Electrical Engineering and Computer Science University of Patras
DATA PROCESSING AND MACHINE LEARNING ALGORITHMS

ΕΠΕΞΕΡΓΑΣΙΑ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΑΛΓΟΡΙΘΜΟΙ ΜΑΘΗΣΗΣ

Academic Year 2020-2021

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ
UNIVERSITY OF PATRAS

Restore Images Using Gans

Fotios - Panagiotis	Basamakis

Problem 1:

Firstly from the file “data21.mat” we loaded the pre trained $G(Z)$ a generator of a GAN that produces hand written eights and produces a vector Z[1x10] with elements independent random variables that follow a normal distribution with mean = 0 and std = 1

$N(0,1)$

```python
import numpy as np
import scipy.io
import matplotlib.pyplot as plt

Z = np.random.normal(0, 1, (100, 10))
mat = scipy.io.loadmat('data21.mat')
A1 = mat['A_1']
A2 = mat['A_2']
B1 = mat['B_1']
B2 = mat['B_2']
```

The generative model is $G(Z), Z$ and that is used for the production of 100 eights with the below procedure :

$$\begin{aligned} W_1 &= A_1 * Z + B_1 \\ Z_1 &= \max\{W_1, 0\} \text{ (Relu)} \\ W_2 &= A_2 * Z_1 + B_2 \\ X &= 1./(1 + \exp(W_2)) \text{ (Sigmoid).} \end{aligned}$$

In reality the above is the forward run of the Neural Network of the generator with input the vector Z. The size of the matrices are:

A1 [128 , 10] , B1 [128 , 1]

A2 [784 , 128] , B1 [784 , 1]

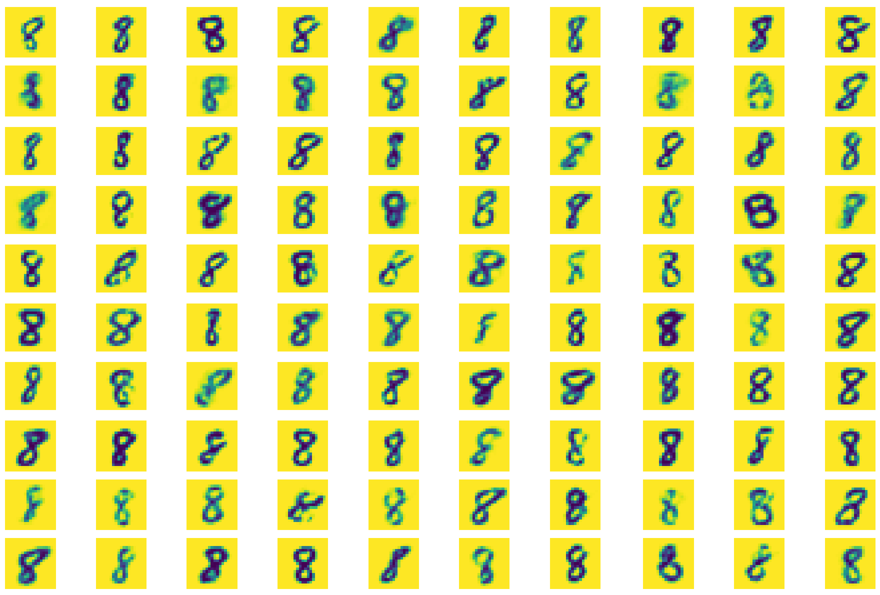
Relu and Sigmoid functions as well as the forward NN in Python:

```python
def relu(x):
    temp = np.where(x > 0, x, 0)
    return temp


def sigmoid(x):
    return 1 / (1 + np.exp(x))

def printEights(Z, A1, A2, B1,B2):
    W1 = np.dot(A1, Z[:].T) + B1
    Z1 = relu(W1)
    W2 = np.dot(A2, Z1[:]) + B2
    X = sigmoid(W2)
    return X.T
```

Code for printing 100 eights in an array (10 x 10):

```
X = printEights(Z, A1, A2, B1, B2)
for i in range(1, len(Z) + 1):
    X_2D = np.reshape(X[i - 1], (28, 28))
    plt.subplot(10, 10, i)
    plt.axis('off')
    plt.imshow(X_2D.T)

plt.show()
```

Results:



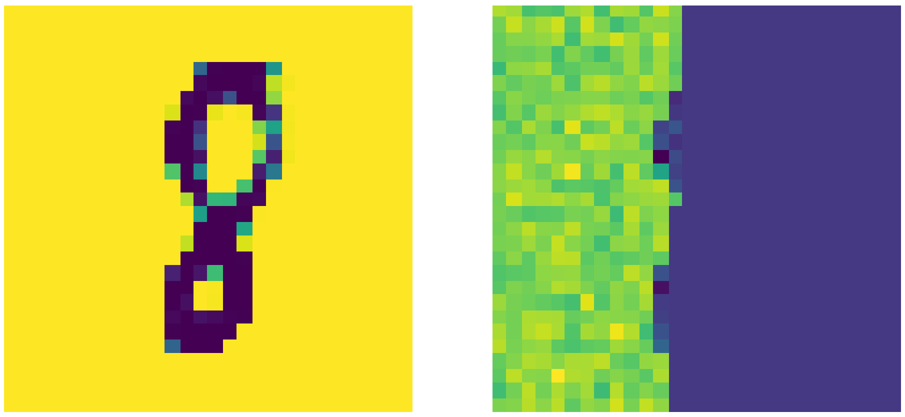
Problem 2:

From the file “data22.mat” we loaded the 4 ideal images [Xi] and the 4 noisy images [Xn].

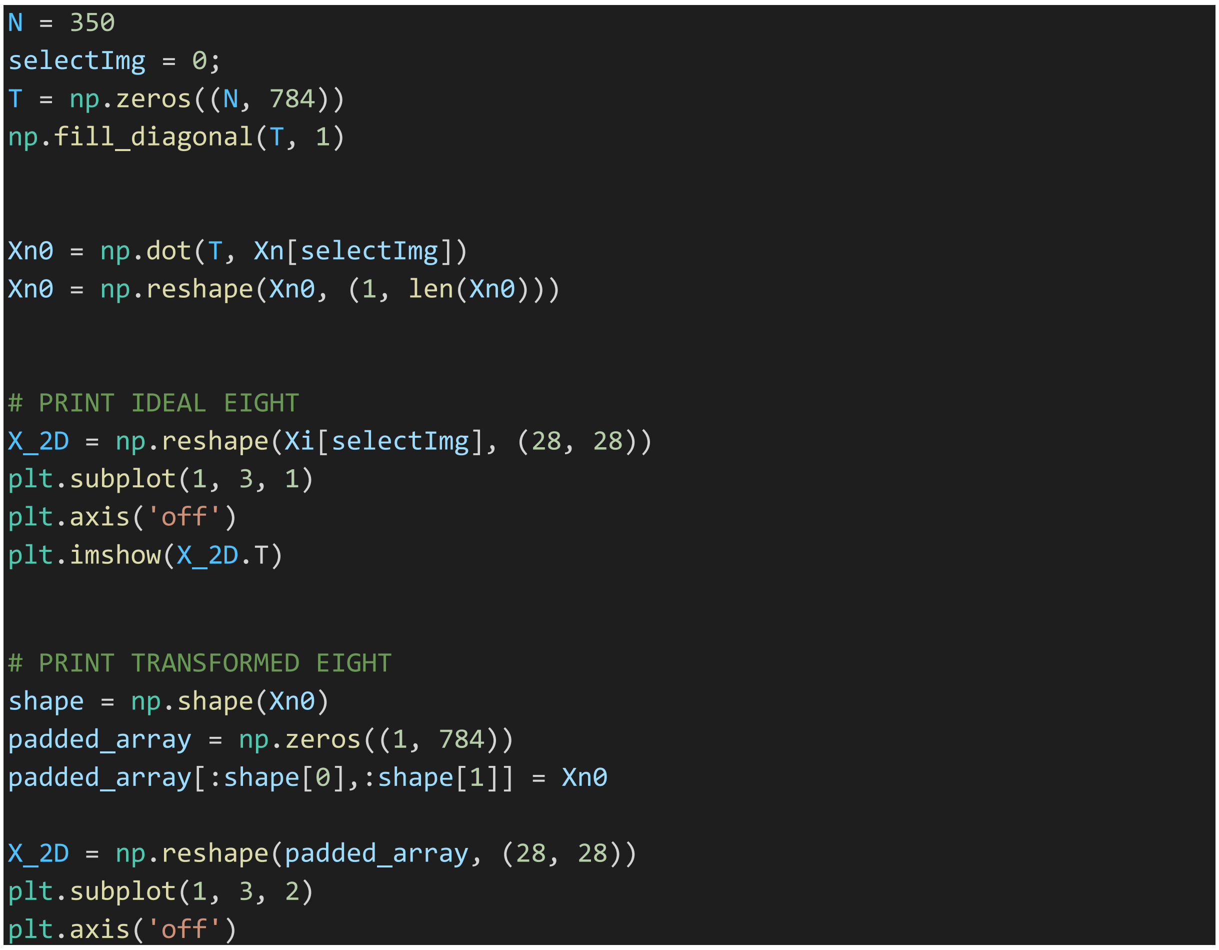
```
mat = scipy.io.loadmat('data22.mat')

Xi = mat['X_i']
Xn = mat['X_n']
Xi = Xi.T
Xn = Xn.T
```

