ΕΠΕΞΕΡΓΑΣΙΑ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΑΛΓΟΡΙΘΜΟΙ ΜΑΘΗΣΗΣ

Ακαδημαϊκό Έτος 2020-2021

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ
UNIVERSITY OF PATRAS

2ο Σετ Ασκήσεων

Φώτιος-Παναγιώτης	Μπασαμάκης	1046975

ΠΡΟΒΛΗΜΑ 1:

Αρχικά φορτώθηκε από το αρχείο “data21.mat” ο generator $G(Z)$ που είναι εκπαιδευμένος στο να παράγει οχτάρια όταν η είσοδος του είναι ένα διάνυσμα Z[1x10] όπου τα στοιχεία του είναι ανεξάρτητες μεταξύ τους τυχαίες μεταβλητές που ακολουθούν την κανονική κατανομή

$N(0,1)$

```
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']
```

Το generative μοντέλο είναι $G(Z), Z$ και αυτό χρησιμοποιήθηκε για την παραγωγή 100 οχταριών σύμφωνα με την παρακάτω διαδικασία:

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

Όπου στην πραγματικότητα είναι η forward εκτέλεση του νευρωνικού δικτύου του generator για είσοδο Z. Τα μεγέθη των μήτρων είναι:

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

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

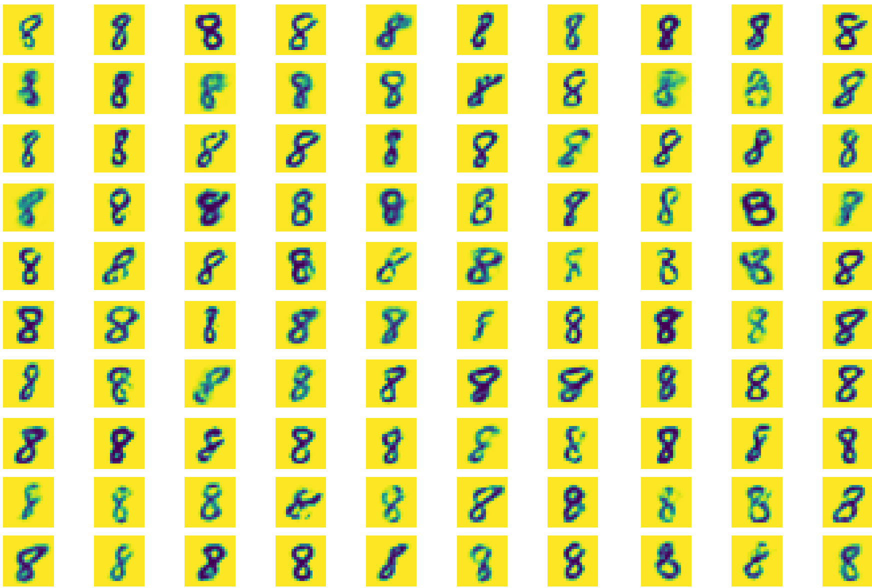
Οι συναρτήσεις Relu και Sigmoid καθώς και η forward λειτουργία στην Python:

```
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
```

κώδικας για εκτύπωση 100 οχταριών σε μορφή πίνακα (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()
```

Αποτελέσματα:



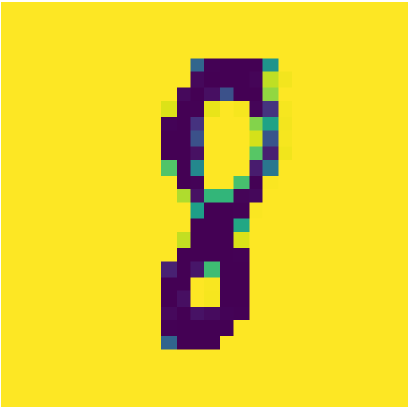
ΠΡΟΒΛΗΜΑ 2:

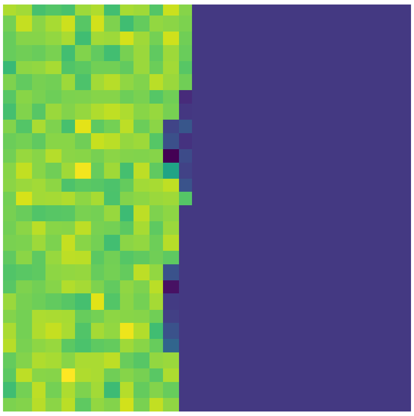
Αρχικά φορτώθηκαν από το αρχείο “data22.mat” οι 4 ιδανικές εικόνες αλλά και οι 4 εικόνες που έχουν υποστεί θόρυβο.

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

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

Στην συνέχεια ορίστηκε ο γραμμικός μετασχηματισμός T ο οποίος υποβάλλεται στις εικόνες του διανύσματος Xn. Αυτός ο μετασχηματισμός είναι ουσιαστικά μια μήτρα μεγέθους (N ,

784) η οποία έχει άσσους στην κύρια διαγώνιο και η οποία πολλαπλασιάζεται με την μήτρα Xn (μεγέθους (1 , 784)) δίνοντας ως αποτέλεσμα ένα διάνυσμα το οποίο εμφανίζει τα N πρώτα στοιχεία του ενώ τα υπόλοιπα από (N έως 784) είναι μηδενικά αυτό εκφρασμένο σε εικόνα είναι:





Το παραπάνω είναι για N = 350 (για N 500 θα είχαμε περισσότερη πληροφορία) και ο κώδικας για τα παραπάνω:

```
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')
plt.imshow(X_2D.T)
```

Έπειτα ορίστηκε η συνάρτηση κόστους σύμφωνα με τον τύπο:

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

```
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))
```

(Η συνάρτηση printEights επιστρέφει την έξοδο X του generator καθώς και τις μήτρες W1 , W2 που θα χρειαστούν στην συνέχεια στον υπολογισμό των παραγώγων.)

Σκοπός είναι η ελαχιστοποίηση της παραπάνω συνάρτησης κόστους ως προς Z δεδομένου ότι αν συμβεί αυτό θα σημαίνει ότι το ελάχιστο Z που θα έχω υπολογίσει θα είναι σε θέση αν μπει ως είσοδο στον generator να δημιουργήσει μια εκτίμηση του Xi (ιδανικού) που θα είναι ικανοποιητική.

Η συνάρτηση κόστους υπολογίζει το λογάριθμο του τετράγωνου της διαφοράς της μετασχηματισμένης εικόνας και της κάθε παραγόμενης από τον generator αφού πρώτα σε αυτή εφαρμοστεί ο μετασχηματισμός T. Εδώ επειδή ο μετασχηματισμός είναι γραμμικός έχουμε το γινόμενο TX. Επίσης υπάρχει και ο όρος $||Z||^2$ που ουσιαστικά είναι η ενέργεια εισόδου.

Για την ελαχιστοποίηση της παραπάνω συνάρτησης χρησιμοποιήθηκε gradient descent αλγόριθμος σύμφωνα με τον οποίο

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

Και για τον υπολογισμό του gradient της συνάρτησης κόστους εφαρμόστηκε ο κανόνας αλυσίδας:

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

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

Ο όρος $(TX - X_n)$ είναι μεγέθους (1 x N) ενώ ο T είναι μεγέθους (N x 784) και τα υπόλοιπα καθαροί αριθμοί που εφαρμόζονται σε κάθε στοιχείο του τελικού διανύσματος.

Προφανώς το τελικό διάνυσμα u2 είναι μεγέθους (1 x 784).

$v2 = u2 \circ f_2'(W_2)$ όπου $f_2'(x)$ είναι η παράγωγος της sigmoid $f_2'(x) = -\frac{e^x}{(1+e^x)^2}$

το παραπάνω γινόμενο καθώς και η $f_2'(x)$ είναι element wise δηλαδή εφαρμόζονται σε κάθε στοιχείο των διανυσμάτων.

$u_1 = A_2^T * v2$ (γινόμενο πινάκων και το u_1 θα έχει μέγεθος (1 x 128))

$v1 = u1 \circ f_2'(W_1)$ όπου $f_1'(x)$ είναι η παράγωγος της ReLu $f_1'(x) = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases}$

και τέλος $u_0 = A_1^T * v1 = \nabla_z \left[log\left(||TX - Xn||^2 \right) \right]$ (το u_0 θα έχει μέγεθος (1 x 10))

Το τελικό gradient της συνάρτησης κόστους επομένως θα είναι:

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

Τα παραπάνω σε κώδικα 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']
```

