PATTERN RECOGNITION

WITH SINGLE LAYER NEURAL NETWORK

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*x*5 pixel matrix representation, with values 0 and 1 (*figure 1*).

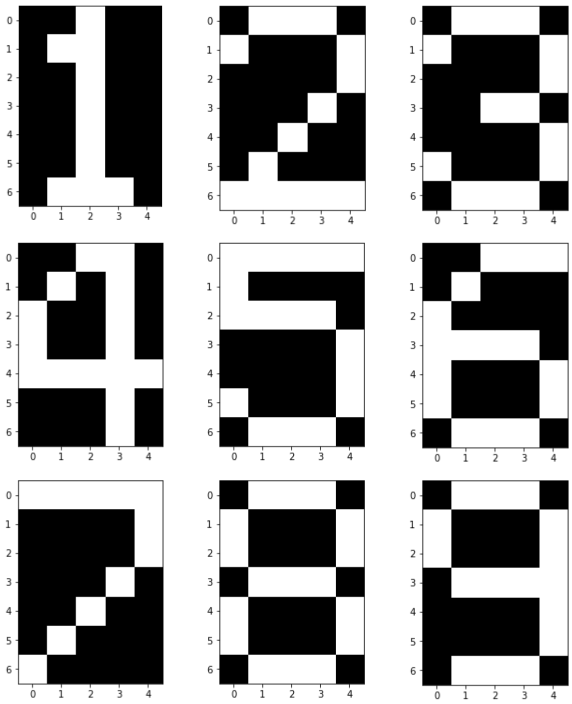


Figure 1: *Pixel matrix representation of [0,9] numbers.*

Network architecture

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

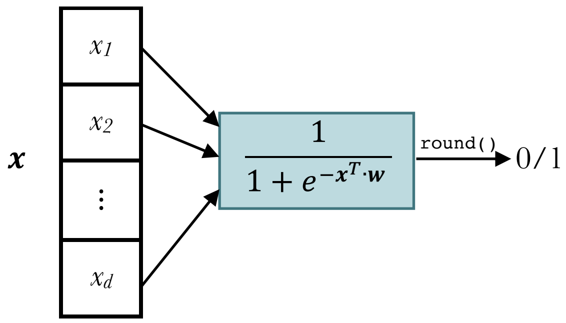


Figure 2: *Architecture of a perceptron (SLNN) with a sigmoidal activation function.*

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

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

We define 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*". 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* $(\boldsymbol{X}^{TR}, \boldsymbol{y}^{TR})$ and *test* $(\boldsymbol{X}^{TE}, \boldsymbol{y}^{TE})$ sets, respectively.

The loss function is defined as follows:

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

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

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

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

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

To train the network, i.e. find the optimum $\boldsymbol{w}$, we need to minimize $L(\boldsymbol{X}^{TR}, \boldsymbol{y}^{TR})$ with some optimization algorithm. To minimize the loss function, we need its gradient:

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

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

Solving for target = [4]

Now that we are ready – we have the objective function $L(\boldsymbol{X}^{TR}, \boldsymbol{y}^{TR})$, the first-derivative optimization algorithms which we have implemented during this course (GM, CGM and BFGS), the data sets $\boldsymbol{X}^{TR}, \boldsymbol{y}^{TR}$ and $\boldsymbol{X}^{TE}, \boldsymbol{y}^{TE}$ – we can solve the pattern recognition problem for any set of numbers. 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:

- $\lambda = 0.0$ (*L2* regularization).
- $\varepsilon = 1.0\text{e}{-}0.6$
- $k^{max} = 1000$ (iterations).

The hyperparameters for *line search*:

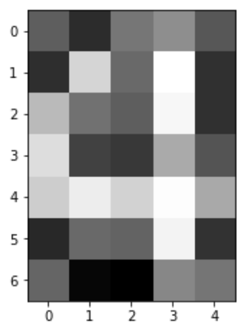
- $\alpha_0^{max} = 1$ initially. Later on, we update it by $\alpha_k^{max} = \frac{2(f^k - f^{k-1})}{\nabla f^{k^T} \cdot d^k}$.
- $c_1 = 0.01$
- $c_2 = 0.45$
- $\varepsilon = 1.0\text{e}{-}0.6$
- $k^{max} = 500$.

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

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

The output (300 iterations):

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

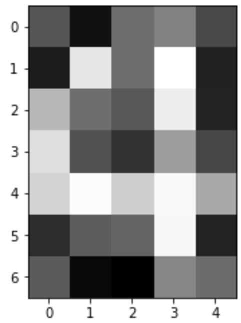



```
Loss:                          1.8216667708018856e-05
Training accuracy:             100.0%
Test accuracy:                 100.0%
```

```
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 just 4 iterations, which is really fast:

	alpha	f(x)	||g(x)||
0	NaN	1.250000e+02	40.456701
1	3.1250	2.895490e+01	9.429834
2	1.5625	9.704749e-01	3.570048
3	3.1250	4.110602e-11	0.000000

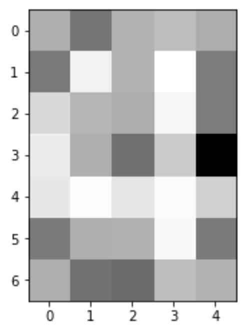



```
Loss:                          4.110601874626356e-11
Training accuracy:             100.0%
Test accuracy:                 100.0%
```

```
Gradient:
 [4.48663204e-11 4.48608548e-11 6.01015057e-11 6.01014082e-11
 6.00959844e-11 4.11058070e-11 5.63409937e-11 4.11452196e-11
 4.11478962e-11 4.11112718e-11 5.63408964e-11 4.11058293e-11
 4.11059052e-11 4.11058086e-11 4.11112716e-11 5.63464458e-11
 5.63408968e-11 5.63409914e-11 6.01014082e-11 4.11085815e-11
 5.63409323e-11 4.48608664e-11 6.00593936e-11 6.00931965e-11
 5.63409460e-11 5.63408964e-11 4.48663188e-11 4.11452195e-11
 4.11112725e-11 4.11479542e-11 6.00592961e-11 5.63381635e-11
 5.63409822e-11 5.63409939e-11 4.11058071e-11]
```

Eventually, we solve it by *Broyden-Fletcher-Goldfarb-Shanno* (BFGS), we finish in 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
1	1.000000e+00	24.391942	29.139734
2	1.583714e-07	0.058668	0.335474
3	1.984351e-12	0.018684	0.109012
4	1.036548e-18	0.009785	0.018913
5	1.261522e-24	0.009468	0.007268
6	7.572507e-31	0.009458	0.006787
7	1.453318e-37	0.009458	0.006786

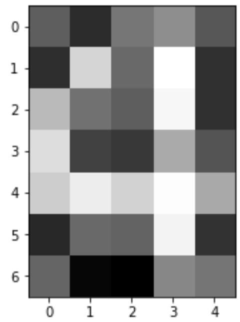



```
Loss:                          0.009458452706506937
Training accuracy:             100.0%
Test accuracy:                 99.96%
```

```
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]
```

Solving for target = [8]

Now that we are ready – we have the objective


0
1
2
3
4
5
6
0
1
2
3
4