Following up we determined the linear transformation T in which we submit the images of the vector Xn (noisy ones). This linear transformation is a matrix of size (N , 784) that has 1 in its main diagonal and we multiply it with the vector Xn (size (1 , 784)) .This gives as a result a matrix that show its first N elements and from N to 784 is black (zeros)

As shown below:



This is for N = 350 (for N = 500 we would have more information) and the code for the above:



```
N = 350
selectImg = 0;
T = np.zeros((N, 784))
np.fill_diagonal(T, 1)


Xn0 = np.dot(T, Xn[selectImg])
Xn0 = np.reshape(Xn0, (1, len(Xn0)))


# PRINT IDEAL EIGHT
X_2D = np.reshape(Xi[selectImg], (28, 28))
plt.subplot(1, 3, 1)
plt.axis('off')
plt.imshow(X_2D.T)


# PRINT TRANSFORMED EIGHT
shape = np.shape(Xn0)
padded_array = np.zeros((1, 784))
padded_array[:shape[0],:shape[1]] = Xn0

X_2D = np.reshape(padded_array, (28, 28))
plt.subplot(1, 3, 2)
plt.axis('off')
```

```python
plt.imshow(X_2D.T)
```

Afterwards we calculated the cost function as shown below:

$$J(Z) = Nlog\left(||TX - Xn||^2\right) + ||Z||^2$$

```python
def printEights(Z, A1, A2, B1, B2):
    W1 = np.dot(A1, Z.T) + B1
    Z1 = relu(W1)
    W2 = np.dot(A2, Z1) + B2

    X = sigmoid(W2)

    forwardParams = {'X': X.T, 'W2': W2, 'W1': W1}
    return forwardParams


def costFunction(params, T, Z, Xn):
    X = params['X']
    temp1 = np.dot(X, T.T) - Xn
    temp2 = np.sum(np.power(temp1, 2))
    return len(Xn.T) * np.log(temp2) + np.sum(np.power(Z, 2))
```

(The function printEights returns the output X of the generator as long as the matrices W1 , W2 that we need for the calculation of the grads.)

We want to minimize the above cost function in respect to the vector Z. That means that if we give for input in the generator the calculated min Z then the generator will calculate a sufficient estimation of the Xi (ideal) .

The cost function calculates the log of the square of the difference of the transformed image and every image that the generator produces. Because we have linear transformation we calculate the product TX and not the function T(X). The$||Z||^2$ is the power of the input.

For the minimization of the cost function we used the gradient descent algorithm:

$$Z_{next} = Z_{previous} - LearningRate\nabla_z J(Z)$$

And for calculating the gradients we followed the below procedure:

$\Phi(X) = \nabla_z\left[log\left(||TX - Xn||^2\right)\right]$

$u_2 = \nabla x\Phi(X) = \frac{2(TX-X_n)T}{||TX-X_n||^2}$

$(TX - X_n)$ is size of (1 x N) and T (N x 784) the rest are numbers that apply to every element of the final vector.

Obivously the final vector u2 has size (1 x 784).

$v2 = u2 \circ f_2'(W_2)$ where $f_2'(x)$ is the sigmoid derivative $f_2'(x) = -\frac{e^x}{(1+e^x)^2}$

The above product is element wise.

$u_1 = A_2^T * v2$ *(matrix product)*

$v1 = u1 \circ f_2'(W_1)$ where $f_1'(x)$ is the ReLu derivat $f_1'(x) = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases}$

$u_0 = A_1^T * v1 = \nabla_z \left[log\left(||TX - Xn||^2 \right) \right]$ (u_0 size (1 x 10))

The final gradient of the cost function is:

$$\nabla_z J(Z) = N * u_0 + 2 * Z$$

Python code:

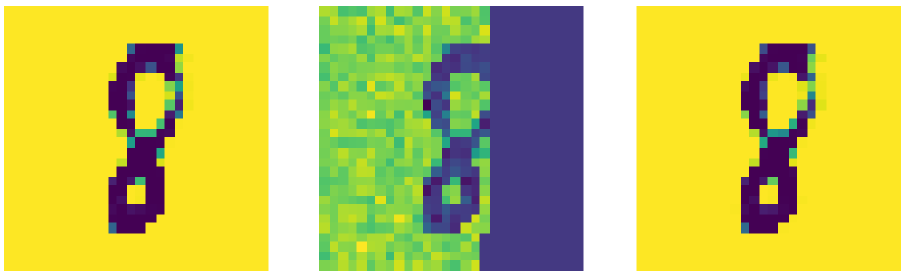
```python
def relu_derivative(x):
    temp = np.where(x > 0, 1, 0)
    temp = np.reshape(temp, (len(temp), 1))
    return temp


def sigmoid_derivative(x):
    return -np.exp(x) / np.power((1 + np.exp(x)), 2)

def costFunction(params, T, Z, Xn):
    X = params['X']
    temp1 = np.dot(X, T.T) - Xn
    temp2 = np.sum(np.power(temp1, 2))
    return len(Xn.T) * np.log(temp2) + np.sum(np.power(Z, 2))

def gradientCalc(A1, A2, params, T, Z, Xn):
    X = params['X']
    X = np.dot(X, T.T)
    W2 = params['W2']
    W2 = W2.T
    W2 = np.dot(W2, T.T)
    A2temp = A2.T
    A2temp = np.dot(A2temp, T.T)
    W1 = params['W1']
    # CALCULATION OF U2 and V2
    temp1 = X - Xn
    temp2 = np.sum(np.power(temp1, 2))
    u2 = 2 * (X - Xn) / temp2
    v2 = u2 * sigmoid_derivative(W2)
    # CALCULATION OF U1 and V1
    u1 = np.dot(A2temp, v2.T)
    v1 = u1 * relu_derivative(W1)

    #return the derivative

    u0 = np.dot(A1.T, v1)
    return N * u0 + 2 * Z.T
```

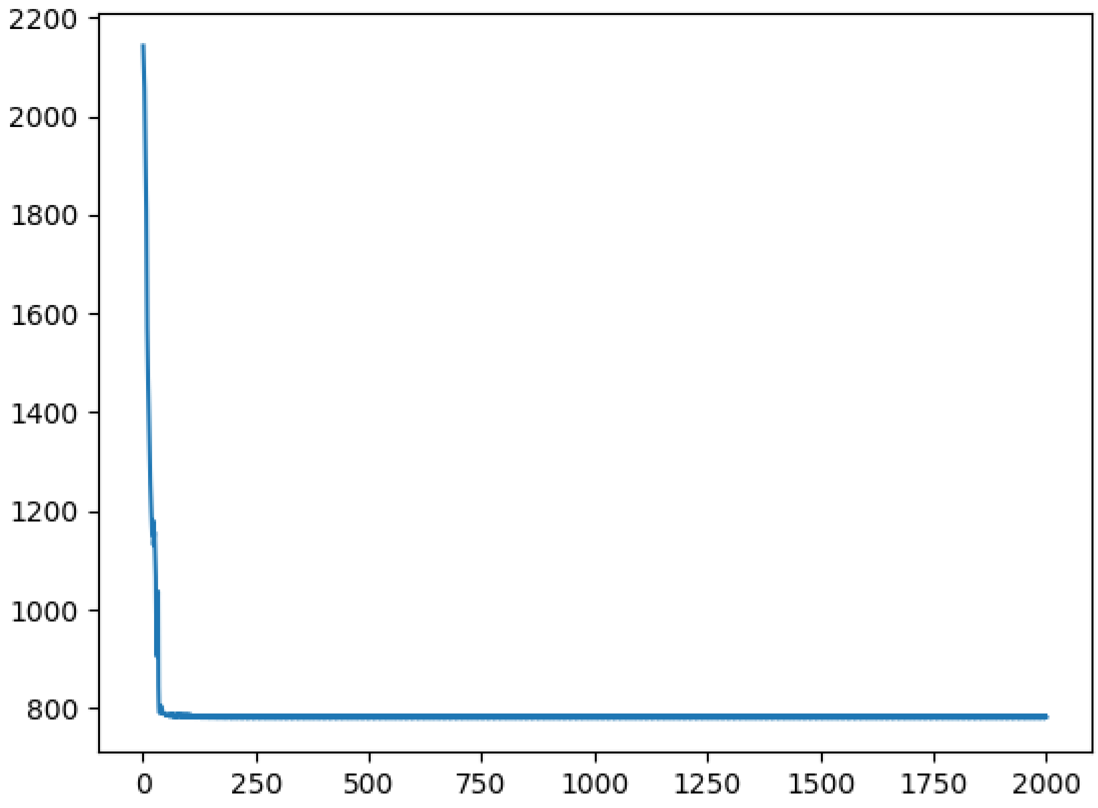
Every function work in a loop until we have convergence of the cost function:

```
learning_rate = 0.001
iterations = 2000
COST =[]

for i in range(iterations):
    params = printEights(Z, A1, A2, B1, B2)
    x = params['X']
    COST.append(costFunction(params, T, Z, Xn0))
    grad = gradientCalc(A1, A2, params, T, Z, Xn0)
    grad = grad.T
    Z = Z - learning_rate * grad



# PRINT MINE EIGHT
params = printEights(Z, A1, A2, B1, B2)
X_2D = np.reshape(params['X'], (28, 28))
plt.subplot(1, 3, 3)
plt.axis('off')
plt.imshow(X_2D.T)

plt.show()

x = np.linspace(1, len(COST), len(COST))
plt.plot(x, COST)
plt.show()
```

