Machine Learning Homework II

PAPAGEORGIOU GEORGIOS
AM: 1092811

December 2024

1 Πρόβλημα Πρώτο

Σε αυτό το πρόβλημα μας δίνετε ένα μοντέλο το οποίο παράγει τυχαίες μεταβλητές οι οποίες είναι χειρόγραφες εικόνες του αριθμού οκτώ πρόκειται δηλαδή για ένα generative model. Το μοντέλο αυτό, δέχεται ως είσοδο ένα διάνυσμα $\mathbf{Z}$ μήκους 10 του οποίου τα στοιχεία είναι ανεξάρτητες υλοποιήσεις μίας Γκαουσιανής με μέση τιμή 0 και διασπορά 1, και δίνει ως έξοδο ένα διάνυσμα $\mathbf{X}$ μήκους 784, το οποίο αν το μετατρέψουμε σε μήτρα 28×28, δηλαδή κάθε 28 στοιχεία του διανύσματος τα κάνουμε μία στήλη του πίνακα, και την αναπαραστήσουμε σαν εικόνα, θα είναι ένα χειρόγραφο του αριθμού οκτώ.

Χρησιμοποιώντας το αρχείο data21.mat που περιέχει τα βάρη και τα offset του νευρωνικού δικτύου που παράγει τις τυχαίες μεταβλητές, και ως μορφή αυτή που περιγράφετε από την (1), προχωράμε στην δημιουργία των υλοποιήσεων. Οι προσομοιώσεις για αυτό το πρόβλημα έγιναν με την χρήση της **Python** και ο κώδικας παρατίθεται στο [A′]

$$\mathbf{W_1} = \mathbf{A_1} \times \mathbf{Z} + \mathbf{B_1}, \quad \mathbf{Z_1} = \max\{\mathbf{W_1}, 0\}(ReLU), \quad \mathbf{W_2} = \mathbf{A_2} \times \mathbf{Z_1} + \mathbf{B_2}, \quad \mathbf{X} = 1./(1 + exp(\mathbf{W_2})) \tag{1}$$

1.1 Δημιουργία Υλοποιήσεων

Δημιουργούμε επαναληπτικά 100 υλοποιήσεις του διανύσματος εισόδου $\mathbf{Z}$, και τα εφαρμόζουμε στο νευρωνικό δίκτυο, κρατώντας κάθε φορά το αποτέλεσμα . Τέλος κάνουμε απεικόνιση των τυχαίων διανυσμάτων, με τον τρόπο που αναφέρθηκε παραπάνω, και παίρνουμε το αποτέλεσμα στο Σχ. 1, προφανώς άμα ξανά τρέξουμε τον αλγόριθμο οι εικόνες θα αλλάξουν, αφού οι είσοδος είναι τυχαία.

Σχήμα 1: Αποτέλεσμα generator για 100 υλοποιήσεις

2 Πρόβλημα Δεύτερο

Σε αυτό το πρόβλημα καλούμαστε να κάνουμε inpainting με την χρήση του generator, δηλαδή θα πάρουμε μία εικόνα η οποία έχει χαμένη πληροφορία και προσθετικό θόρυβο, και θα προσπαθήσουμε να την ανακτήσουμε πλήρως. Θεωρούμε ότι η εικόνα $\mathbf{X_n}$, υπόκειται στον μετασχηματισμό (2)

$$\mathbf{X_n} = \mathbf{TX} + \mathbf{W} \tag{2}$$

όπου $\mathbf{T}$ σε αυτό το πρόβλημα είναι γνωστός μετασχηματισμός και $\mathbf{W}$ ο θόρυβος και $\mathbf{X}$ η ιδανική εικόνα. Από το αρχείο data22.mat θα χρησιμοποιήσουμε τον πίνακα $\mathbf{X_n}$ όπου περιέχει τις εικόνες με θόρυβο, και από κάθε μία στήλη του δηλαδή κάθε μία εικόνα, θα χρησιμοποιήσουμε τα πρώτα N στοιχεία, τα $N+1$ έως το 784 θεωρούνται χαμένα και στόχος μας είναι να τα αποκαταστήσουμε. Ως generator θα χρησιμοποιηθεί το νευρωνικό δίκτυο (1)

2.1 Μήτρα T

Σε αυτό το πρόβλημα και σύμφωνα με τις οδηγίες που δίνονται η μήτρα $\mathbf{T}$ θα έχει διαστάσεις $N \times 784$ και μορφή $\mathbf{T} = [\mathbf{I}\ \mathbf{0}]$ όπου $\mathbf{I}$ ο μοναδιαίος πίνακας διάστασης $N \times N$ και 0 ο μηδενικός πίνακας διάστασης $N \times (784 - N)$. Ουσιαστικά, η μήτρα $\mathbf{T}$ κρατάει την πληροφορία από τα πρώτα N στοιχεία και μηδενίζει τα υπόλοιπα, έτσι όταν αναπαριστούμε την εικόνα αυτά τα pixels εμφανίζονται μάυρα.

2.2 Συνάρτηση Κόστους

Σύμφωνα με την μεθοδολογία της διάλεξης 9 θα χρησιμοποιήσουμε την συνάρτηση κόστους (3)

$$\mathrm{J}(\mathbf{Z}) = \mathrm{M} \log ||\mathbf{X_n} - \mathbf{T}(\mathbf{G}(\mathbf{Z}))||^{\mathbf{2}} + ||\mathbf{Z}||^{\mathbf{2}} \tag{3}$$

πρέπει να λύσουμε το πρόβλημα βελτιστοποίησης (4)1

$$\hat{\mathbf{Z}} = \arg\min_{\mathbf{Z}} \{\mathbf{J}(\mathbf{Z})\} \tag{4}$$

δηλαδή ψάχνουμε ένα $\hat{\mathbf{Z}}$ (εκτιμητής) πού όταν το εισάγουμε στον generator $\mathbf{G(Z)}$ η ποσότητα $\mathbf{TG}(\hat{\mathbf{Z}})$ θα είναι όσο πιο κοντά γίνετε στο $\mathbf{X_n}$, οπού είναι το διάνυσμα με την χαμένη πληροφορία. M είναι το μήκος του διανύσματος $\mathbf{X_n}$.

