mathbf{X}^{TE}, \mathbf{y}^{TE}$ ) sets, respectively.

The loss function is defined as follows:

$$L(\mathbf{X}^{TR}, \mathbf{y}^{TR}) = \sum_{j=1}^p (y(\mathbf{x}_j^{TR}, \mathbf{w}) - \mathbf{y}_j^{TR})^2$$

being  $p$  the size of the training set. We can also add a regularization parameter  $\lambda$  (of type  $L2$ ):

$$\tilde{L}(\mathbf{X}^{TR}, \mathbf{y}^{TR}) = L(\mathbf{X}^{TR}, \mathbf{y}^{TR}) + \lambda \cdot \frac{\|\mathbf{w}\|^2}{2}$$

We define them in *Python* in one method. If  $\lambda = 0$ , then we return  $L(\mathbf{X}^{TR}, \mathbf{y}^{TR})$ , otherwise we return  $\tilde{L}(\mathbf{X}^{TR}, \mathbf{y}^{TR})$  with its corresponding regularization parameter  $\lambda$ :

```
def loss(w, X^TR, y^TR, lambda=0):
    return np.linalg.norm(y(X^TR, w)
                           - y^TR)**2 + lambda/2 *
           np.linalg.norm(w)**2
```

To train the network, i.e. find the optimum  $\mathbf{w}$ , we need to minimize  $L(\mathbf{X}^{TR}, \mathbf{y}^{TR})$  with some optimization algorithm<sup>[3]</sup>. To minimize the loss function, we need its gradient:

$$\begin{aligned} \frac{\partial \tilde{L}(\mathbf{X}^{TR}, \mathbf{y}^{TR})}{\partial w_i} &= \sum_{j=1}^p 2 \cdot (y(\mathbf{x}_j^{TR}, \mathbf{w}) - \mathbf{y}_j^{TR}) \\ &\quad \cdot y(\mathbf{x}_j^{TR}, \mathbf{w}) \cdot (1 - y(\mathbf{x}_j^{TR}, \mathbf{w})) \\ &\quad \cdot \left( \frac{1}{1 + e^{-x_{ij}^{TR}}} \right) + \lambda \cdot w_i \end{aligned}$$

```
def g_loss(w, X^TR, y^TR, lambda=0):
    return np.squeeze(2 *
                      sigmoid(X^TR.T) @
                      ((y(X^TR, w) - y^TR) *
                      y(X^TR, w) * (1 -
                      y(X^TR, w))) + lambda * w.T)
```

**(a.1) Solving for target = [4]**

Now that we are ready – having the objective function  $L(\mathbf{X}^{TR}, \mathbf{y}^{TR})$ , the first-derivative optimization algorithms which we have implemented during this course<sup>[3]</sup> (GM, CGM and BFGS), and the data sets  $\mathbf{X}^{TR}, \mathbf{y}^{TR}$  and  $\mathbf{X}^{TE}, \mathbf{y}^{TE}$  – we can solve the pattern recognition problem for any set of digits. We are going to start with the target set [4]. The training data set for this problem:

- Has 500 observations ( $p = 500$ ).
- Train frequency = 0.5
- Noise frequency = 0.1

The hyperparameters for optimization defined for this problem are the following:

- $\lambda = 0.0$  (no regularization).
- $\varepsilon = 1.0e-06$
- $k^{max} = 1000$  (iterations).

The hyperparameters for *line search*:

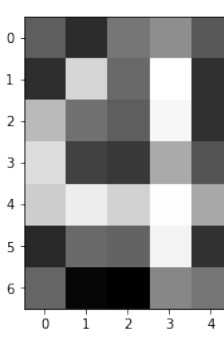
- $\alpha_0^{max} = 1$  initially. Later on, we update it by  $\alpha_k^{max} = \frac{2(f^k - f^{k-1})}{\nabla f^k \cdot d^k}$  [1].
- $c_1 = 0.01$
- $c_2 = 0.45$
- $\varepsilon = 1.0e-06$
- $k^{max} = 500$ .