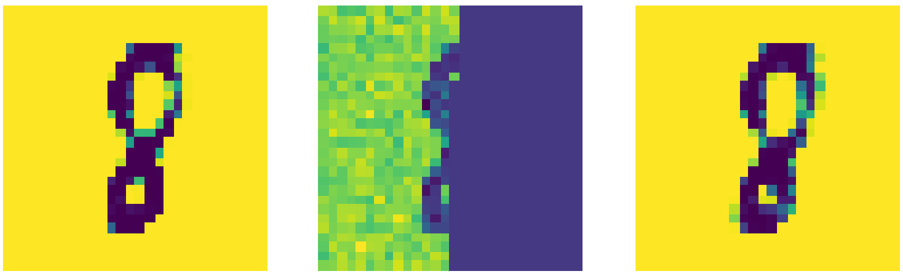
<u>Results for the first Image:</u>

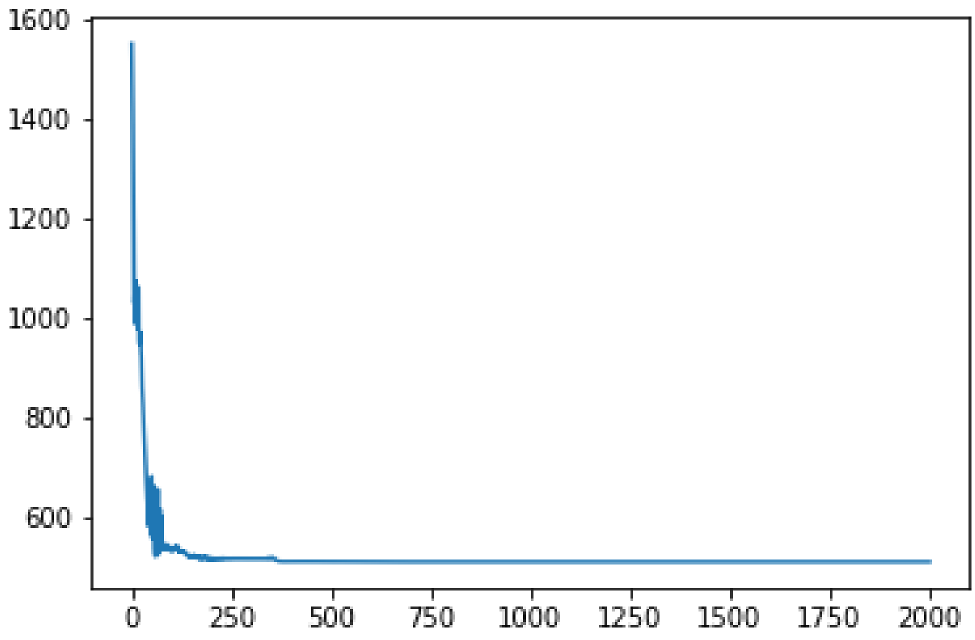
N = 500 , learning rate = 0.001 , iterations = 2000



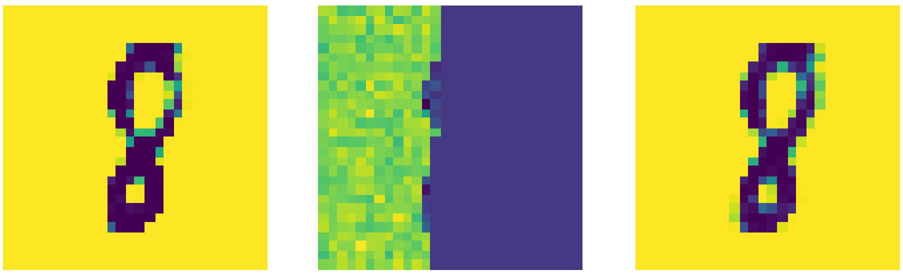


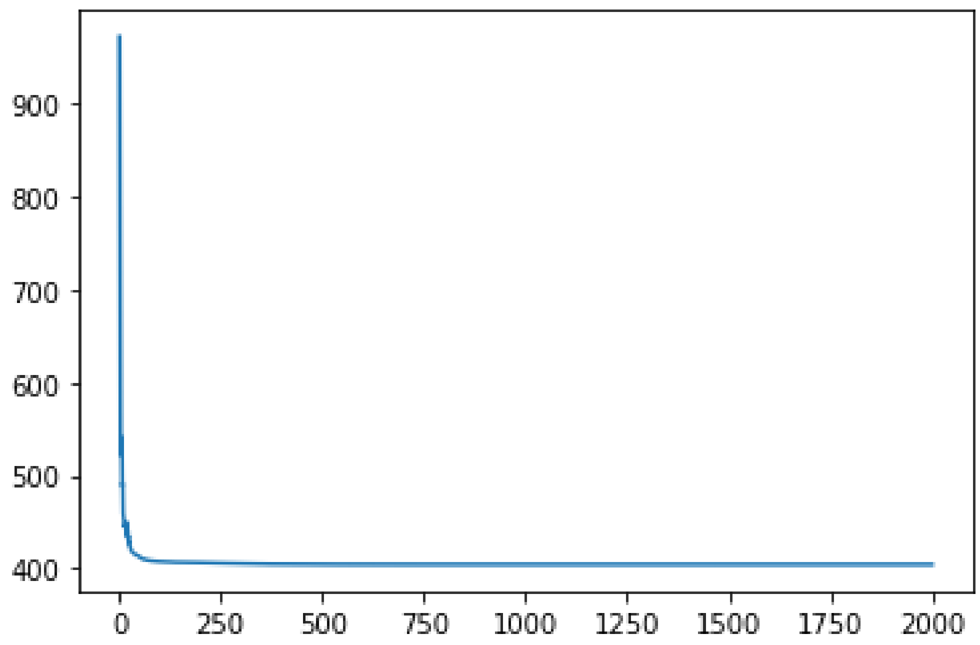
N = 400 , learning rate = 0.001 , iterations = 2000





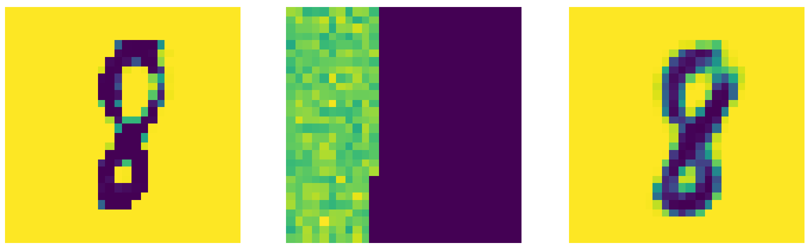
N = 350 , learning rate = 0.001 , iterations = 2000

