Θέματα Όρασης Υπολογιστών

Ακαδημαϊκό Έτος 2024-2025

Εργαστηριακή Άσκηση 5

Ανάλυση Κυρίων Συνιστωσών-PCA Autoencoders-AE
και Variational Autoencoders-VAE

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ

Ονοματεπώνυμο: Φωτάκης Ανδρέας
ΑΜ: 1084674

Μέρος 1
Μείωση Διαστατικότητας των Δεδομένων

Αποδείξεις

- Η λύση του προβλήματος για $L = 1$, προκύπτει από τη λύση του ακόλουθου, με περιορισμούς προβλήματος βελτιστοποίησης :

$$\max_{v_1} {v_1}^{\mathrm{T}} S v_1 \qquad subject\ to\ \| v_1 \|_2 = 1$$

όπου $\| z \|_2$, η $l2$ στάθμη του διανύσματος z.

Απόδειξη

$$X = \{x_{\mathrm{i}} \in \mathbb{R}^{\mathrm{n}}\}_0^M \ , X = N \times M\ matrix$$

$$y = v_1^T X$$

$$\mathrm{Var}(y) = \frac{1}{M}|y|^2 = \frac{1}{M}|v_1^T X|^2 = \frac{1}{M} v_1^T X X^T v_1$$

Μητρώο συνδιασπορών των δεδομένων: $S = \frac{1}{M} X X^T$

Επακόλουθος: $\mathrm{Var}(y) = v_1^T S v_1$

Για να βρεθεί η κύρια κατεύθυνση (principal direction), μεγιστοποιούμε τη διασπορά:

$$\max_{v_1} v_1^T S v_1 v1$$

Αφού το v_1 είναι διάνυσμα κατεύθυνσης, πρέπει να έχει μοναδιαίο μήκος:

$$\max_{v_1} v_1^T S v_1 \quad \text{subject to } |v_1|^2 = 1$$

- Η λύση του παραπάνω προβλήματος είναι το ιδιοδιάνυσμα v_1^* που αντιστοιχεί στην μέγιστη ιδιοτιμή (έστω λ_1) του μητρώου S.

Απόδειξη

Constrained optimization πρόβλημα που θα λυθεί με Lagrange multipliers:

$$\mathcal{L}(v_1, \lambda) = v_1^T S v_1 - \lambda(v_1^T v_1 - 1)$$

Υπολογίζοντας την παράγωγο:

$$\frac{\partial \mathcal{L}}{\partial v_1} = 0 \Rightarrow 2Sv_1 = 2\lambda v_1 \implies Sv_1 = \lambda v_1$$

Η εξίσωση που προκύπτει δείχνει ότι το v_1 πρέπει να είναι ιδιοδιάνυσμα του S και το λ η αντίστοιχη ιδιοτιμή.

Αφού το S είναι συμμετρικό και θετικά ορισμένο, όλες οι ιδιοτιμές είναι θετικές και πραγματικές.

Για να μεγιστοποιήσουμε το $v_1^T S v_1$, επιλέγουμε το v_1 ως το ιδιοδιάνυσμα που αντιστοιχεί στη μεγαλύτερη ιδιοτιμή λ_1.

- Η λύση του προβλήματος για $L = 2$, προκύπτει από τη λύση του ακόλουθου, με περιορισμούς προβλήματος βελτιστοποίησης :

$$\max_{v_2} {v_2}^{\mathrm{T}} S v_2 \quad subject\ to\ \| v_2 \|_2 = 1\ , \langle v_1^*, v_2 \rangle = 0$$

όπου $\| z \|_2$, η $l2$ στάθμη του διανύσματος z και $\langle\ ,\ \rangle$ ο τελεστής εσωτερικού γινομένου.

Απόδειξη

Αφού έχουμε βρει την πρώτη κύρια συνιστώσα $\boldsymbol{v_1}$, η PCA στοχεύει να βρει τη δεύτερη κύρια συνιστώσα $\boldsymbol{v_2}$ η οποία:

- Μεγιστοποιεί τη διασπορά των δεδομένων που προβάλλονται πάνω σε αυτήν.
- Είναι ορθογώνια προς την πρώτη κύρια συνιστώσα $\boldsymbol{v_1}$.

Η προβολή των δεδομένων πάνω στη δεύτερη συνιστώσα $\boldsymbol{v_2}$ είναι:

$$y_2 = v_2^T X.$$

Η διασπορά αυτής της προβολής είναι:

$$\text{Var}(y_2) = \frac{1}{M} v_2^T X X^T v_2 = v_2^T S v_2,$$

όπου $S = \frac{1}{M} X X^T$ είναι ο πίνακας συνδιασποράς.