First, we use the *Gradient Descent* (GM) algorithm to minimize the loss function in the neural network.

```
 $\mathbf{X}^{TR}, \mathbf{X}^{TE}, \mathbf{y}^{TR}, \mathbf{y}^{TE} =$ 
    gen_data(123456, 500,
             [4], 0.5, 0.1)
net = SLNN()
net.train("GM",  $\mathbf{X}^{TR}, \mathbf{y}^{TR}$ )
net.summary( $\mathbf{X}^{TE}, \mathbf{y}^{TE}$ )
```

The output (300 iterations):

	alpha	f(x)	g(x)	
0	NaN	125.000000	40.188919	0
1	1.000000	15.796008	24.173601	1
2	0.093438	0.008158	0.053878	2
3	10.622513	0.000228	0.000674	3
4	136.503566	0.000210	0.000291	4
295	1910.660877	0.000019	0.000010	5
296	2484.514941	0.000019	0.000012	6
297	1929.837670	0.000018	0.000010	
298	2532.521113	0.000018	0.000011	
299	1949.029427	0.000018	0.000010	

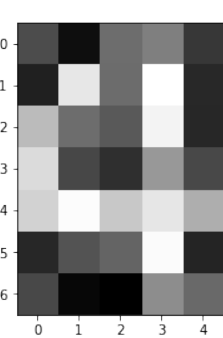


Loss: 1.8216667708018856e-05  
 Training accuracy: 100.0%  
 Test accuracy: 100.0%

Gradient:  
 [ 1.39508877e-06 6.86067063e-07 1.25624980e-06 9.27444777e-09  
 1.24933761e-06 1.02400665e-06 1.45817790e-06 2.27446974e-06  
 -2.02614002e-07 9.31133081e-07 2.83716521e-08 1.18237844e-06  
 2.22951532e-06 -1.55850183e-06 7.33653553e-07 3.12194833e-07  
 3.87212822e-06 1.61579652e-06 2.89661978e-08 1.05201981e-06  
 5.01824374e-07 1.10124825e-06 3.82445203e-07 -8.19933923e-07  
 5.95582419e-08 3.78435258e-06 1.08067482e-06 2.50751307e-06  
 -3.45722870e-07 1.07753526e-06 1.09897810e-06 3.78349079e-06  
 3.42813547e-06 1.57444321e-06 1.03685962e-06]

Solving with *Conjugate Gradient Descent* (CGM), Fletcher-Reeves variant without restart condition yields a result in 15 iterations, which is really fast in comparison with GM:

	alpha	f(x)	g(x)	
0	NaN	125.000000	40.456701	0
1	1.000000	24.391942	29.139734	1
2	0.058977	0.158842	0.904987	2
3	1.000000	0.000322	0.002213	3
4	64.000000	0.000128	0.000749	4
10	1024.000000	0.000063	0.000226	5
11	1024.000000	0.000029	0.000175	6
12	512.000000	0.000015	0.000071	
13	1024.000000	0.000010	0.000013	
14	1024.000000	0.000009	0.000006	

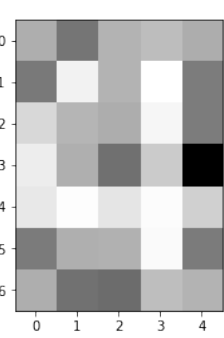


Loss: 9.405133460512638e-06  
 Training accuracy: 100.0%  
 Test accuracy: 100.0%

Gradient:  
 [-4.61148375e-07 -8.72512133e-07 -1.10942039e-06 -1.33940181e-06  
 6.76399783e-07 -7.98098872e-07 -1.21181858e-06 -2.15213263e-07  
 -1.17409631e-06 -8.18152637e-07 -1.56324633e-06 -6.55873629e-07  
 -4.95715050e-07 -1.77724591e-06 -6.55502062e-07 -1.16977558e-06  
 6.35607716e-07 -8.73882637e-08 -1.31237760e-06 -5.25775763e-07  
 -1.09471890e-06 -2.62685088e-06 -1.70196453e-06 -1.17884805e-06  
 -1.52770063e-06 4.33062445e-07 -5.19037715e-07 -2.37519294e-07  
 -1.72828267e-06 -3.33750722e-08 2.09167799e-07 1.14300604e-08  
 5.17718948e-07 -1.39389380e-06 -7.62106624e-07]