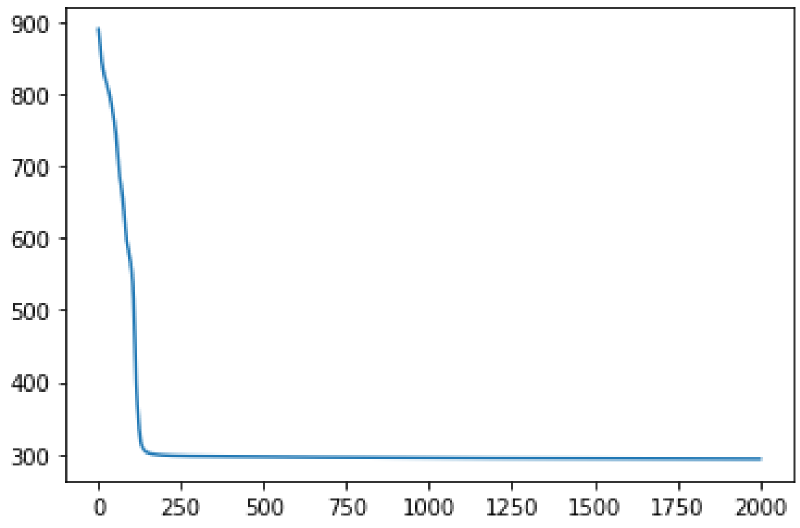




a

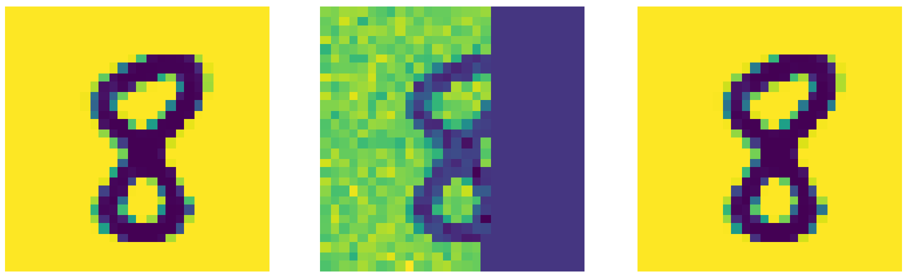
N = 300 , learning rate = 0.0001 , iterations = 2000

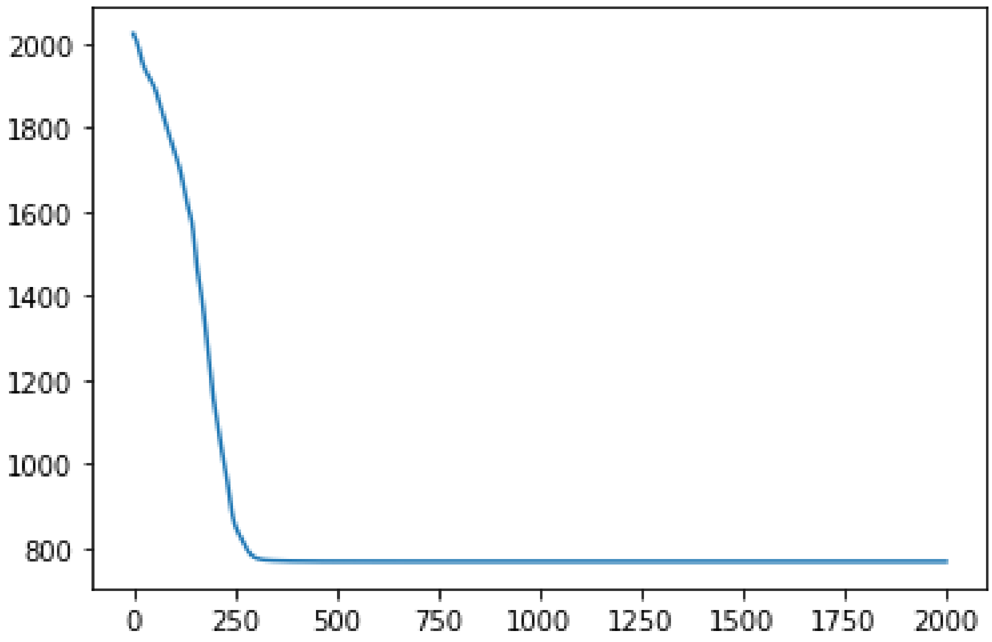




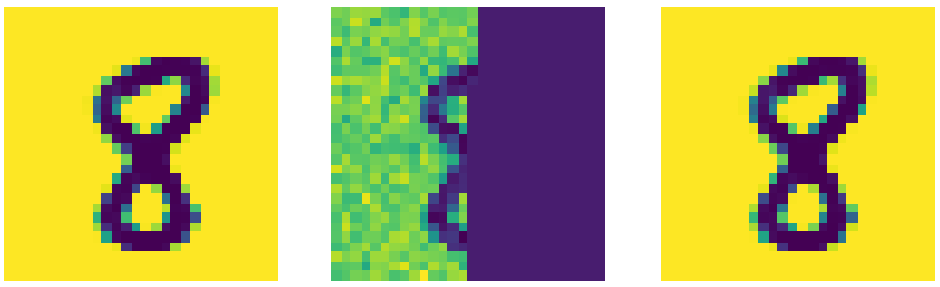
<u>Results for the second image:</u>

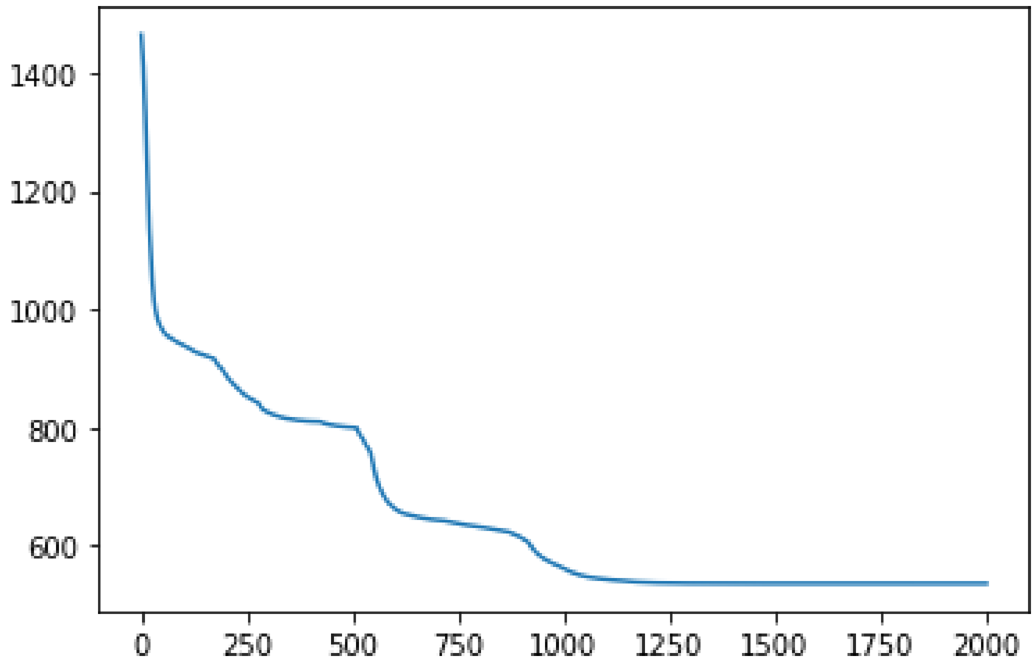
N = 500 , learning rate = 0.0001 , iterations = 2000



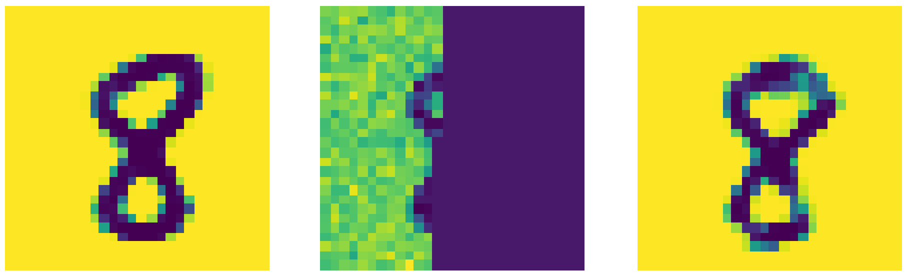


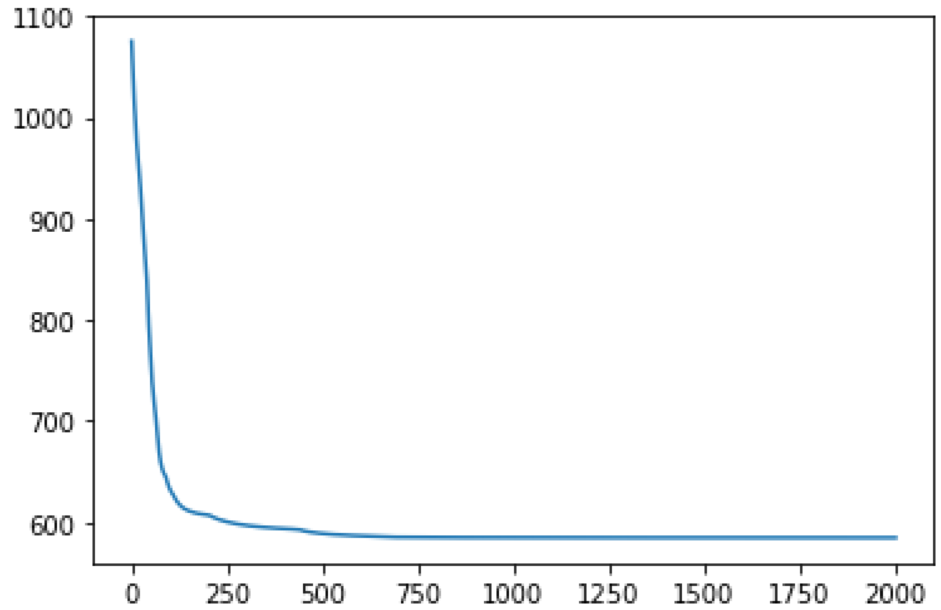
N = 400 , learning rate = 0.0001 , iterations = 2000