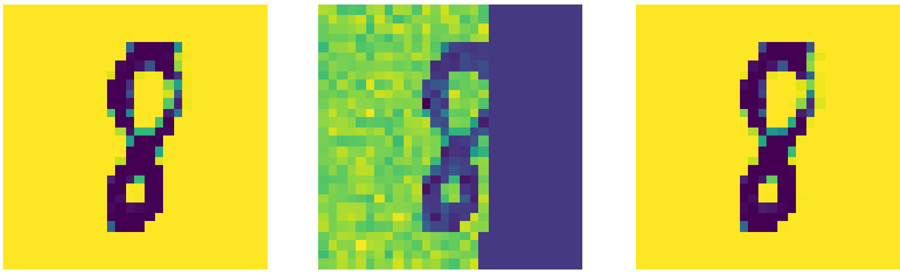
```
    # CALCULATION OF U2 and V2
    temp1 = X - Xn
    temp2 = np.sum(np.power(temp1, 2))
    u2 = 2 * (X - Xn) / temp2
    v2 = u2 * sigmoid_derivative(W2)
    # CALCULATION OF U1 and V1
    u1 = np.dot(A2temp, v2.T)
    v1 = u1 * relu_derivative(W1)

    #return the derivative

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

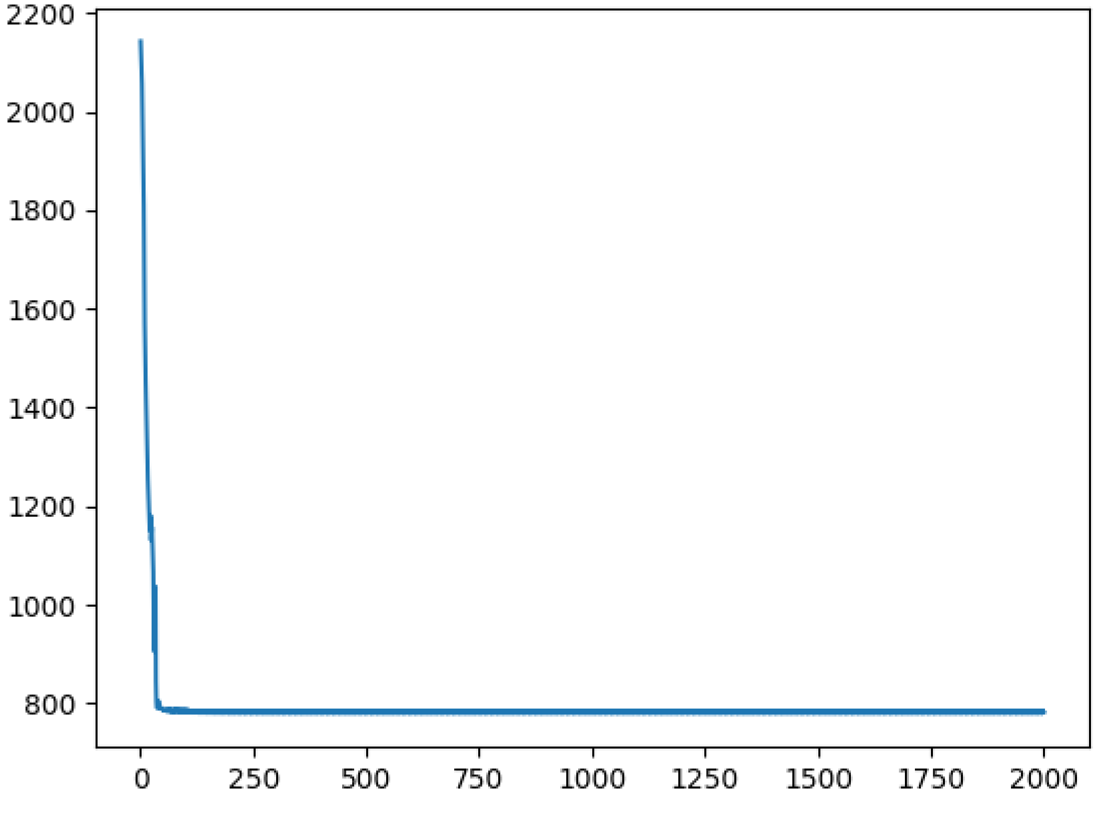
Όλα τα παραπάνω λειτουργούν σε μια επαναληπτική δομή μέχρι να συγκλίνει η συνάρτηση κόστους :