Finally, we solve it using *Broyden-Fletcher-Goldfarb-Shanno* (BFGS): we finish in just 8 iterations, but the accuracy is a bit smaller (<100%) than in the first two methods:

	alpha	f(x)	g(x)	
0	NaN	125.000000	40.456701	0
1	1.000000e+00	24.391942	29.139734	1
2	1.583714e-07	0.058668	0.335474	2
3	1.984351e-12	0.018684	0.109012	3
4	1.036548e-18	0.009785	0.018913	4
5	1.261522e-24	0.009468	0.007268	5
6	7.572507e-31	0.009458	0.006787	6
7	1.453318e-37	0.009458	0.006786	

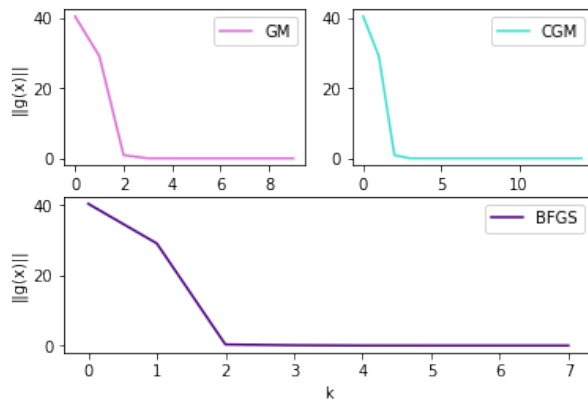


Loss: 0.009458452706506937  
 Training accuracy: 100.0%  
 Test accuracy: 99.96%

Gradient:  
 [ 1.90968642e-03 1.86146849e-03 2.36618691e-04 2.27888247e-04  
 2.23956476e-03 1.61965271e-04 -1.33871647e-03 2.09608661e-04  
 -1.84461578e-03 -3.16866392e-06 -1.50840200e-03 1.75381442e-04  
 1.86910383e-04 -1.50048964e-03 1.94071284e-04 -4.27414678e-04  
 5.45561543e-04 -5.78870631e-04 2.28663380e-04 -1.88968978e-03  
 -1.49841345e-03 4.06025417e-04 3.33071068e-04 8.74324829e-04  
 -1.50408204e-03 -2.60932272e-04 1.89829357e-03 2.15254052e-04  
 -1.62546809e-03 2.25169324e-04 2.21815169e-03 5.46615144e-04  
 -1.48662094e-04 -1.49958220e-03 1.62244701e-04]

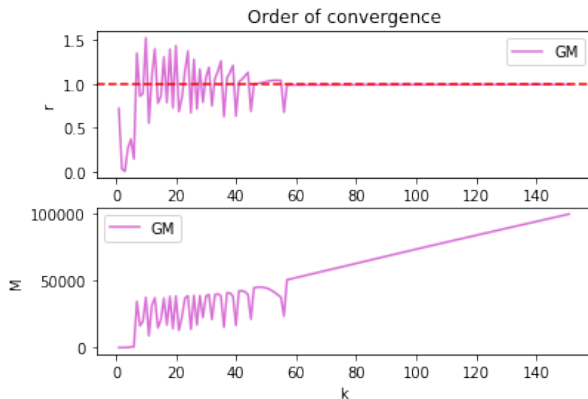
Note that in all the executions we have used initial weights equal to *zero*. We first tried to sample them from a *Normal distribution* with zero mean, but it gave much worse results.

Let's analyze the convergence of the three methods. All of them converge to a local optimum, since they achieved minimizing the loss function up to  $<0.0095$  and the gradient norm  $<0.007$  (*figure 3*).



**Figure 3:** The decrease of the gradient norm.

The fastest one to converge is *BFGS*, but it is the less accurate from the three methods since its test accuracy is 99.96%, whereas training accuracy is 100%. The other two yield 100% training and test accuracy, which is actually really good – the neural network has learnt well the pattern with *GM* and *CGM*.