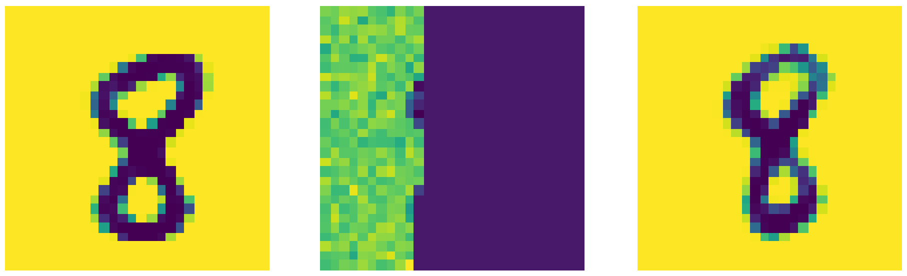
N = 350 , learning rate = 0.0001 , iterations = 2000

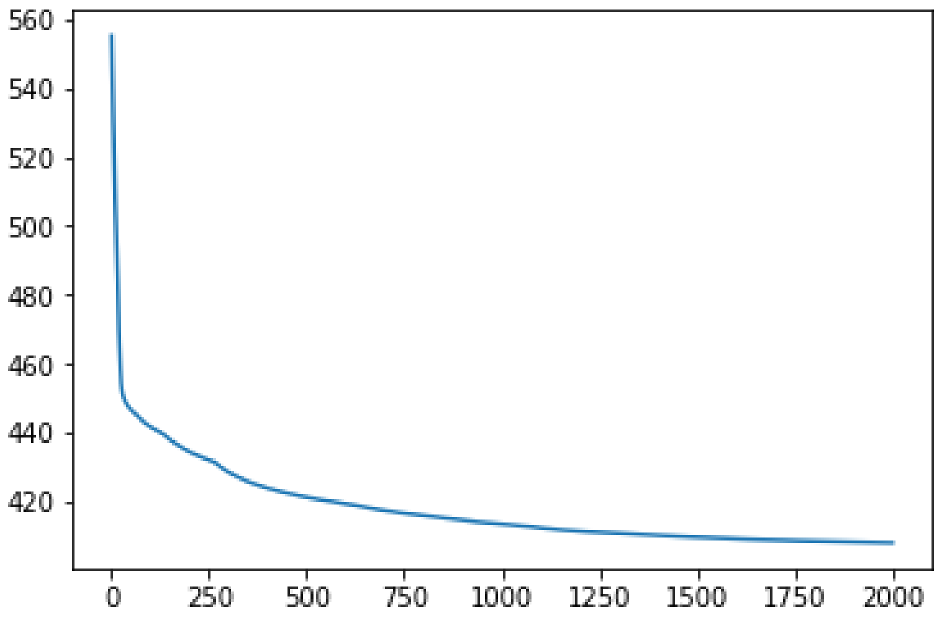




α

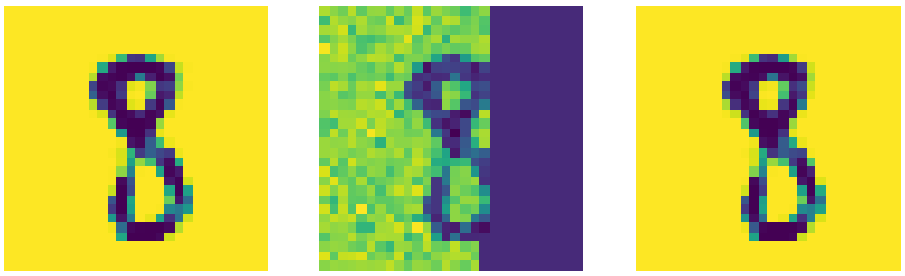
N = 300 , learning rate = 0.0001 , iterations = 2000

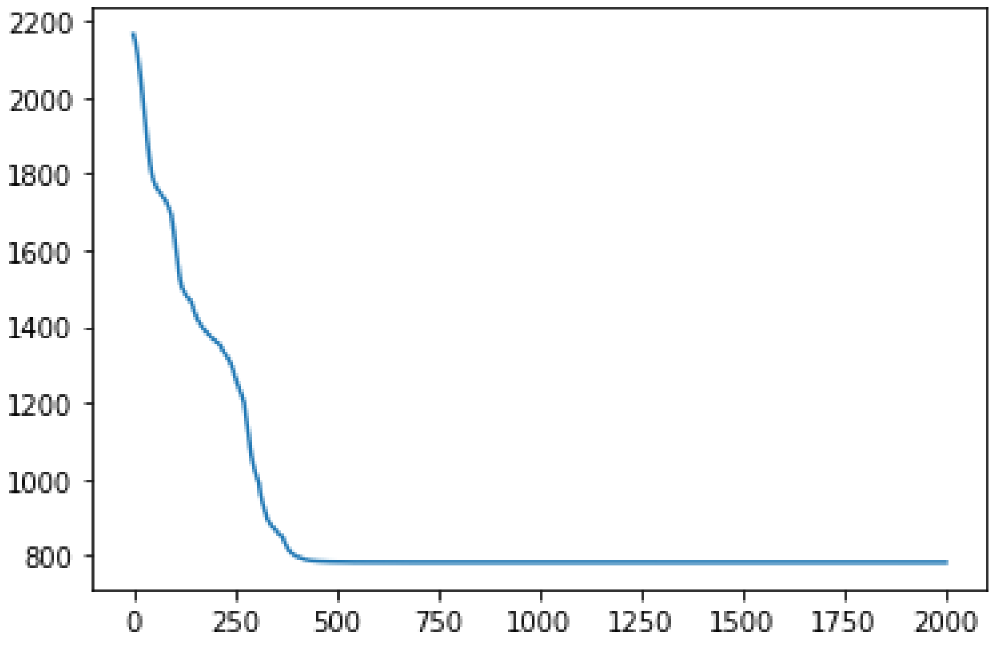




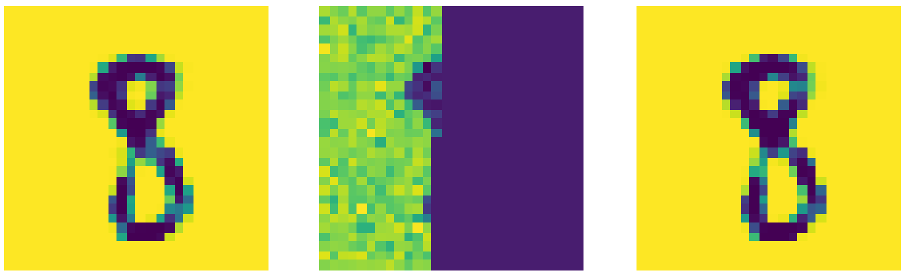
Results for the first image:

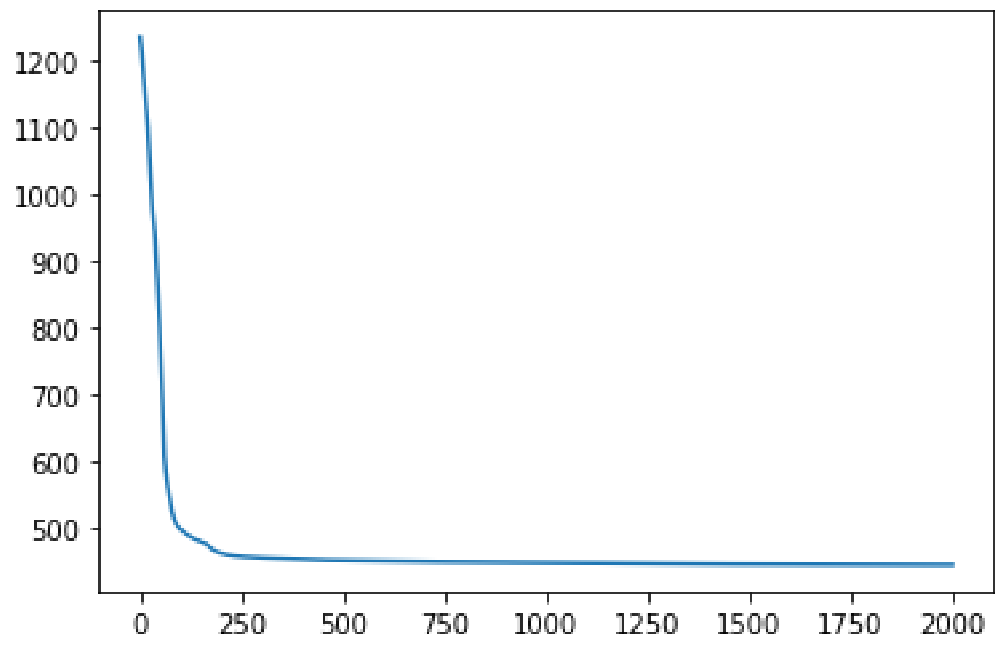
N = 500 , learning rate = 0.0001 , iterations = 2000