Λύση Προβλήματος Βελτιστοποίησης

Για να λύσουμε το πρόβλημα (4) θα κάνουμε χρήση το αλγορίθμου gradient descent, δηλαδή θα πρέπει να εκτελέσουμε την αναδρομή (5)

$$\mathbf{Z_n} = \mathbf{Z_{n-1}} - \mu \nabla_{\mathbf{Z}} \mathrm{J}(\mathbf{Z}) ./ \sqrt{\mathbf{c} + \mathbf{P(t)}} \tag{5}$$

όπου $\mathbf{P(t)} = (1-\lambda)\mathbf{P(t-1)} + \lambda(\nabla_{\mathbf{Z}_{t-1}} J(\mathbf{Z}))$. Η διαίρεση τς κλίσης στοιχείο προς στοιχείο με την τετραγωνική ρίζα της ισχύος των παραγώγων είναι η εφαρμογή του αλγορίθμου Adam που βοηθάει στην ομοιόμορφη σύγκλιση όλων των παραγώγων.

Υπολογισμός $\nabla_{\mathbf{Z}} \mathbf{J}(\mathbf{Z})$

Για τους υπολογισμούς θα χρησιμοποιήσουμε ευκλείδεια νόρμα. Θα ακολουθήσουμε την μεθοδολογία που παρουσιάζετε στο αρχείο derivaitives.pdf σύμφωνα με αυτή η κλήση της συνάρτησης $J(\mathbf{Z})$ θα είναι,

$$\nabla_{\mathbf{Z}} \mathrm{J}(\mathbf{Z}) = \nabla_Z [M \log ||\mathbf{X_n} + \mathbf{TX}||^2 + ||Z||^2] = M\mathcal{U}_0 + 2Z \tag{6}$$

Άρα τώρα μένει να υπολογίσουμε το $\mathcal{U}_0$ που είναι ουσιαστικά η κλήση $\nabla_{\mathbf{Z}} \log ||\mathbf{X_n} - \mathbf{TG}(\mathbf{Z})||^2$, γνωρίζοντας ότι $\mathbf{X} = \mathbf{G}(\mathbf{Z})$ θα ξεκινήσουμε πρώτα από το $\mathcal{U}_2 = \nabla_{\mathbf{X}} \log ||\mathbf{X_n} - \mathbf{TX}||^{\mathbf{2}}$ και θα συνεχίσουμε, αναδρομικά μέχρι το

$\mathcal{U}_0$. Αρχικά,

$$\mathcal{U}_2 = \nabla_{\mathbf{X}} \log ||\mathbf{X_n} - \mathbf{TX}||^{\mathbf{2}} = \frac{\nabla_{\mathbf{X}} ||\mathbf{X_n} - \mathbf{TX}||^{\mathbf{2}}}{||\mathbf{X_n} - \mathbf{TX}||^2} = \frac{\nabla_{\mathbf{X}} (\mathbf{X_n} - \mathbf{TX})^{\mathbf{T}} (\mathbf{X_n} - \mathbf{TX})}{||\mathbf{X_n} - \mathbf{TX}||^{\mathbf{2}}} =$$

$$\frac{\nabla_{\mathbf{X}} [\mathbf{X_n^T X_n} - \mathbf{X_n^T}(\mathbf{TX}) - (\mathbf{TX})^{\mathbf{T}} \mathbf{X_n} + (\mathbf{TX})^{\mathbf{T}} \mathbf{TX})]}{||\mathbf{X_n} - \mathbf{TX}||^2} \Leftrightarrow$$

παρατηρούμε ότι ο όρος $\mathbf{X_n^T}(\mathbf{TX})$ και $(\mathbf{TX})^{\mathbf{T}} \mathbf{X_n}$ είναι και τα δύο βαθμωτά μεγέθη αφού το $\mathbf{X_n^T}$ έχει διαστάσεις 1×784 και το $(\mathbf{TX})$, 784×1, και ο ένας πίνακας αποτελεί ανάστροφος του άλλου, και αφού πρόκειται για βαθμωτό μέγεθος, είναι ουσιαστικά η ίδιες ποσότητες.

$$\mathcal{U}_2 = \frac{\nabla_{\mathbf{X}} [\mathbf{X_n^T X_n} - 2\mathbf{X^T T^T X_n} + (\mathbf{X^T T^T TX})]}{||\mathbf{X_n} - \mathbf{TX}||^2} \Rightarrow$$

Ο όρος $\mathbf{X^T T^T TX}$ αφού ο $\mathbf{T^T T}$ είναι συμμετρικός $N \times N$, αποτελεί μια τετραγωνική μορφή, $\mathbf{Q(X)} = \mathbf{X^T AX}$. και εύκολα αποδεικνύεται ότι $\nabla_{\mathbf{X}} \mathbf{Q(X)} = \mathbf{2AX}$, Άρα συνεχίζοντας με την παραγώγιση

$$\mathcal{U}_2 = \frac{-2\mathbf{T^T X_n} + 2\mathbf{T^T TX}}{||\mathbf{X_n} - \mathbf{TX}||^2} = \frac{-2\mathbf{T^T}(\mathbf{X_n} - \mathbf{TX})}{||\mathbf{X_n} - \mathbf{TX}||^2}$$