**Figure 4:** We have observed linear convergence in the Gradient Descent method, with rate = 1.

### (a.2) Solving for target = [8]

Let's solve now for the target set [8]. The training data set for this problem:

- Has 500 observations ( $p = 500$ ).
- Train frequency = 0.5
- Noise frequency = 0.1

The hyperparameters for optimization defined for this problem are the following:

- $\lambda = 0.0$  (no regularization).
- $\varepsilon = 1.0e-06$
- $k^{max} = 1000$  (iterations).

The hyperparameters for *line search*:

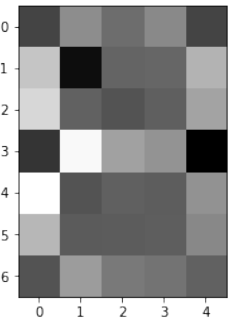
- $\alpha_0^{max} = 1$  initially. Later on, we update it by  $\alpha_k^{max} = \frac{2(f^k - f^{k-1})}{\nabla f^k \cdot d^k}$  [1].
- $c_1 = 0.01$
- $c_2 = 0.45$
- $\varepsilon = 1.0e-06$
- $k^{max} = 500$ .

Using GM, we reach  $k^{max} = 1000$ :

	alpha	f(x)	g(x)
0	NaN	125.000000	23.498640
1	0.012545	121.438869	69.109355
2	0.001506	118.254841	27.115333
3	0.003138	117.113696	35.119194
4	0.001869	115.983980	27.143027
996	0.016728	9.160516	0.748599
997	0.010197	9.157672	0.581202
998	0.017010	9.154829	0.753820
999	0.010105	9.151970	0.577749
1000	0.017300	9.149113	0.759205

Loss: 9.149113476270594  
 Training accuracy: 98.6%  
 Test accuracy: 94.86%

Gradient:  
 [ 0.00763287 -0.09295857 -0.12139392 -0.20208507 -0.05734815 -0.24112008  
 0.05372104 -0.02434171 -0.03876911 -0.21666789 -0.24852801 -0.02279227  
 -0.0017461 -0.02426949 -0.16204628 -0.03906595 -0.25734013 -0.14489296  
 -0.15743184 0.03364612 -0.25564692 -0.02506217 -0.09436931 -0.01998683  
 -0.13349138 -0.16784354 -0.0295514 -0.01932105 -0.02406408 -0.10942633  
 -0.01048636 -0.18372634 -0.13236543 -0.08516605 -0.03530699]

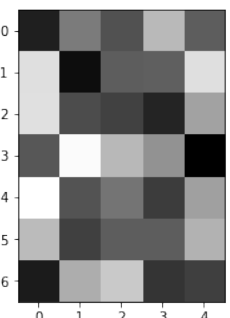


In contrast, *CGM Fletcher-Reeves* variant without restart reaches a local optimum in 328 iterations:

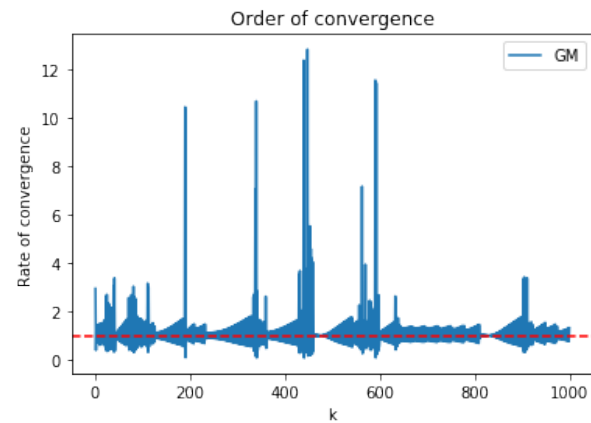
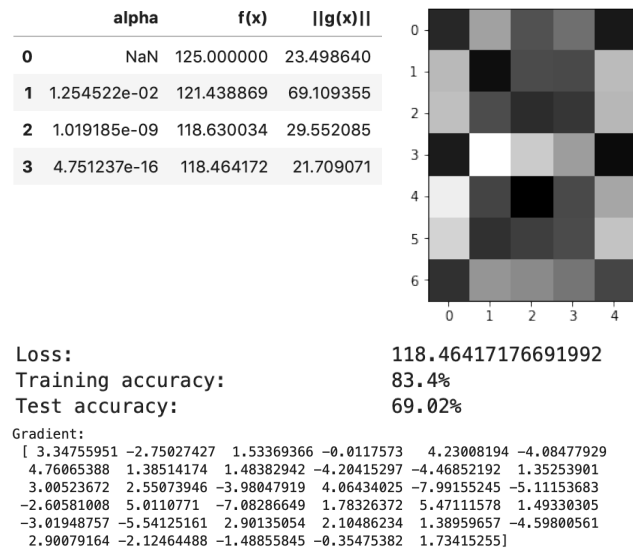
	alpha	f(x)	g(x)
0	NaN	125.000000	23.498640
1	0.012545	121.438869	69.109355
2	0.001506	118.254841	27.115333
3	0.010094	113.755561	44.911436
4	0.003360	107.232204	32.831670
323	1024.000000	0.000069	0.000024
324	1024.000000	0.000069	0.000027
325	1024.000000	0.000067	0.000013
326	1024.000000	0.000067	0.000018
327	1024.000000	0.000066	0.000010