```
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()
```

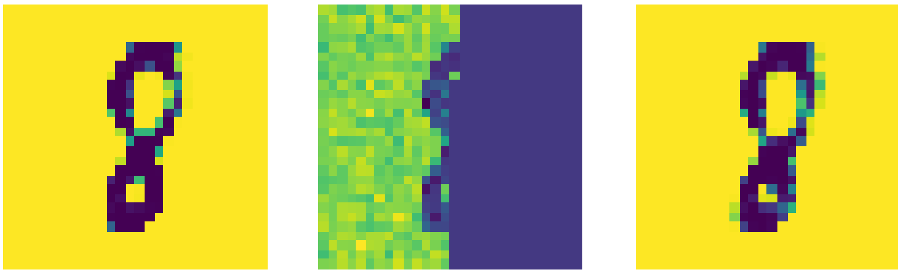
Αποτελέσματα για την πρώτη εικόνα:

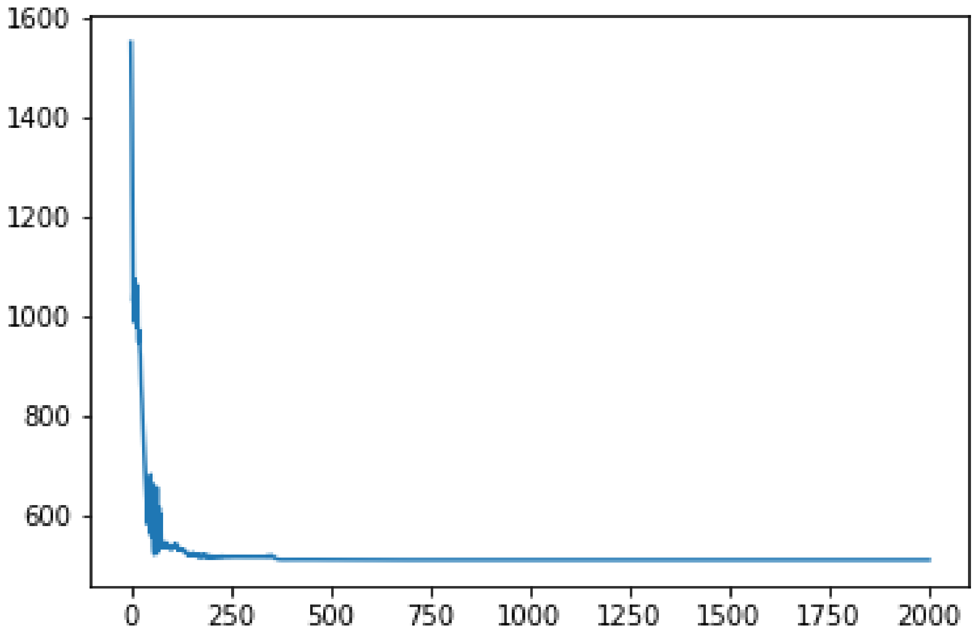
N = 500 , learning rate = 0.001 , iterations = 2000



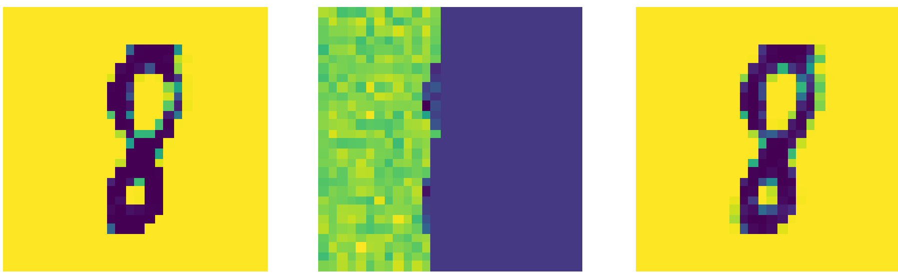


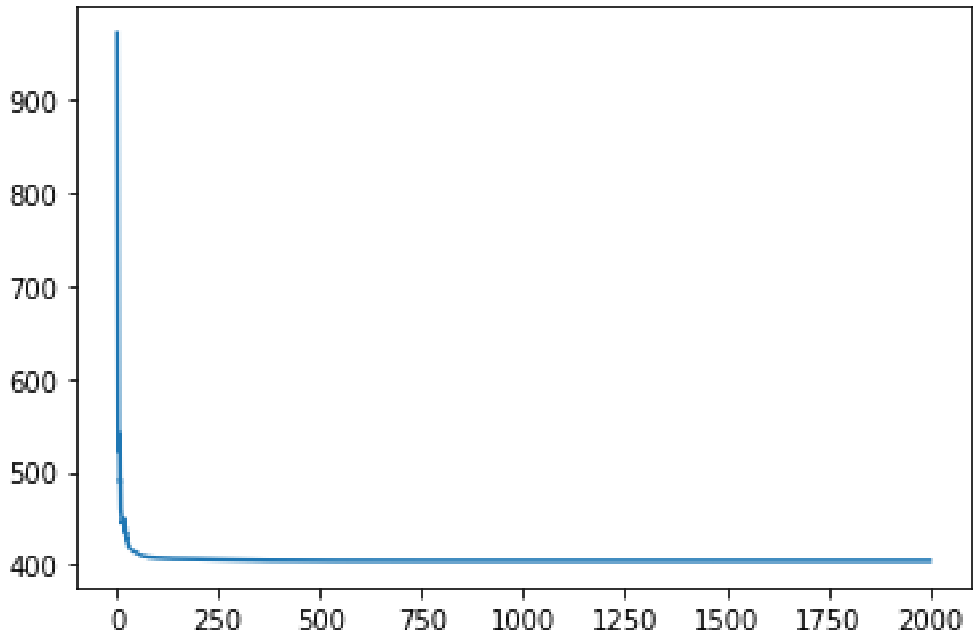
N = 400 , learning rate = 0.001 , iterations = 2000





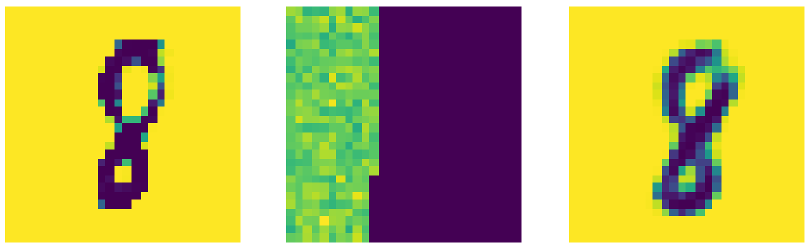
N = 350 , learning rate = 0.001 , iterations = 2000

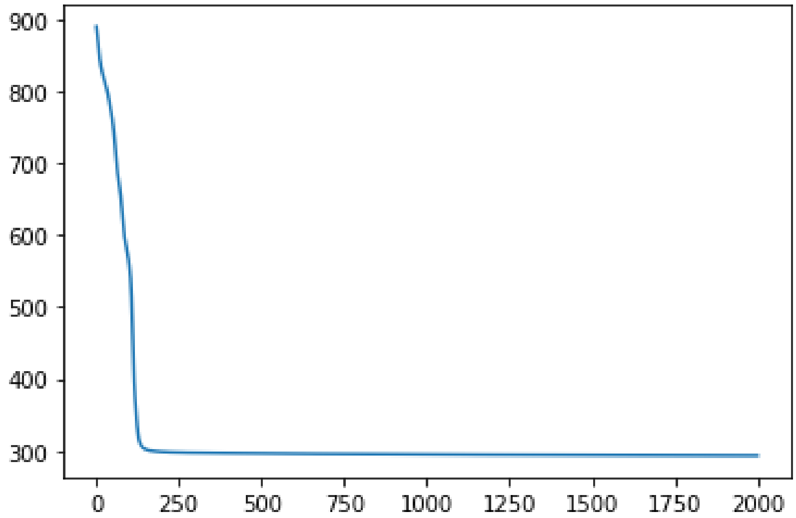




a

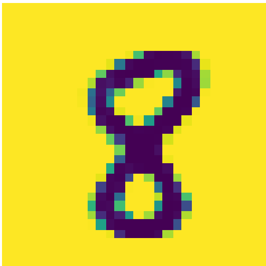
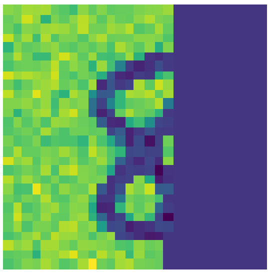
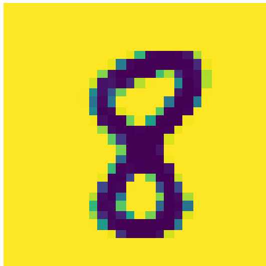
N = 300 , learning rate = 0.0001 , iterations = 2000

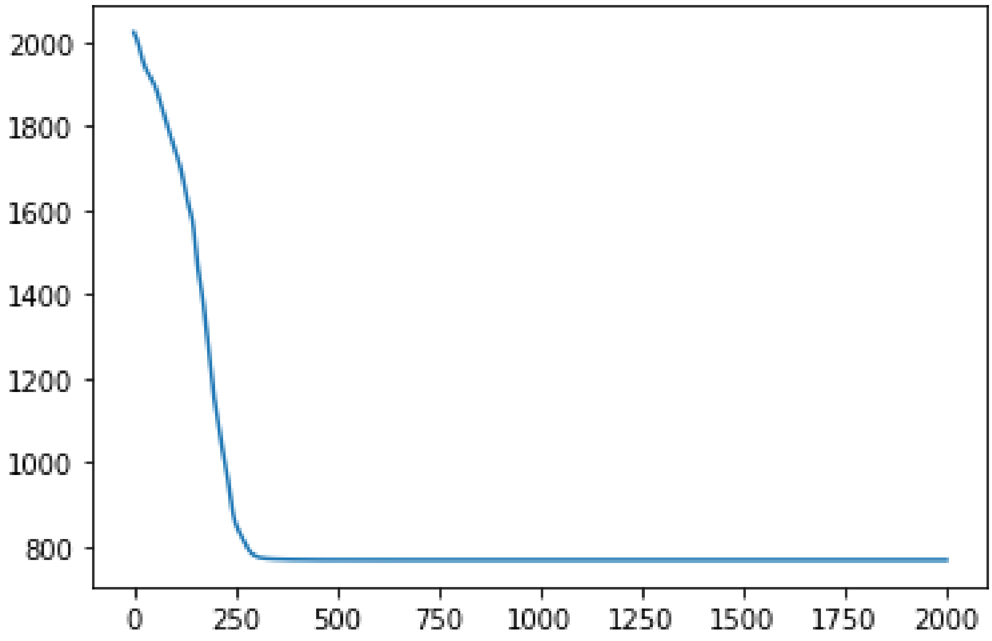




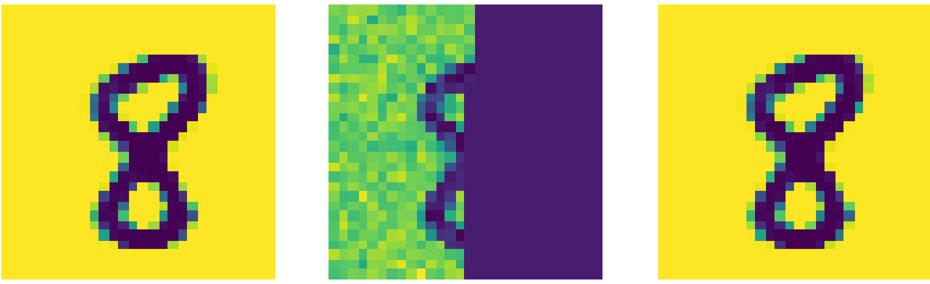
Αποτελέσματα για την δεύτερη εικόνα:

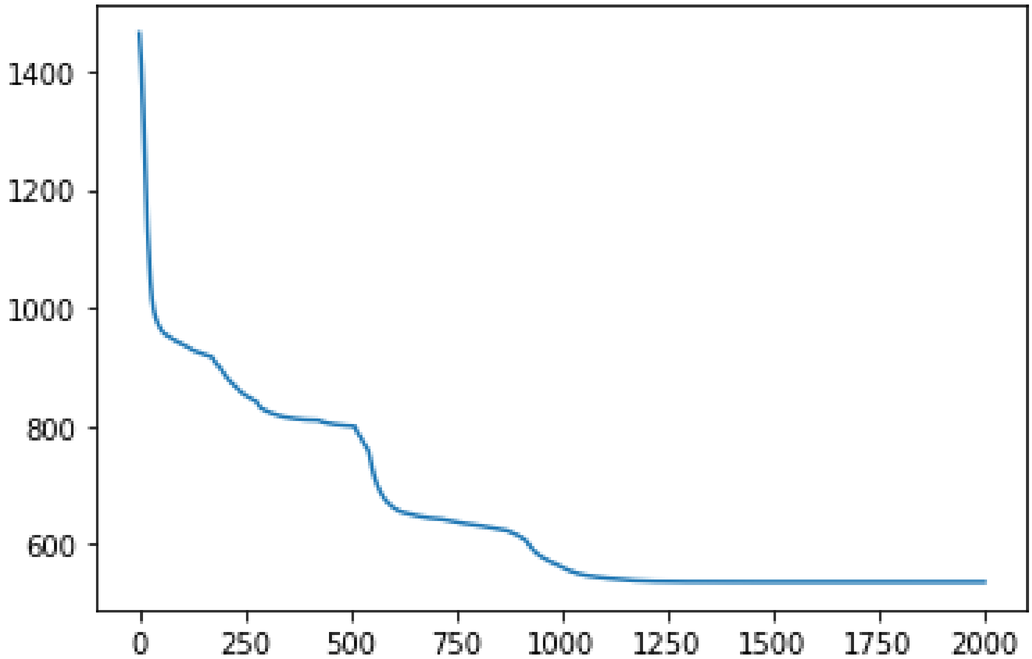
N = 500 , learning rate = 0.0001 , iterations = 2000