το οποίο είναι ένα διάνυσμα με διαστάσεις 784×1.
Έπειτα υπολογίζουμε το $\mathcal{V}_2 = \mathcal{U}_2 \odot f_2'(W_2)$ όπου και αυτό θα έχει διαστάσεις 784×1, αφου προκύπτει από τον πολλαπλασιασμό στοιχείο προς στοιχείο.
Υπολογίζουμε το $\mathcal{U}_1 = \mathbf{A_2^T} \mathcal{V}_2$ που θα έχει διαστάσεις 128×1 και το $\mathcal{V}_1 = \mathcal{U}_1 \odot f_1'(\mathbf{W_1})$ με διαστάσεις 128×1, και τελικά το ζητούμενο $\mathcal{U}_0 = \mathbf{A_1^T} \mathcal{V}_1$ το οποίο είναι ένα διάνυσμα με διαστάσεις 10×1. Τέλος η κλίση δίνεται με αντικατάσταση στην (6).

Αποτελέσματα

Αφού έχουμε υπολογίσει τις παραγώγους και τον πίνακα μετασχηματισμού $\mathbf{T}$ είμαστε σε θέση να εφαρμόσουμε την αναδρομή (5) ώστε να λάβουμε το επιθυμητό $\hat{\mathbf{Z}}$. Τρέχουμε τον αλγόριθμο για διάφορα N ως που να μην μπορεί να ανακτήσει την εικόνα επαρκώς ικανοποιητικά. Στο Σχ. (2) παρατηρούμε τα αποτελέσματα για τα διάφορα N, για κάθε υπό εικόνα του Σχ. (2) στην πρώτη στήλη παρουσιάζονται οι ιδανικές εικόνες (πίνακας $\mathbf{X_i}$), στην δεύτερη οι αλλοιωμένες που προσπαθούμε να ανακτήσουμε και στην τρίτη οι ανακτημένες, είναι φανερό ότι για N=500 η ανάκτηση είναι σχεδόν τέλεια ενώ καθώς μειώνουμε το N, δηλαδή δίνουμε όλο και λιγότερη πληροφορία στον αλγόριθμο, η ποιότητα των αποτελεσμάτων φθίνει, ως που στο N=300 η εικόνα εμφανίζετε αισθητά παραμορφωμένη σε σχέση με την αρχική. Για το gradient descent χρησιμοποιήθηκε βήμα $\mu = 0.01$ και για τον Adam $\lambda = 0.01$. Στο Σχ. (3) φαίνεται η πορεία του κόστους καθώς ο αλγόριθμός συγκλίνει.

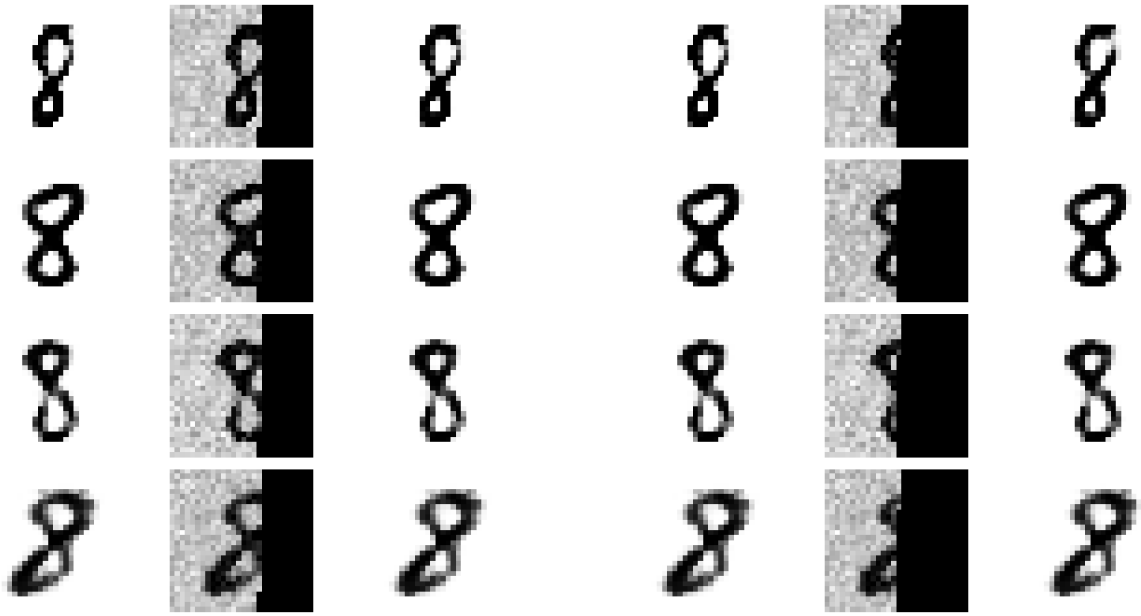
Οι προσομοιώσεις έγιναν σε **Python** και ο κώδικας για αυτο το πρόβλημα παρατίθεται στο [Β΄]