N = 350 , learning rate = 0.0001 , iterations = 2000

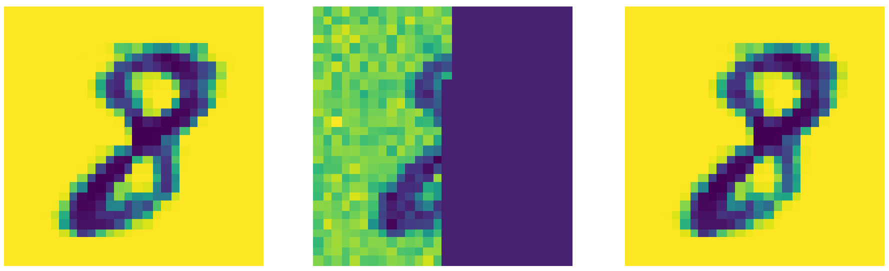


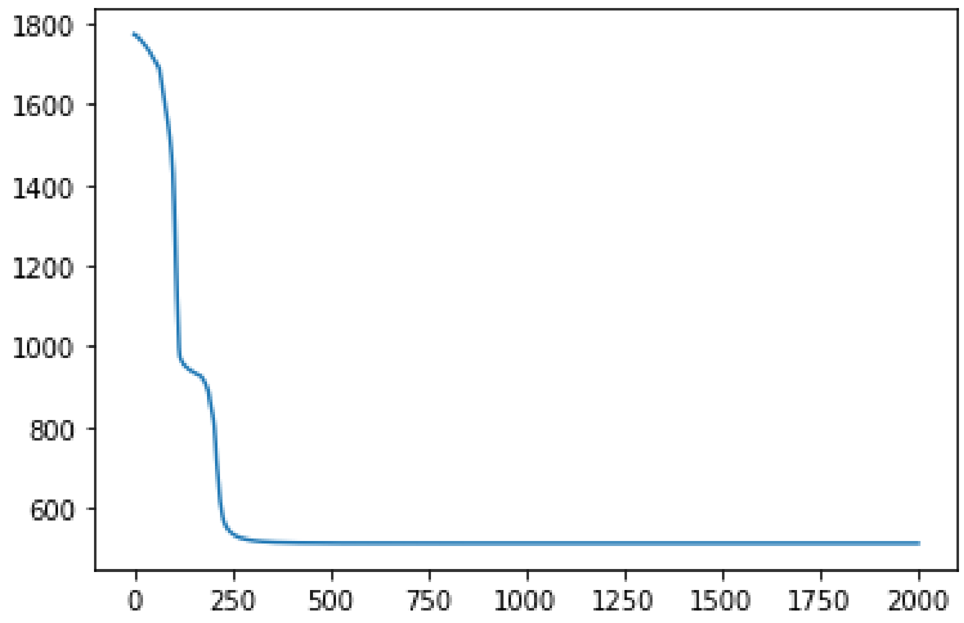


α

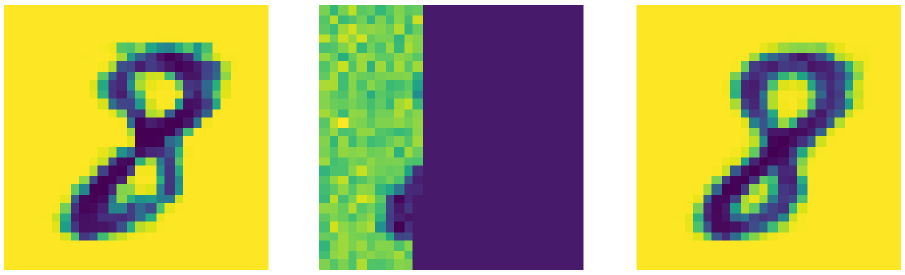
Results for the fourth image:

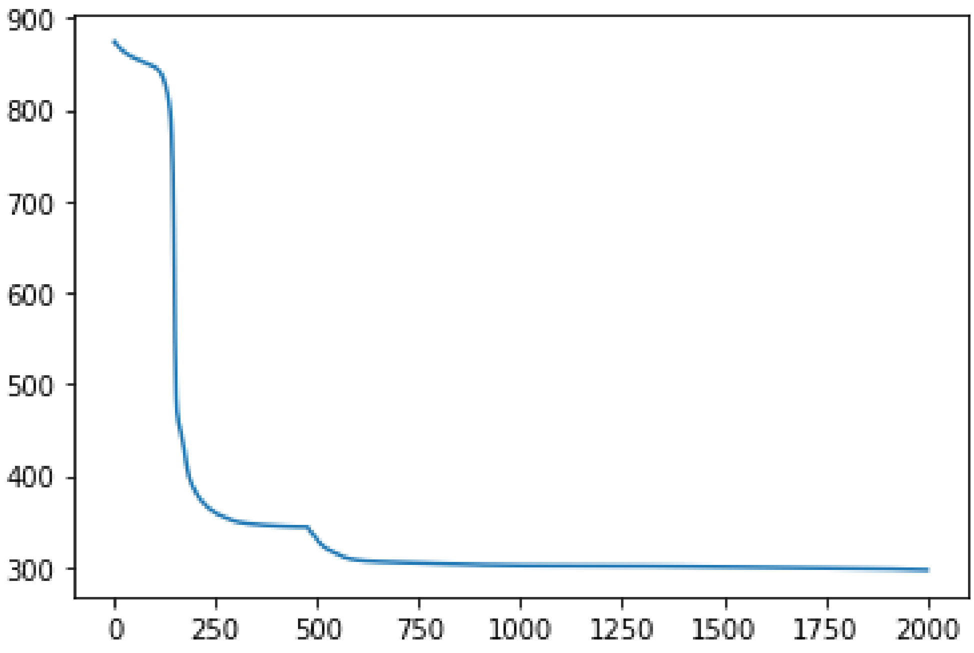
N = 400 , learning rate = 0.0001 , iterations = 2000





N = 300 , learning rate = 0.0001 , iterations = 2000





Problem 3:

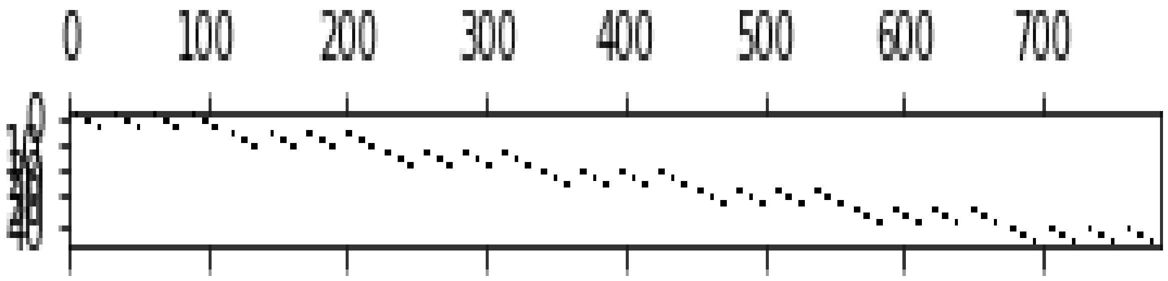
Its exactly the same problem as 2 only with different transformation T (linear).In this transformation we downgrade the original (28 x 28) image into a (7 x 7) image. For doing that we need “windows” (4 x 4) that we pass across the original image περνάω and we take the mean of every pixel of the window as the value of the a pixel in the new image.

The difficulty is that we need the transformation to be linear so we need a matrix T that produces images (7,7) when we multiply it with an image (28,28). For that I created the below program in python.

```
T = np.zeros((49, 784))

k = 0
for j in range(0, 196, 28):
    for i in range(7):
        T[k + i][j * 4:j * 4 + 4] = 1.0/16
        T[k + i][j * 4 + 28:j * 4 + 32] = 1.0/16
        T[k + i][j * 4 + 56:j * 4 + 60] = 1.0/16
        T[k + i][j * 4 + 84:j * 4 + 88] = 1.0/16
        j += 1
    k += 7
```

2 for loops are needed to create the matrix T (49 x 784) originally filled with zeros and afterwards filled with value of 1/16 in the points where if we multiply TX we will get the mean of the 16 desirable pixel.