Loss: 6.640498856902224e-05  
 Training accuracy: 100.0%  
 Test accuracy: 94.38%

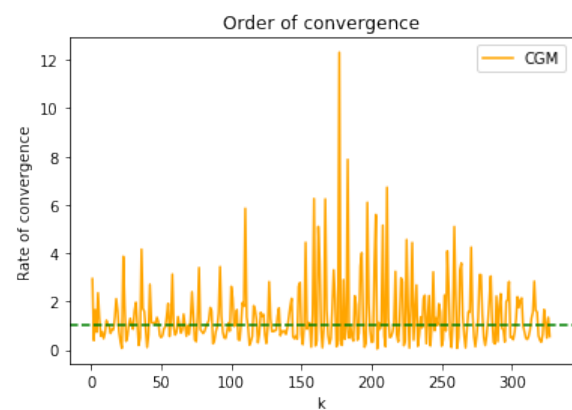
Gradient:  
 [ 1.57037960e-07 -5.15993848e-07 -2.51979707e-07 -4.09493612e-07  
 -8.46613117e-08 -2.99155700e-06 4.78762156e-06 8.59580534e-08  
 -9.94293613e-07 -2.93405157e-06 -1.59156194e-06 7.37128710e-07  
 -2.61429570e-06 -9.84305279e-07 -3.17626172e-06 2.82194835e-07  
 -4.46126255e-07 -7.75183310e-07 -1.68271266e-07 1.42375782e-07  
 3.11586100e-06 1.25755726e-07 -1.23222807e-08 -1.97652704e-06  
 -5.69778867e-07 8.00403291e-07 6.90101224e-07 -9.58539365e-07  
 2.45260755e-06 -1.32194045e-06 9.76953817e-07 -1.17142817e-06  
 -9.05194687e-07 -5.91698141e-08 -2.68394017e-06]



Output by *BFGS* is quite poor, it finishes its execution in 4 iterations without a decent loss minimization (we could improve that if we had added the regularization term):



**Figure 5:** *GM rate oscillates around  $r = 1$ .*



**Figure 6:** *CGM rate of convergence with target set = [8].*

Again, the fastest method to finish is *BFGS*, but it does not find an optimum. *GM* does not find the minimum neither (it exceeds  $k^{max}$ ). Thus, both of them yielded a <95% test accuracy, in the case of *BFGS* <70% test accuracy. However, *CGM* does find a local minimum and outputs 100% train accuracy and quite a good test accuracy, 94.38%.

## (b) BFGS for all digits one by one

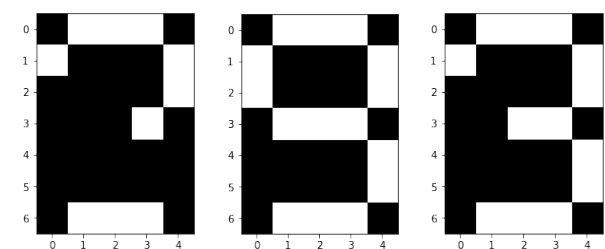
We use the *SLNN* to identify each single number from 0 to 9. We use a training set of size 500, and the results can be reproduced with the 123456 seed. After training, we get the following results:

	ITER.	TRAIN ACC.	TEST ACC.
0	3	94.6%	89.48%
1	5	99.6%	99.68%
2	3	97.8%	94.94%
3	3	94.6%	90.44%
4	5	100.0%	99.72%
5	3	98.4%	97.66%
6	4	96.4%	94.14%
7	5	98.0%	96.04%
8	3	87.2%	74.08%
9	3	93.2%	85.46%

We can see how there are some numbers that have a much lower accuracy than others. This can be explained easily: the *SLNN* only takes into account the position of the pixels, giving each of them higher weights if they are more important for identifying our target number. But here comes the problem: other numbers also share those same pixels!

In the following matrix you can see the number of shared non-empty pixels of each pair of numbers:

	0	1	2	3	4	5	6	7	8	9
0	16	4	9	12	7	12	11	5	14	13
1	4	10	5	5	4	5	6	2	5	5
2	9	5	14	10	5	7	6	9	10	10
3	12	5	10	14	5	10	10	6	14	13
4	7	4	5	5	15	6	9	4	7	6
5	12	5	7	10	6	17	10	5	11	11
6	11	6	6	10	9	10	16	4	13	11
7	5	2	9	6	4	5	4	11	6	6
8	14	5	10	14	7	11	13	6	17	15
9	13	5	10	13	6	11	11	6	15	16



**Figure 7:** *Shared pixels of (8, 2), (8, 9) and (3, 9).*

We can see in the matrix a group of numbers that share a high number of pixels between

them: 0, 3, 8 and 9. These are also the numbers with the lowest test accuracy!

If we sum the columns of the previous matrix (excluding the diagonal) we will get the sum of shared pixels with all other numbers:

[87, 41, 71, 85, 53, 77, 80, 47, 95, 90]

Sorting the numbers according to this, we get:

8, 9, 0, 3, 6, 5, 2, 4, 7, 1

While sorting according to the test accuracy (from lower to higher) we get:

8, 9, 0, 3, 6, 2, 7, 5, 1, 4

The first 5 digits with the most shared pixels are also the ones with the lowest test accuracy. There seems to be an obvious relationship between the shared pixels and the accuracy of our one-digit models.

### Final discussion and conclusions

We have achieved our objective, that is – to build a *Single Layer Neural Network* with our optimization algorithms to solve the pattern recognition problem (in *Python*). During this project we had some troubles such as finding a *line search* algorithm that worked correctly, as the one provided in *Python* worked strangely and did not converge to an optimum.

Our first attempt was to look for other open source implementations, such as `optinpy`, but they suffered from the same problems. After many failed attempts to find a working line search, we decided to translate the one provided by the instructor in *MATLAB* into *Python*. It worked much better, but when assigning an  $\alpha_k^{max}$  using any of the formulas provided to us it failed to converge to an optimal solution.

This was eventually solved after checking the documentation and source code of the line search in the `scipy.optimize` library. This one implemented a variant of the  $\alpha_k^{max}$  update formula from *N&W<sup>[1]</sup>* and the results were much better.

We have tried to use the basic *Backtracking Line Search (BLS)* algorithm we implemented in this course. It gave rather poor results and the time execution was relatively slow.

During this project we have also seen with our own eyes the truth behind a much-repeated sentence during this course: “*most of the machine learning problems are, in the end, optimization problems*”.

In fact, a good understanding of numerical optimization seems essential when working in the field of *machine learning*. With the same model architecture, we have been able to reduce the number of iterations from thousands (if we just blindly use *Gradient Descent*) to the single digits. And implementing this simple neural network and the optimization routines ourselves has given us the confidence that not only we can use these tools but understand its strengths and be able to solve any difficulties that we might face regarding the convergence of the training.

### References

- [1] J. SPRINGER & S. J. WRIGHT, *Numerical Optimization*, Springer, 1999.
- [2] C. M. BISHOP, *Pattern recognition and Machine learning*, Springer, 2006.
- [3] F. J. HEREDIA & J. CASTRO, *Mathematical Optimization lecture notes*.
- [4] L. A. BELANCHE, *Machine Learning lecture notes*.

N.B. All the figures used in this project – own source.