3 Πρόβλημα Τρίτο

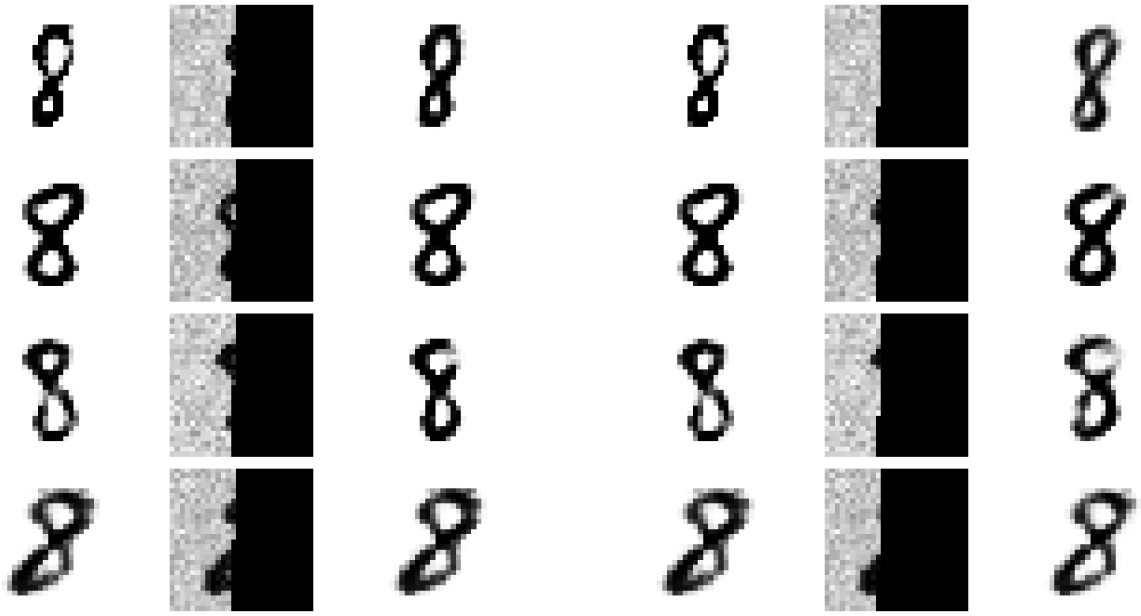
Σε αυτό το πρόβλημα θα κάνουμε αύξηση της ανάλυσης μίας εικόνας. Το πρόβλημα αυτό είναι ισοδύναμο με το προηγούμενο από μαθηματικής απόψεως, δηλαδή ψάχνουμε πάλι ένα $\hat{\mathbf{Z}}$ (εκτιμητή) πού όταν υπολογίσουμε το $\mathbf{TG}(\hat{\mathbf{Z}})$ θα είναι όσο πιο κοντά γίνεται στο $\mathbf{X_n}$, αλλάζει μόνο ο μετασχηματισμός $\mathbf{T}$. Για αυτό το ερώτημα ως $\mathbf{X_n}$ χρησιμοποιήθηκε κάθε μία από τις στήλες του πίνακα $\mathbf{X_n}$ που βρίσκεται στο αρχείο data23.mat. Και εδώ ως generator θα χρησιμοποιηθεί το νευρωνικό δίκτυο (1).

3.1 Υπολογισμός Μήτρας T

Εδώ η μήτρα αυτή θα πρέπει να χαμηλώνει την ανάλυση της εικόνας που παράγει ο generator. Αυτό θα γίνει τοποθετώντας στην εικόνα ένα grid 7×7 και παίρνοντας τον μέσο όρο κάθε κουτιού του grid και κάνοντας το αυτό