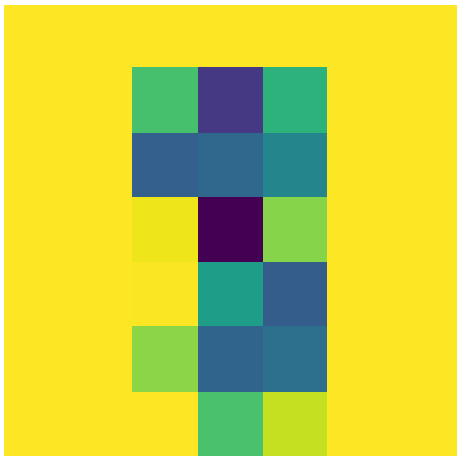
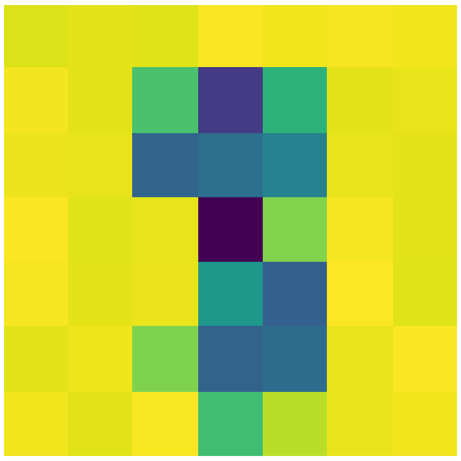
With the spy command we visualize the T matrix as shown below:



Python Code:

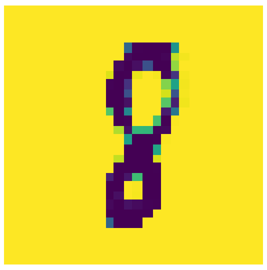
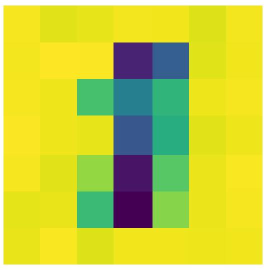
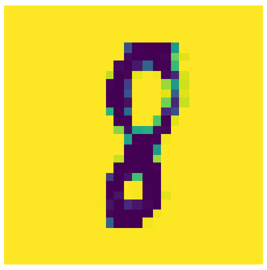
```
Xn0 = Xn[selectImg]
Xi0 = Xi[selectImg]
print(np.shape(Xi0),np.shape(T))
Xi0 = np.reshape(Xi0,(1,784))

# PRINT IDEAL EIGHT TRANSFORMED
X_2D = np.reshape(np.dot(Xi0,T.T), (7, 7))
plt.subplot(1, 2, 1)
plt.axis('off')
plt.imshow(X_2D.T)

# PRINT TRANSFORMED EIGHT

X_2D = np.reshape(Xn0, (7, 7))
plt.subplot(1, 2, 2)
plt.axis('off')
plt.imshow(X_2D.T)
```

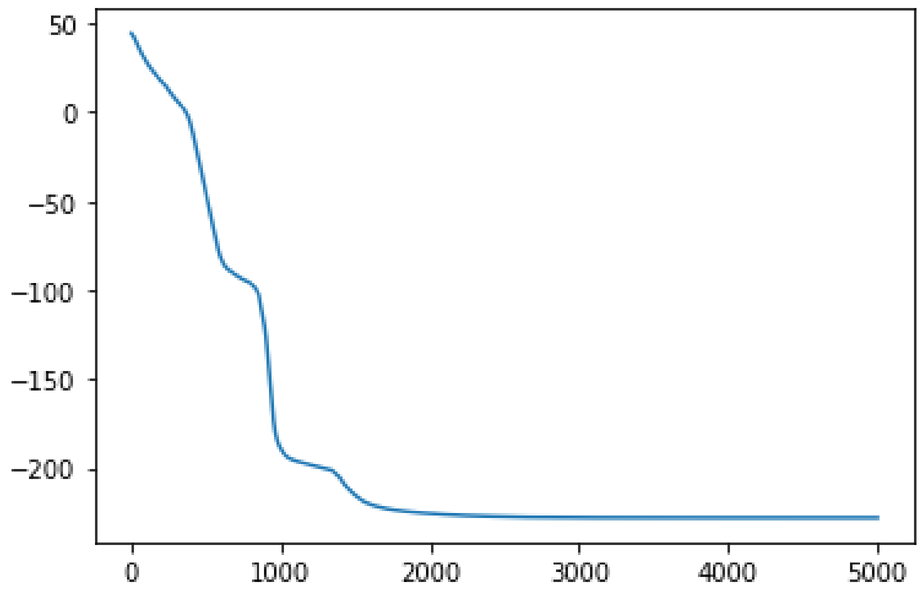
Results of the transformation:

<u>Results for the first image:</u>

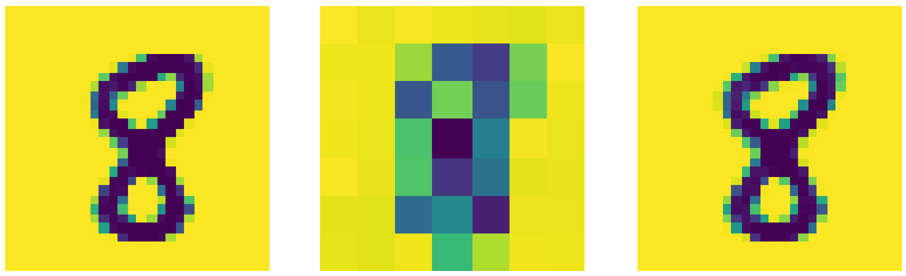
learning rate = 0.0001 , iterations = 5000

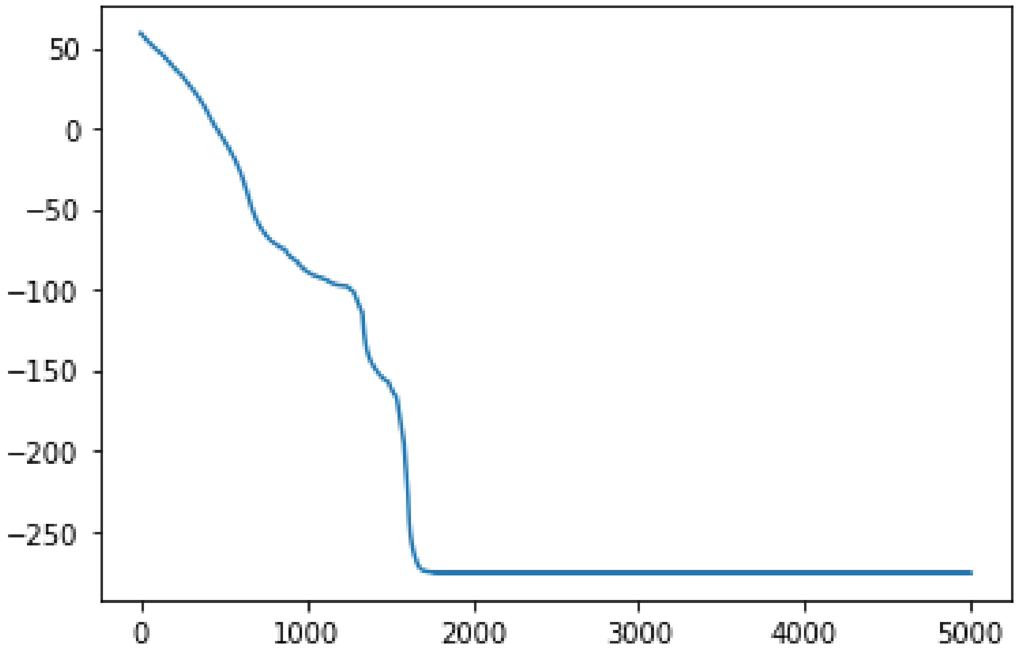






Results for the second image:

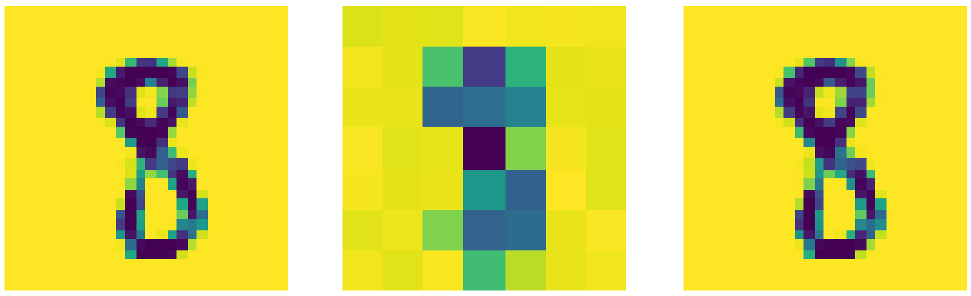
learning rate = 0.0001 , iterations = 5000