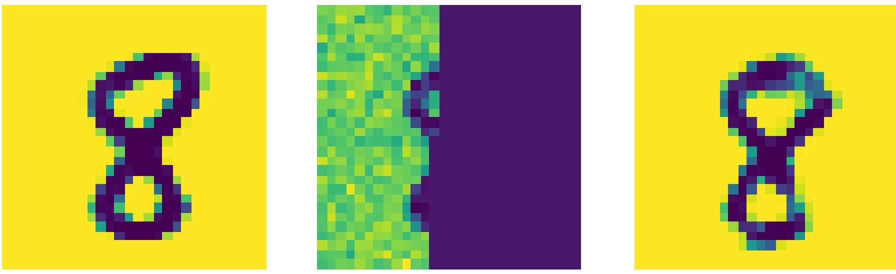


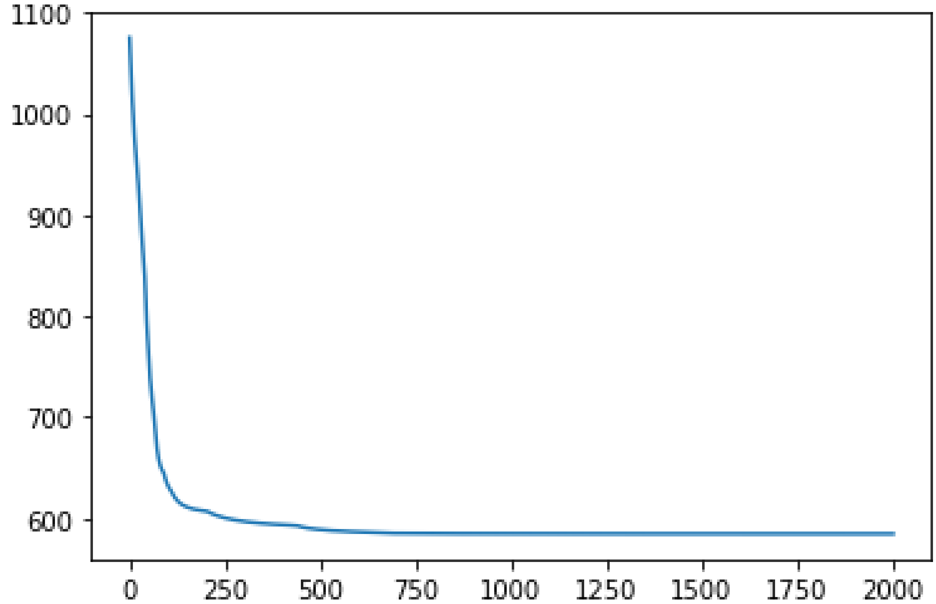
N = 400 , learning rate = 0.0001 , iterations = 2000





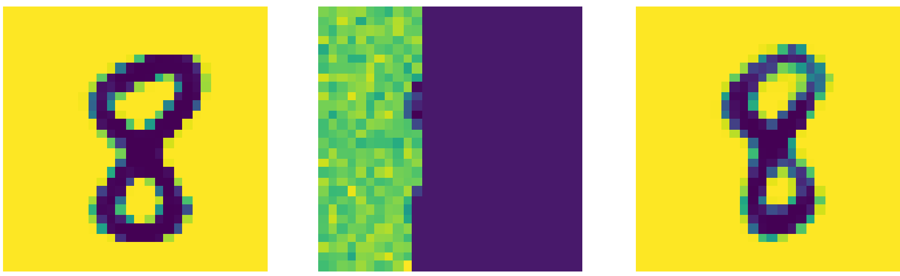
N = 350 , learning rate = 0.0001 , iterations = 2000

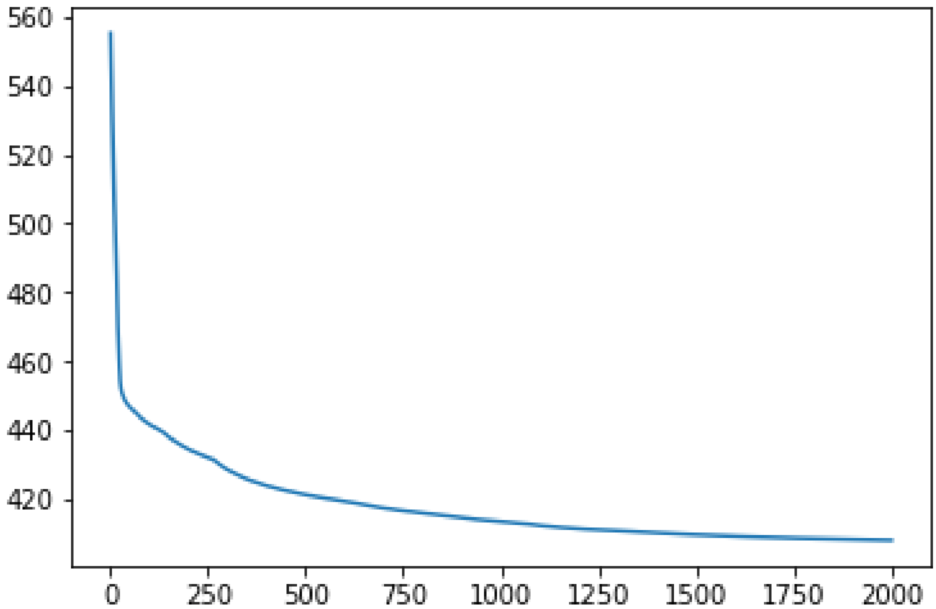




α

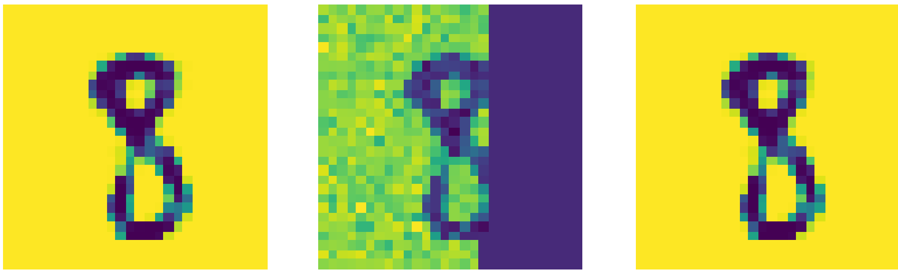
N = 300 , learning rate = 0.0001 , iterations = 2000

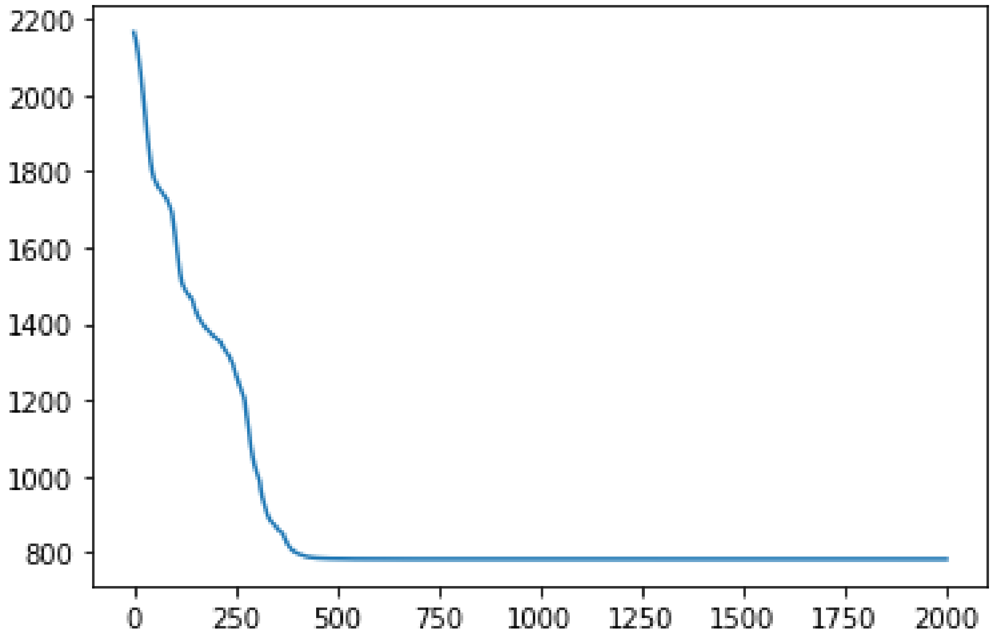




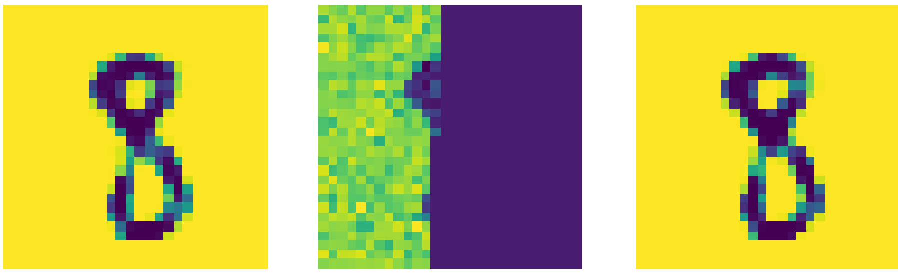
Αποτελέσματα για την τρίτη εικόνα:

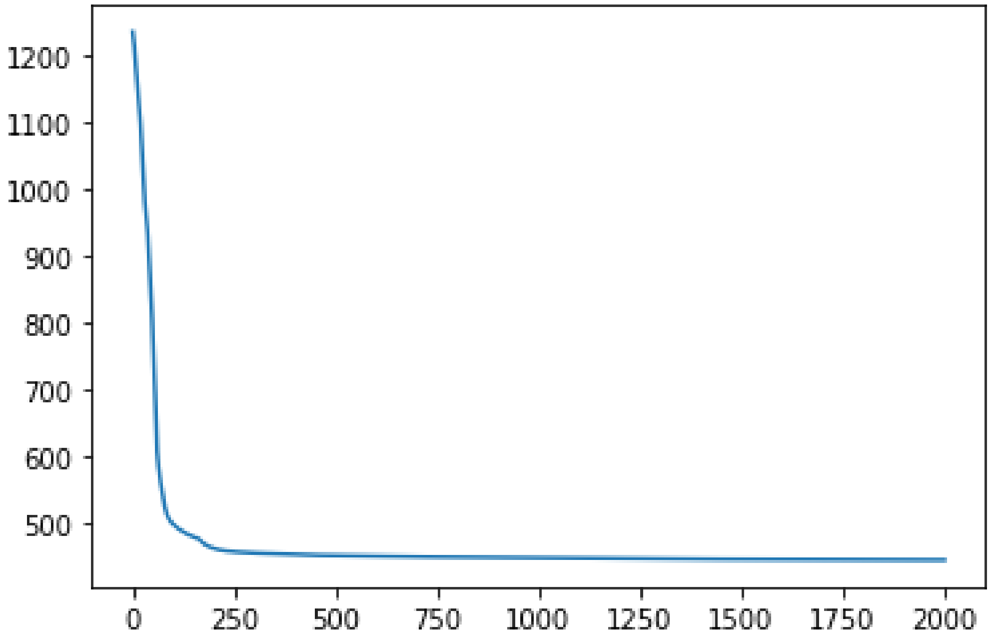
N = 500 , learning rate = 0.0001 , iterations = 2000





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

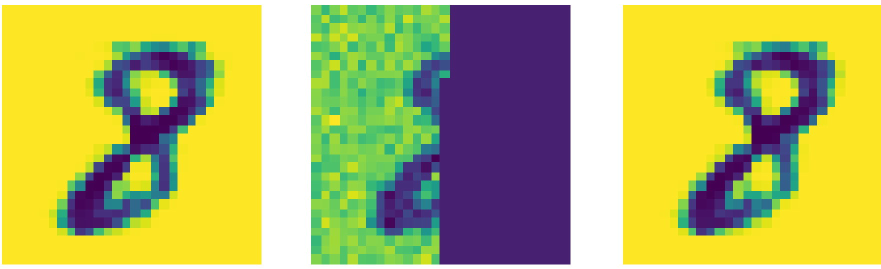


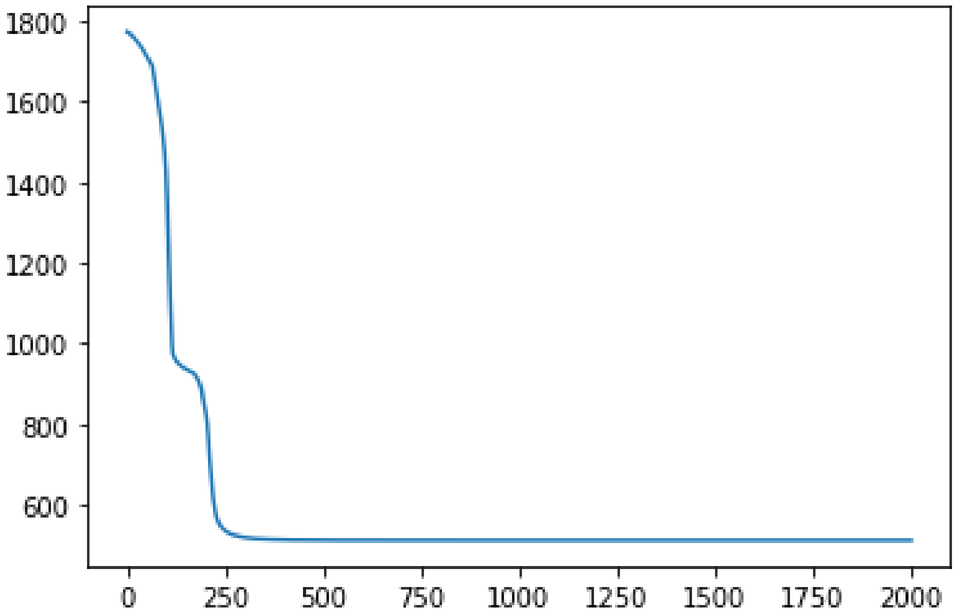


α

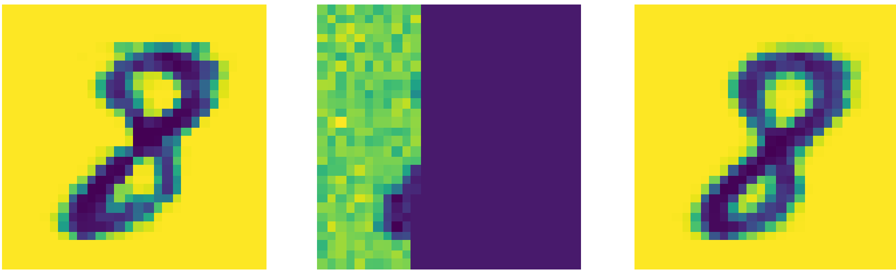
<u>Αποτελέσματα για την τέταρτη εικόνα:</u>

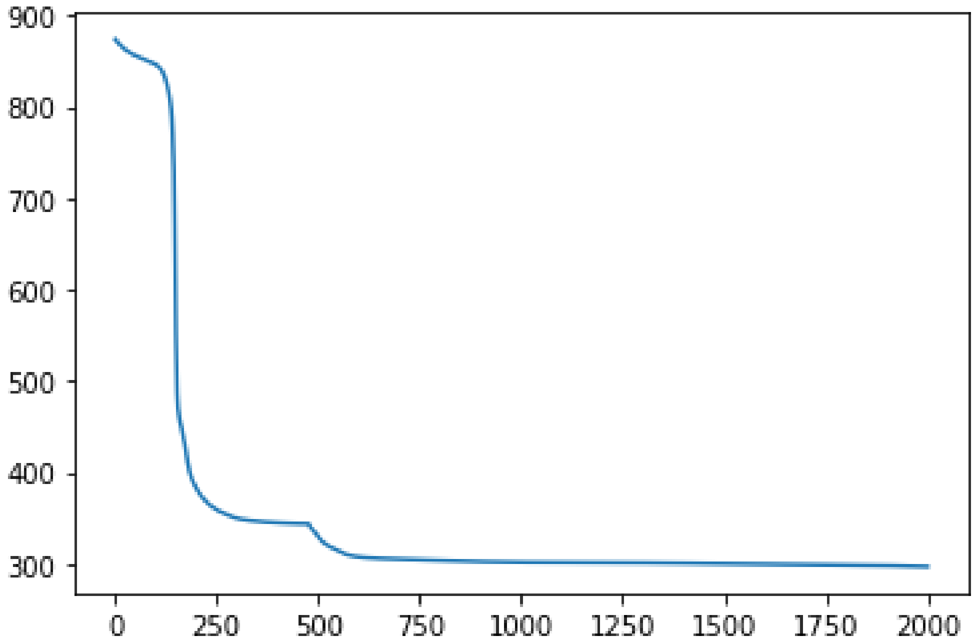