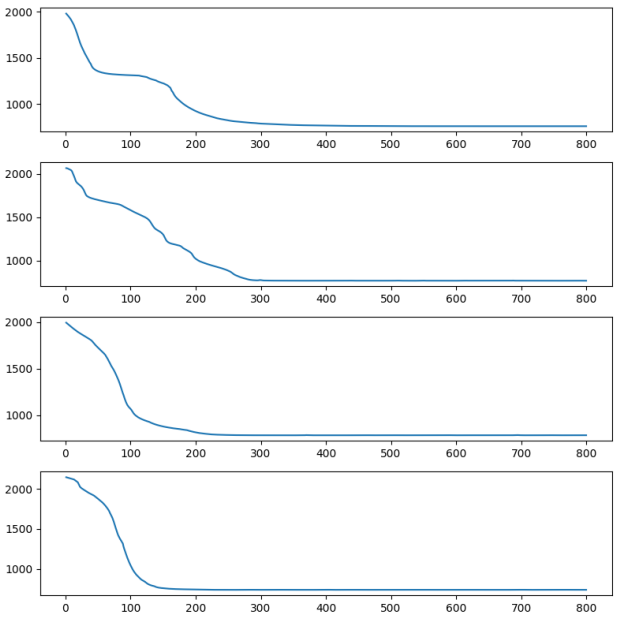


N = 500 N = 400

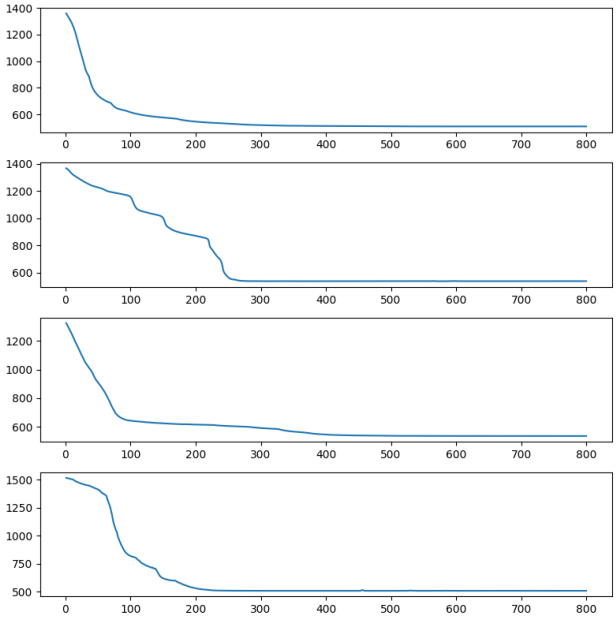


N = 350 N = 300

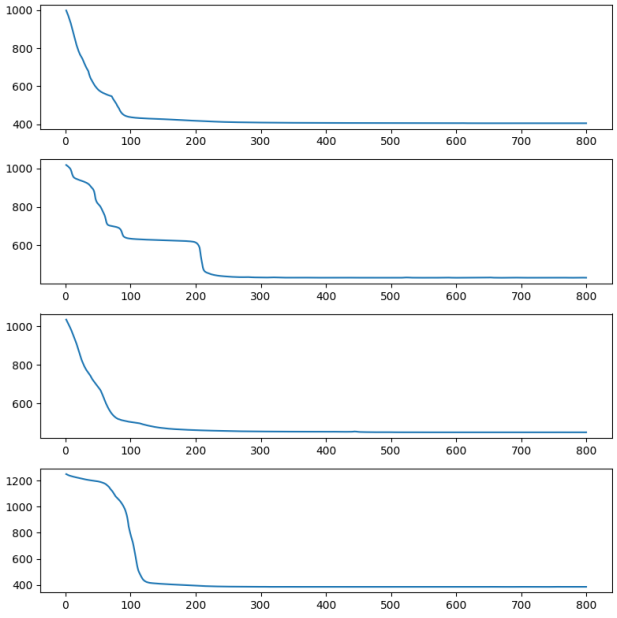
Σχήμα 2: Ανακτημένες Εικόνες



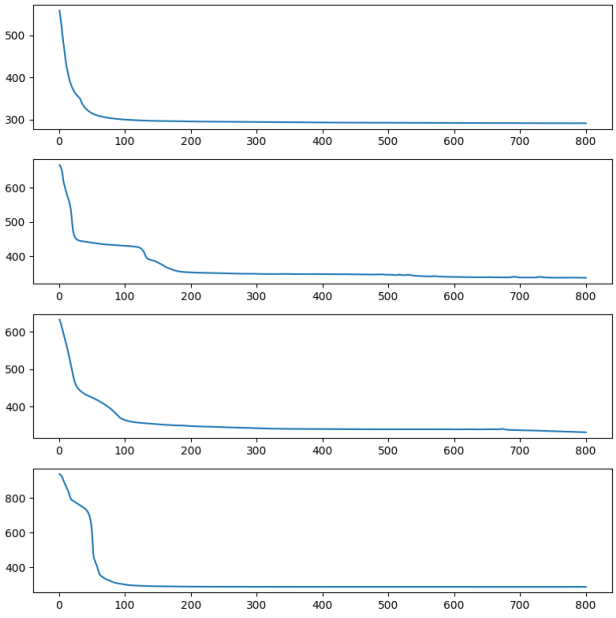
N = 500



N = 400

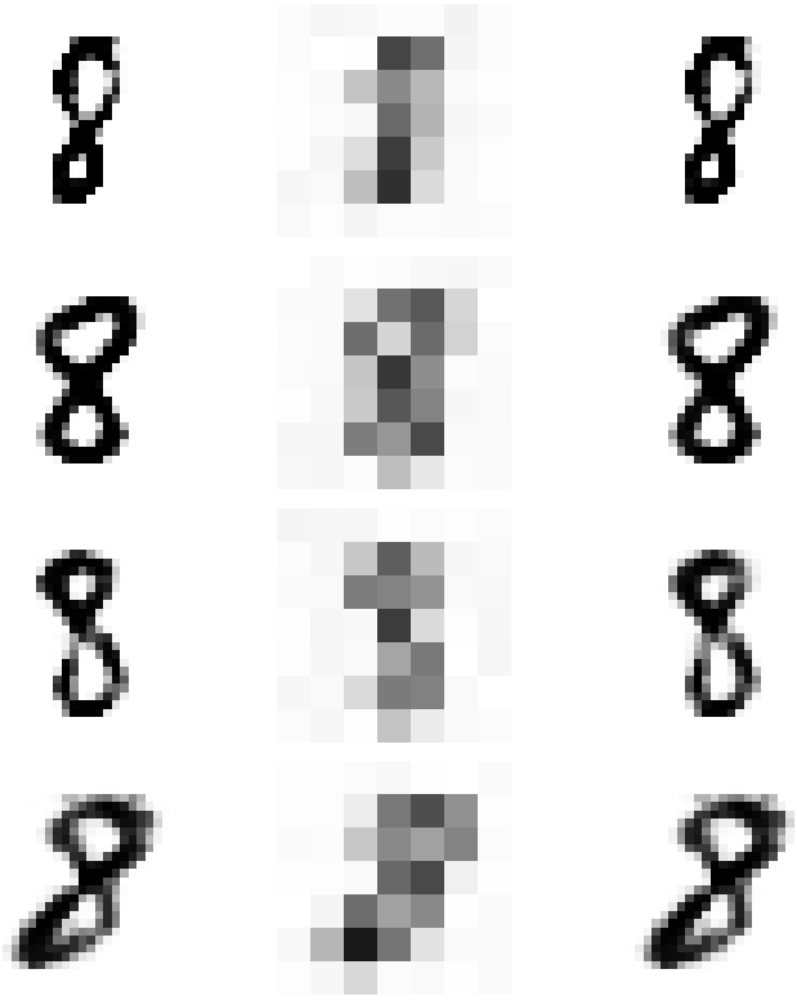


N = 350



N = 300

Σχήμα 3: Πορεία του κόστους



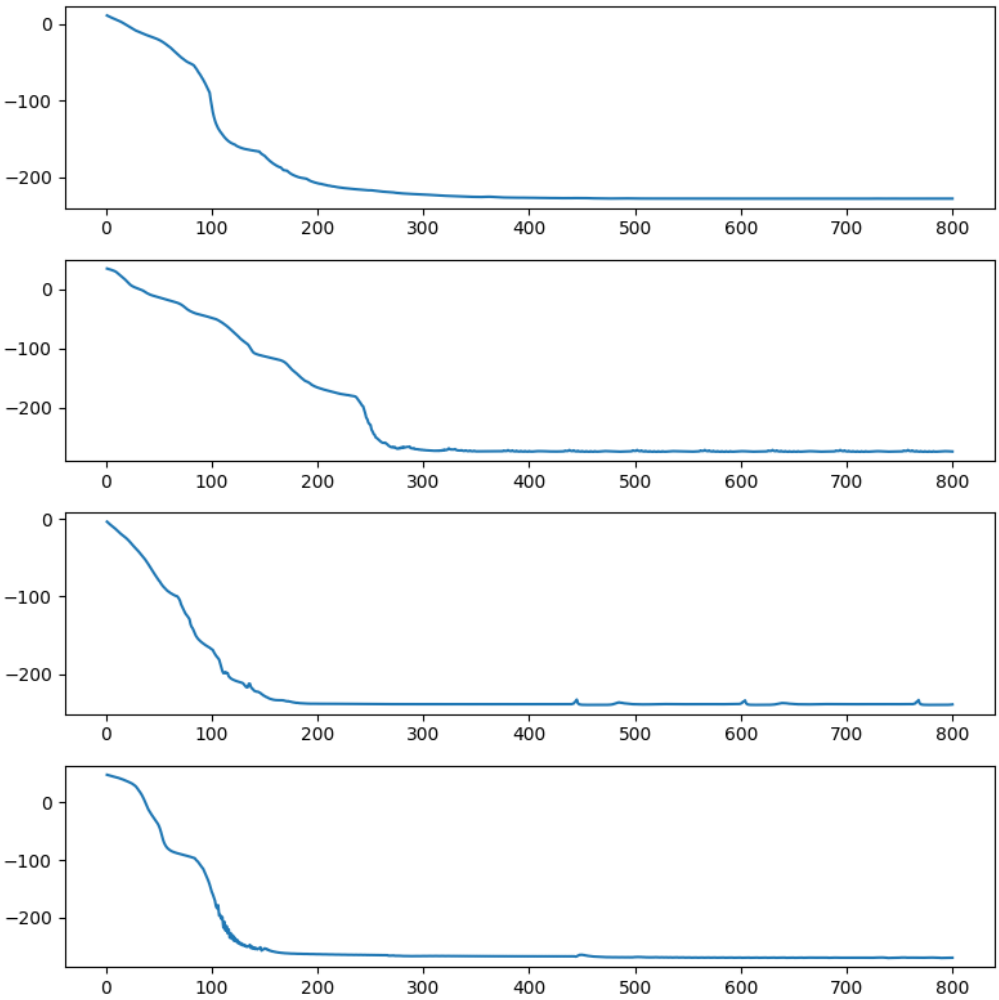
Σχήμα 4: Αποτελέσματα αύξησης ανάλυσης

το αντίστοιχο pixel της εικόνας χαμηλότερης ανάλυσης. Έτσι στο τέλος θα έχουμε μία εικόνα 7×7. Τα στοιχεία του πίνακα $\mathbf{T}$ θα είναι ως εξής. Η πρώτη γραμμή στις θέσεις 1 έως 4 θα έχει τιμή $\frac{1}{16}$ το ίδιο και για τις $1 \times 28 + (1$ έως $4)$, $2 \times 28 + (1$ έως $4)$, $3 \times 28 + (1$ έως $4)$ και στις υπόλοιπες 0. Έτσι πετυχαίνουμε να αθροίζουμε τα πρώτα 4 στοιχειά, μετά τα επόμενα 4 άλλα με offset 28 διότι πρέπει να μεταφερθούμε στην δεύτερη στήλη του grid κ.ο.