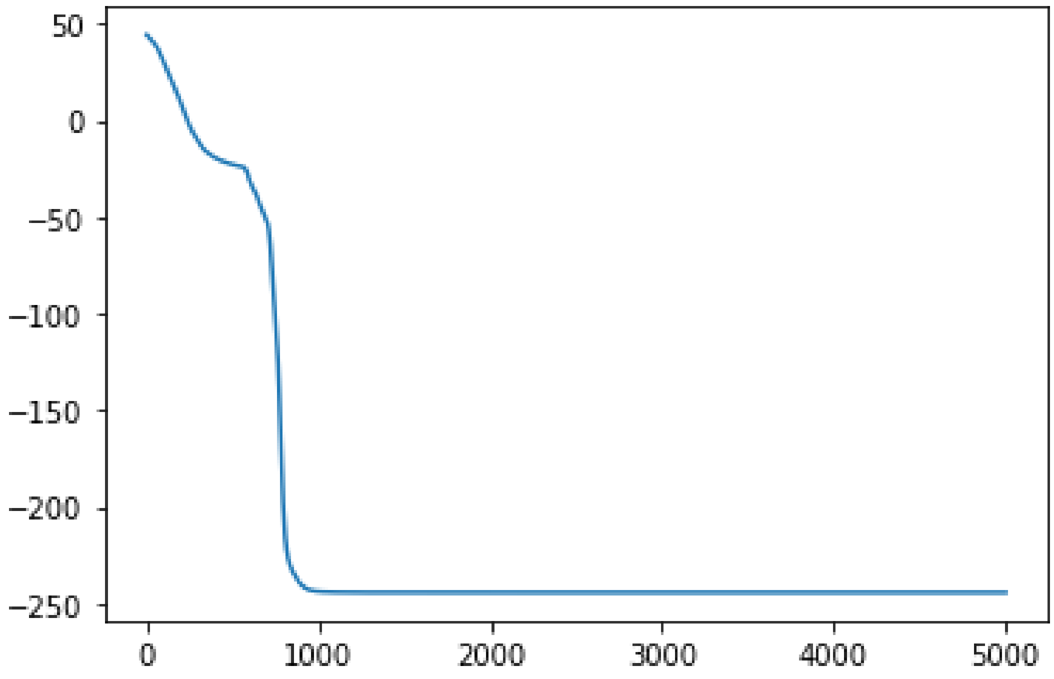




Results for the third image:

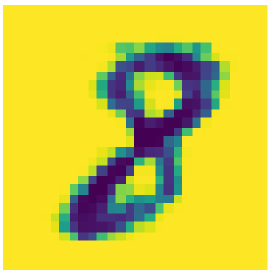
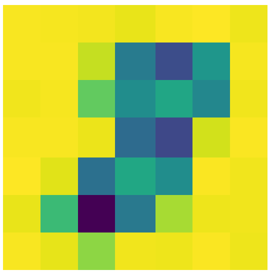
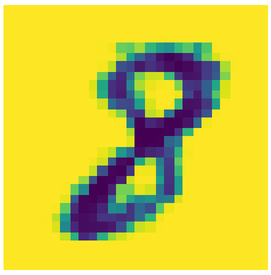
learning rate = 0.0001 , iterations = 5000

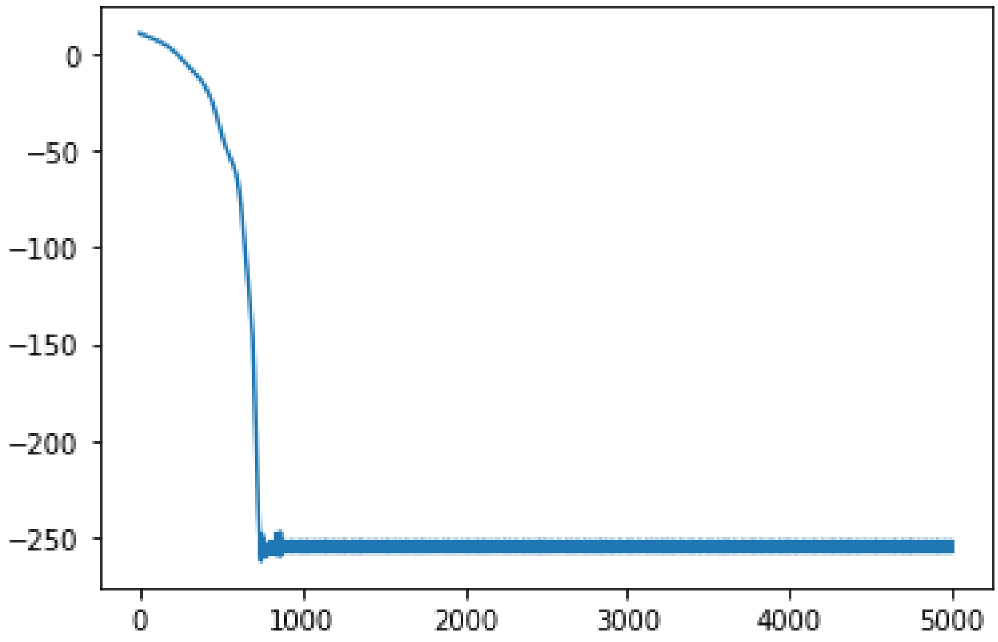




Results for the fourth image:

learning rate = 0.0001 , iterations = 5000

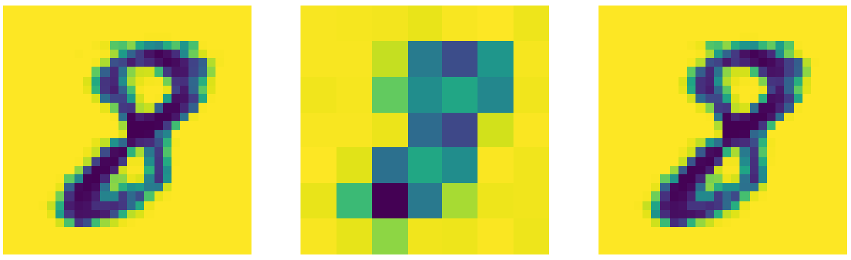


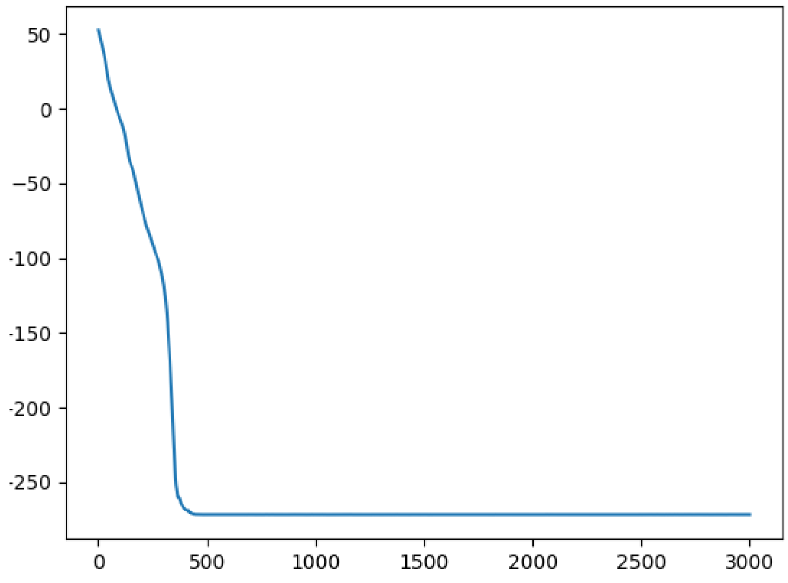
Adam optimization:

```
b1 = 0.9
b2 = 0.85
m = u = np.zeros((1,10))
mm = uu = np.zeros((1,10))
t = 0
for i in range(3000):
    t+=1
    params = printEights(Z, A1, A2, B1, B2)
    x = params['X']
    COST.append(costFunction(params, T, Z, Xn0))
    grad = gradientCalc(A1, A2, params, T, Z, Xn0)
    grad = grad.T
    m = b1 * m + (1 - b1) * grad
    u = b2 * u + (1 - b2) * np.power(grad,2)
    mm = m / (1 - np.power(b1,t))
    uu = u / (1 - np.power(b2,t))
    Z = Z - learning_rate * mm/(np.sqrt(uu)+0.0000001)
```

Results with adam optimization:

Learning rate = 0.01 , b1=0.9 , b2=0.85 , iterations = 3000




50
0
−50
-100
-150
-200
-250
0
500
1000
1500
2000
2500
3000