N = 400 , learning rate = 0.0001 , iterations = 2000





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





Παρατηρήσεις:

1) Τα 2000 iterations είναι πολλά δεδομένου ότι ο αλγόριθμος συγκλίνει από τα 200 για learning rate = 0.0001 και ακόμα πιο πριν για learning rate = 0.001 αλλά λόγω της μικρής πολυπλοκότητας του προγράμματος δεν επηρεάζει σε χρόνο
2) Όσο πιο μικρό το N τόσες περισσότερες επαναλήψεις του κώδικα χρειάζεται για να πετύχουμε μια ικανοποιητική εικόνα δεδομένου ότι έχουμε ελλιπή πληροφορία στη συνάρτηση κόστους. Μάλιστα για N < 300 οι εκτιμήσεις αρχίσουν να είναι περισσότερο τυχαίες (όπως φαίνεται και παραπάνω για N = 300)
3) Δεν χρησιμοποιήθηκε Adam algorithm για stochastic optimization δεδομένου ότι η σύγκλιση είναι ιδιαίτερα ομαλή

ΠΡΟΒΛΗΜΑ 3:

Ακριβώς το ίδιο πρόβλημα με το προηγούμενο με διαφορετικό απλά μετασχηματισμό. Η λογική του μετασχηματισμού είναι ότι για να υποβιβάσω την εικόνα από ανάλυση (28 x 28) σε ανάλυση (7 x 7) παίρνω "παράθυρα" (4 x 4) τα οποία τα περνάω από την αρχική εικόνα και παίρνω τον μέσο όρο των pixel του παραθύρου ως ένα pixel στην καινούργια μου εικόνα.

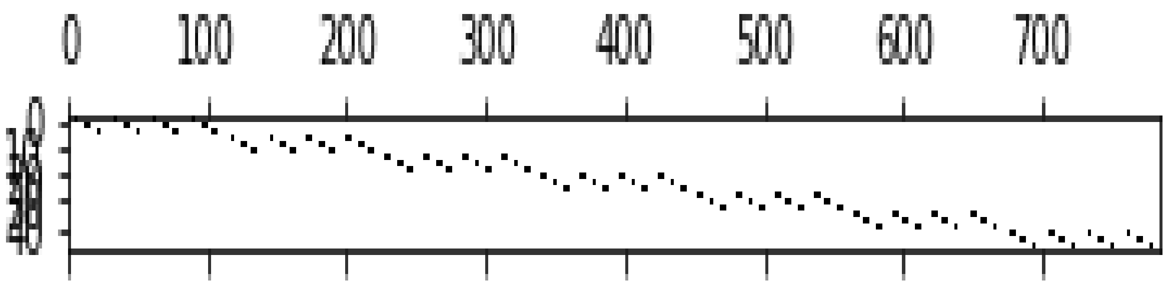
Η δυσκολία είναι ότι όταν εφαρμόζεται reshape σε ένα 2D array και μετατρέπεται σε vector τα γειτονικά pixel δεν μπαίνουν διαδοχικά αλλά πρώτα μπαίνει ολόκληρη η γραμμή (ή η στήλη ανάλογα) και μετά συνεχίζει η επόμενη. Οπότε η δημιουργία ενός γραμμικού μετασχηματισμού T (δηλαδή ενός πίνακα μεγέθους (49 x 784) ο οποίος όταν πολλαπλασιαστεί με το vector της αρχικής μου εικόνας (1 x 784) να μου δίνει ένα διάνυσμα (1 x 49) το οποίο όταν γίνει reshape να είναι η μετασχηματισμένη εικόνα) έπρεπε να δημιουργηθεί με τέτοιο τρόπο ώστε να κρατάει τα pixel ενδιαφέροντος από το διάνυσμα της αρχικής εικόνας.

Για αυτό το λόγο υλοποιήθηκε ο παρακάτω κώδικας