κ. Ομοίως για την δεύτερη γραμμή άλλα αυτή την φορά θέτουμε $\frac{1}{16}$ τα στοιχεία $k \times 28 + (5$ έως $8)$, $k = 0, 1, 2, 3$ και μηδέν τα υπόλοιπα, με αντίστοιχό τρόπο συμπληρώνουμε και τις υπόλοιπες στήλες του πίνακα.

3.1.1 Αποτελέσματα

Για τον αλγόριθμο χρησιμοποιήθηκε βήμα $\mu = 0.01$ και παράγοντας κανονικοποίησης για τον Adam $\lambda = 0.01$. Εφαρμόζοντας τον ίδιο επαναληπτικό αλγόριθμο με πριν (5), λαμβάνουμε τα αποτελέσματα που φαίνονται στο Σχ. (4). Στην πρώτη στήλη διακρίνουμε τις ιδανικές εικόνες (διάνυσμα $\mathbf{X_i}$) στην δεύτερη στήλη τις εικόνες χαμηλής ανάλυσης 7×7 σε αναπαράσταση 28×28 το οποίο το πετυχένουμε φτιάχνοντας ένα 'τετράγωνο' διαστάσεων 4×4 με όλα τα 16 pixels να έχουν την ίδια τιμή με το pixel που αντικαθιστούν, τέλος στην τρίτη στήλη φαίνονται οι εικόνες αυξημένης ανάλυσης 28×28. Στο Σχ. (5) φαίνεται η πορεία του κόστους σε σχέση με τις επαναλήψεις ως που να συγκλίνει ο αλγόριθμος, για κάθε μία από τις τέσσερις εικόνες.

Ο κώδικας **Python** που χρησιμοποιήθηκε για το πρόβλημα αυτό παρατίθεται στο [Γ']



Σχήμα 5: Πορεία κόστους

Α΄ Κώδικας για το Πρώτο Πρόβλημα

```
import numpy as np
from scipy.io import loadmat
import matplotlib.pyplot as plt

data = loadmat('data21.mat')

#!Loading Weights And Biases From .mat File
A_1 = np.array(data['A_1'])
A_2 = np.array(data['A_2'])
B_1 = np.array(data['B_1'])
B_2 = np.array(data['B_2'])

def u(Z):
    #! Generator producing handwritten eights
    W1 = A_1 @ Z + B_1
    Z1 = np.maximum(0, W1)

    W2 = A_2 @ Z1  + B_2
    X = 1 / (1 + np.exp(W2))

    return X

def generate_eights(num):
    eights = []

    for i in range(num):
        Z = np.random.normal(0,1, size=(10,1))
        X = u(Z)
        eights.append(X.reshape(28,28).T)

    return eights

def main():
    eights = generate_eights(100)
    fig, axes = plt.subplots(10, 10, figsize=(8, 8))

    for i, ax in enumerate(axes.flat):
        ax.imshow(eights[i], cmap='gray')
        ax.axis('off')  # Turn off axis labels
    plt.show()
if __name__ == '__main__':
    main()
```

Β΄ Κώδικας για το Δεύτερο Πρόβλημα

```
import numpy as np
from scipy.io import loadmat
import matplotlib.pyplot as plt
from tqdm import tqdm
import subprocess
from matplotlib.colors import Normalize

np.random.seed(10)
data21 = loadmat('data21.mat')
data22 = loadmat('data22.mat')

#!Loading Weights And Biases From .mat File
A_1 = np.array(data21['A_1'])
A_2 = np.array(data21['A_2'])
B_1 = np.array(data21['B_1'])
B_2 = np.array(data21['B_2'])


#! Loading X_i, X_n
X_i = np.array(data22['X_i'])
X_n = np.array(data22['X_n'])


def u(Z):
    #! Generator producing handwritten eights
    W1 = A_1 @ Z + B_1
    Z1 = np.maximum(0, W1)

    W2 = A_2 @ Z1  + B_2
    X = 1 / (1 + np.exp(W2))

    return X

def plot(X):
    fig, axes = plt.subplots(2, 4, figsize=(8, 8))
    print(X_i.shape)
    X = np.concatenate((X_i, X), axis=1)
    for i, ax in enumerate(axes.flat):
        ax.imshow((X[:,i]).reshape(28,28).T, cmap='gray')
        ax.axis('off')  # Turn off axis labels
    plt.show()
```

```
def gradient(T,Z,N, distored_data, M):

    W1 = A_1 @ Z + B_1
    Z1 = np.maximum(0, W1)

    W2 = A_2 @ Z1  + B_2
    X = 1 / (1 + np.exp(W2))


    res = distored_data.reshape(N,1) - np.dot(T,X)
    norm_squared = np.linalg.norm(res) ** 2


    u2 = 2 * np.dot(-T.T, res) / norm_squared

    v2 = u2 * (X * (X - 1)) #? Sigmoid Rerivative Based On Output X

    u1 = A_2.T @ v2


    v1 = u1 * np.where(W1 > 0, 1, 0)

    u0 = A_1.T @ v1

    nablaJ_z = M * u0 + 2 * Z
    return nablaJ_z, norm_squared


def minimize(Z, n, N, iterations):

    I = np.eye(N)
    zeros = np.zeros((N, 784 - N))

    T = np.concatenate((I, zeros), axis=1) #! Matrix T

    distorted_data = X_n[:,n][:N]

    M = len(distorted_data)

    m = 0.01
    l = 0.001


    c = 10**(-8)
    nablaZ, norm_squared = gradient(T, Z,N, distorted_data, M)

    J = 0
    px =  (nablaZ) ** 2

    cost = []

    pbar = tqdm(range(iterations), colour='blue', position=0, desc=f"Restoring Image... Cost = {J}")

    for _ in pbar:
    #? ---------- Initiating Gradient Descent ----------
        J = M * np.log(norm_squared) + np.linalg.norm(Z)**2 #? Calculating Cost

        Z -= m * nablaZ / np.sqrt(px + c)

        pbar.set_description(f"Restoring Image... Cost = {J}")
        nablaZ, norm_squared = gradient(T, Z,N, distorted_data, M)

        px = (1 - l) * px + l * (nablaZ) ** 2
        cost.append(J)

    X = u(Z)
    padded = np.pad(distorted_data, (0, 784 - len(distorted_data)), 'constant').reshape(784,1)
    return X, padded, cost

def plot_images(original_images, distorted_images, recovered_images):
    fig, axes = plt.subplots(len(distorted_images), 3, figsize=(8, 8))

    for i in range(len(distorted_images)):
        axes[i, 0].imshow(original_images[:,i].reshape(28,28).T, cmap='gray')
        axes[i, 0].axis('off')

        norm = Normalize(vmin=0, vmax=np.max(distorted_images[i]))
        axes[i, 1].imshow(distorted_images[i].reshape(28,28).T, cmap='gray', norm =norm)
        axes[i, 1].axis('off')

        axes[i, 2].imshow(recovered_images[i].reshape(28,28).T, cmap='gray')
        axes[i, 2].axis('off')

    plt.tight_layout()
    plt.show()

def plot_costs(costs):
    fig, axes = plt.subplots(len(costs), 1, figsize=(8, 8))

    for i in range(len(costs)):
        axes[i].plot(np.linspace(1,len(costs[i]), len(costs[i])), costs[i])

    plt.tight_layout()
    plt.show()

def main():
```

```
    Z = np.random.normal(0,1, size=(10,1))

    iterations = 800
    N = 400
    # subprocess.run('clear')

    #! Running recovering algorithm for each image of X_n (keeping the first )
    X1, distorted_data1, cost1 = minimize(Z,0, N, iterations)
    X2, distorted_data2, cost2 = minimize(Z,1, N, iterations)
    X3, distorted_data3, cost3 = minimize(Z,2, N, iterations)
    X4, distorted_data4, cost4 = minimize(Z,3, N, iterations)

    distorted_images = [distorted_data1, distorted_data2, distorted_data3, distorted_data4]
    recovered_images = [X1, X2, X3, X4]
    plot_images(X_i, distorted_images, recovered_images)

    costs = [cost1, cost2, cost3, cost4]
    plot_costs(costs)

if __name__ == "__main__":
    main()
```