```
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
```

ο οποίος με 2 διαδοχικές επαναλήψεις καταφέρνει να συμπληρώσει στο μητρώο T (49 x 784) το οποίο είναι αρχικοποιημένο με 0 και συμπληρώνεται με κλάσματα 1/16 στα σημεία στα οποία μετά τον πολλαπλασιασμό TX θα δώσουν τον μέσο όρο των 16 επιθυμητών pixel.

Λόγω του μεγάλου μεγέθους καθώς και επειδή είναι αραιή μήτρα το T γίνεται visualize με την εντολή spy και έχει την παρακάτω μορφή:



Για τον έλεγχο του μητρώου θα πολλαπλασιάσω την ιδανική μου εικόνα Xi με το T και θα ελέγξω αν είναι ίδια με την εικόνα Xn που δίνεται. Το μόνο που αναμένω να αλλάζει είναι ο θόρυβος που υπάρχει στην δεύτερη.

Κώδικας:

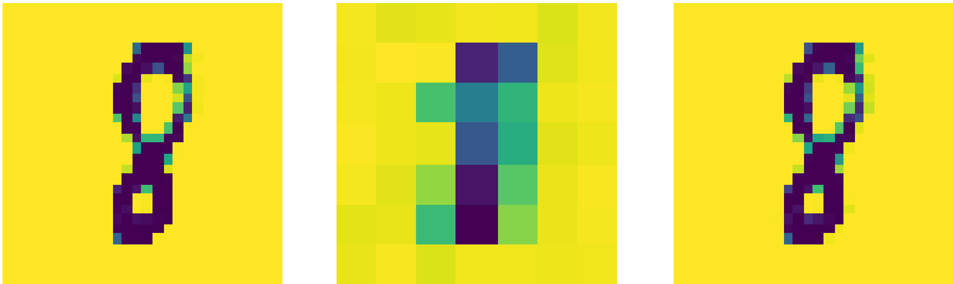
```
Xn0 = Xn[selectImg]
Xi0 = Xi[selectImg]
print(np.shape(Xi0),np.shape(T))
Xi0 = np.reshape(Xi0,(1,784))