Γ′ Κώδικας για το Τρίτο Πρόβλημα

```
import numpy as np
from scipy.io import loadmat
import matplotlib.pyplot as plt
from tqdm import tqdm
import subprocess
from matplotlib.colors import Normalize

np.random.seed(30)
data21 = loadmat('data21.mat')
data22 = loadmat('data23.mat')

#!Loading Weights And Biases From .mat File
A_1 = np.array(data21['A_1'])
A_2 = np.array(data21['A_2'])
B_1 = np.array(data21['B_1'])
B_2 = np.array(data21['B_2'])

#! Loading X_i, X_n
X_i = np.array(data22['X_i'])
X_n = np.array(data22['X_n'])


def u(Z):
    #! Generator producing handwritten eights
    W1 = A_1 @ Z + B_1
    Z1 = np.maximum(0, W1)

    W2 = A_2 @ Z1  + B_2
    X = 1 / (1 + np.exp(W2))

    return X

def gradient(T,Z, distored_data, M):

    W1 = A_1 @ Z + B_1
    Z1 = np.maximum(0, W1)

    W2 = A_2 @ Z1  + B_2
    X = 1 / (1 + np.exp(W2))


    res = distored_data.reshape(49,1) - np.dot(T,X)
    norm_squared = np.linalg.norm(res) ** 2

    u2 = 2 * np.dot(-T.T, res) / norm_squared

    v2 = u2 * (X * (X - 1)) #? Sigmoid Rerivative Based On Output X

    u1 = A_2.T @ v2

    v1 = u1 * np.where(W1 > 0, 1, 0)

    u0 = A_1.T @ v1

    nablaJ_z = M * u0 + 2 * Z
    return nablaJ_z, norm_squared

def minimize(Z, n, iterations):
    high_res_size = 28  # High resolution image size
    low_res_size = 7    # Low resolution image size
    block_size = 4      # Block size in high-res corresponding to one low-res pixel

    rows = 49  # for a 7x7 low-resolution image
    cols = 784  # for a 28x28 high-resolution image
    T = np.zeros((49, 784)) #! Initializing Matrix

    for i in range(7):
        for j in range(7):
```

```
            r_i = 7 * i + j
            h = j * 4
            for bi in range(4):
                T[r_i,(i * 4 + bi) * 28 + h : (i * 4 + bi) * 28 + h + 4] = 1/ 16

    distorted_data = X_n[:,n]

    M = len(distorted_data)

    m = 0.01
    l = 0.01

    c = 10**(-8)

    nablaZ, norm_squared = gradient(T, Z, distorted_data, M)

    J = 0
    px =  (nablaZ) ** 2

    cost = []

    pbar = tqdm(range(iterations), colour='blue', position=0, desc=f"Restoring Image... Cost = {J}")

    for _ in pbar:
    #? ---------- Initiating Gradient Descent ----------*
        J = M * np.log(norm_squared) + np.linalg.norm(Z)**2 #? Calculating Cost

        Z -= m * nablaZ / np.sqrt(px + c)

        pbar.set_description(f"Restoring Image... Cost = {J}")
        nablaZ, norm_squared = gradient(T, Z, distorted_data, M)

        px = (1 - l) * px + l * (nablaZ) ** 2
        cost.append(J)

    X = u(Z)
    return X, distorted_data, cost


def plot_images(original_images, distorted_images, recovered_images):
    fig, axes = plt.subplots(len(distorted_images), 3, figsize=(8, 8))

    for i in range(len(distorted_images)):

        axes[i, 0].imshow(original_images[:,i].reshape(28,28).T, cmap='gray')
        axes[i, 0].axis('off')
        high_res_image = np.repeat(np.repeat(distorted_images[i].reshape(7,7), 4, axis=0), 4, axis=1)
        norm = Normalize(vmin=0, vmax=np.max(distorted_images[i]))
        axes[i, 1].imshow(high_res_image.reshape(28,28).T, cmap='gray', norm =norm)
        axes[i, 1].axis('off')

        axes[i, 2].imshow(recovered_images[i].reshape(28,28).T, cmap='gray')
        axes[i, 2].axis('off')

    plt.tight_layout()
    plt.savefig('upscaled.png')
    plt.show()

def plot_costs(costs):
    fig, axes = plt.subplots(len(costs), 1, figsize=(8, 8))

    for i in range(len(costs)):
        axes[i].plot(np.linspace(1,len(costs[i]), len(costs[i])), costs[i])

    plt.tight_layout()
    plt.savefig('costsUpscale.png')

    plt.show()

def main():
    Z = np.random.normal(0,1, size=(10,1))

    iterations = 800
    subprocess.run('clear')
    #! Running recovering algorithm for each image of X_n (keeping the first )
    X1, distorted_data1, cost1 = minimize(Z,0, iterations)
    X2, distorted_data2, cost2 = minimize(Z,1, iterations)
    X3, distorted_data3, cost3 = minimize(Z,2, iterations)
    X4, distorted_data4, cost4 = minimize(Z,3, iterations)

    distorted_images = [distorted_data1, distorted_data2, distorted_data3, distorted_data4]
    recovered_images = [X1, X2, X3, X4]
    plot_images(X_i, distorted_images, recovered_images)

    costs = [cost1, cost2, cost3, cost4]
    plot_costs(costs)

if __name__ == "__main__":
    main()
```