# 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)
```

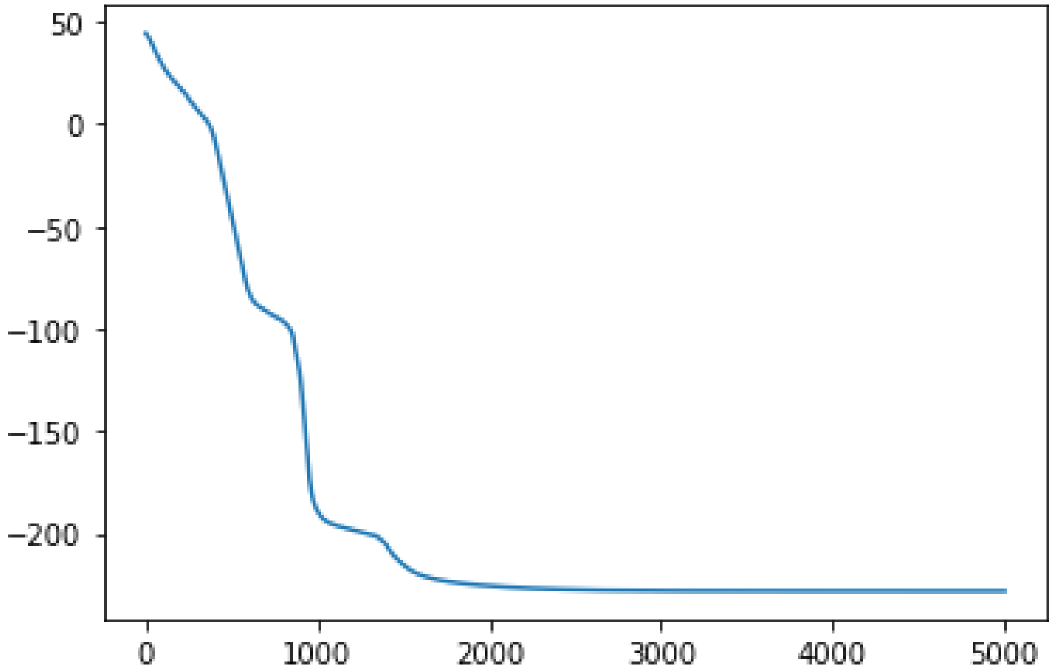
Αποτέλεσμα:

Παρατηρώ ότι οι εικόνες μοιάζουν πάρα πολύ και ότι η δεύτερη απλά έχει λίγο θόρυβο άρα ο μετασχηματισμός μου είναι σωστός. Βάση αυτού του μετασχηματισμού πλέον εφαρμόζω τον ίδιο κώδικα που είχα αναπτύξει και στο πρόβλημα (2.2) παρακάτω φαίνονται τα αποτελέσματα:

Αποτελέσματα για την πρώτη εικόνα:

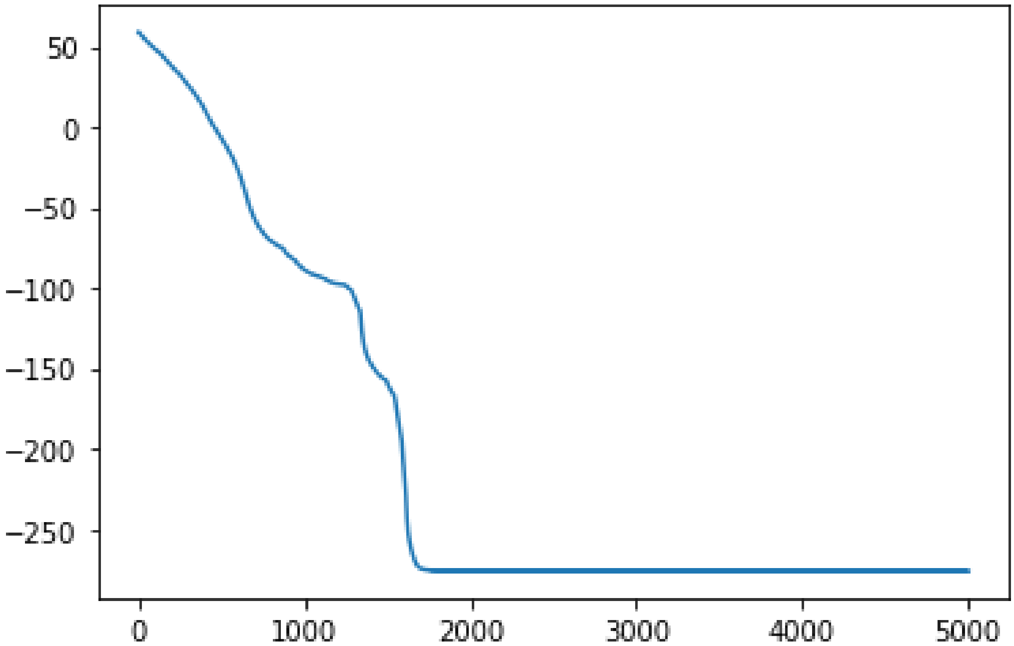
learning rate = 0.0001 , iterations = 5000





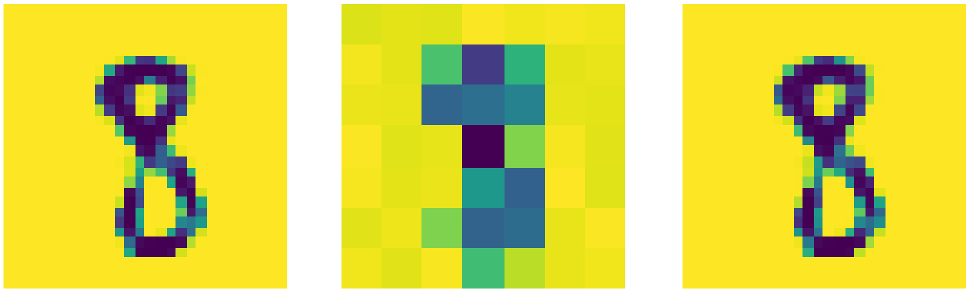
Αποτελέσματα για την δεύτερη εικόνα:

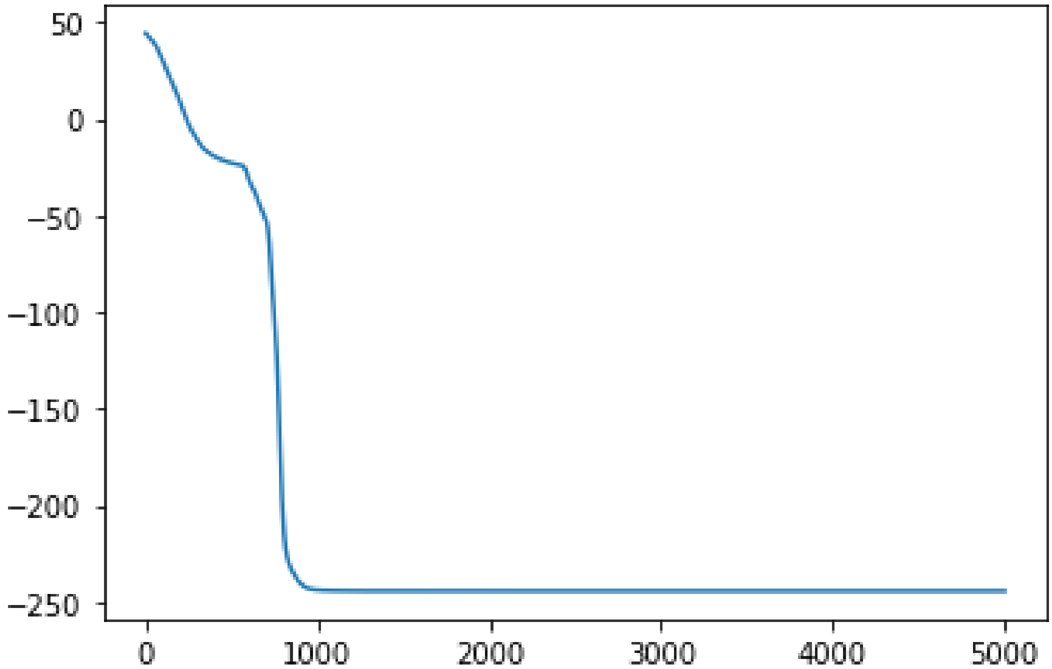
learning rate = 0.0001 , iterations = 5000



Αποτελέσματα για την τρίτη εικόνα:

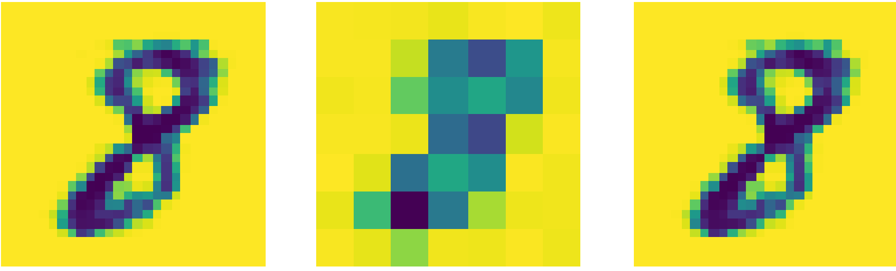
learning rate = 0.0001 , iterations = 5000

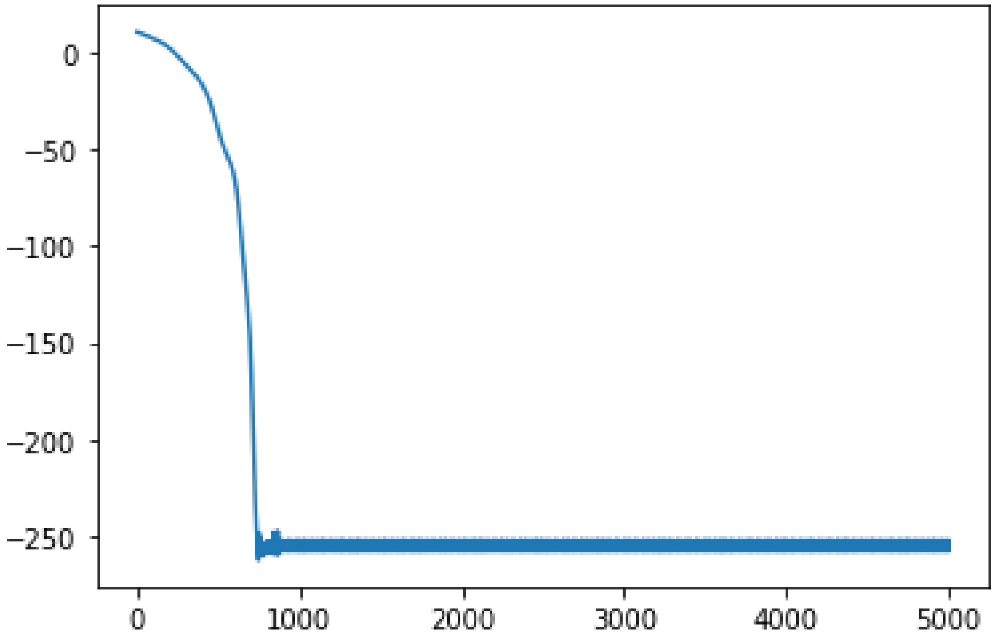




<u>Αποτελέσματα για την τέταρτη εικόνα:</u>

learning rate = 0.0001 , iterations = 5000





Παρακάτω εφαρμόζεται ο αλγόριθμος ADAM για την τελευταία εικόνα της οποίας η σύγκλιση ταλαντώνεται (αν και ενδεχομένως για διαφορετικό learning-rate να μην το κάνει)

Αλλαγές που έγιναν στον κώδικα για την υλοποίηση του αλγόριθμου ADAM:

```
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)
```

Αποτελέσματα:

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