Περιορισμός Ορθογωνιότητας: Για να εξασφαλίσουμε ότι η δεύτερη κύρια συνιστώσα είναι ορθογώνια προς την πρώτη, επιβάλλουμε τον περιορισμό:

$$\boldsymbol{v_2^T v_1} = 0.$$

Περιορισμός Κανονικοποίησης: Όπως και η πρώτη κύρια συνιστώσα, η $\boldsymbol{v_2}$ πρέπει να έχει μοναδιαίο μήκος:

$$\| \boldsymbol{v_2} \|^2 = 1$$

Συνδυάζοντας τη μεγιστοποίηση της διασποράς με τους περιορισμούς ορθογωνιότητας και κανονικοποίησης, το πρόβλημα βελτιστοποίησης διαμορφώνεται ως:

$$\max_{\boldsymbol{v_2}} \boldsymbol{v_2^T S\, v_2} \quad \text{subject to} \quad \| \boldsymbol{v_2} \|^2 = 1 \ \kappa\alpha\iota\ \boldsymbol{v_2^T v_1} = 0$$

- Η λύση του παραπάνω προβλήματος είναι το ιδιοδιάνυσμα v_2^* που αντιστοιχεί στην μέγιστη ιδιοτιμή (έστω λ_2) του μητρώου S.

Διατύπωση Λαγκρανζιανής: Χρησιμοποιούμε τη μέθοδο των πολλαπλασιαστών Lagrange και ορίζουμε τη Λαγκρανζιανή ως:

$$\mathcal{L}(\boldsymbol{v_2}, \lambda, \mu) = \boldsymbol{v_2^T S v_2} - \lambda(|\boldsymbol{v_2}|^2 - 1) - \mu\left(\boldsymbol{v_2^T v_1}\right)$$

Πρώτες Συνθήκες Τάξης: Υπολογίζουμε τη μερική παράγωγο ως προς v_2 και τη μηδενίζουμε:

$$\frac{\partial \mathcal{L}}{\partial v_2} = 2Sv_2 - 2\lambda v_2 - \mu v_1 = 0$$

Αναδιατάσσοντας, προκύπτει:

$$Sv_2 = \lambda v_2 + \frac{\mu}{2} v_1$$

Επειδή v_1 είναι το ιδιοδιάνυσμα που αντιστοιχεί στη μέγιστη ιδιοτιμή λ_1, ο περιορισμός $v_2^T v_1 = 0$ εξασφαλίζει ότι το v_2 ανήκει στον ορθογώνιο συμπληρωματικό χώρο του v_1.

Συνεπώς, η λύση $\boldsymbol{v_2}$ πρέπει να είναι το ιδιοδιάνυσμα του S που αντιστοιχεί στη δεύτερη μεγαλύτερη ιδιοτιμή λ_2. Οι περιορισμοί διασφαλίζουν ότι το δεύτερο ιδιοδιάνυσμα είναι ορθογώνιο προς το πρώτο.

- Διατύπωση του τελικού $L = L$ προβλήματος βελτιστοποίησης.

Βάση της επαγωγής: $L = 1$

Για $L = 1$, έχουμε ήδη δείξει ότι το πρόβλημα βελτιστοποίησης είναι:

$$\max_{v_1} {v_1}^{\mathrm{T}} S v_1 \qquad subject\ to\ \| v_1 \|_2 = 1$$

Η λύση είναι το διάνυσμα v_1, το οποίο είναι το ιδιοδιάνυσμα του S που αντιστοιχεί στη μέγιστη ιδιοτιμή λ_1.

Επαγωγική υπόθεση:

Έστω ότι για $L = k$, έχουμε υπολογίσει τα k κύρια διανύσματα $v_1, v_2, \dots, v_k$, τα οποία είναι ορθοκανονικά και αντιστοιχούν στις k μεγαλύτερες ιδιοτιμές $\lambda_1, \lambda_2, \dots, \lambda_k$ του S.

Βήμα της επαγωγής: $Από\ L = k\ στο\ L = k + 1$

Για $L = k + 1$, αναζητούμε το επόμενο διάνυσμα v_{k+1} που:

1. Μεγιστοποιεί τη διασπορά $v_{k+1}^T S v_{k+1}$.
2. Είναι ορθογώνιο σε όλα τα προηγούμενα διανύσματα:

$$v_{k+1}^T v_i = 0, \quad \forall i = 1,2, \dots, k$$

3. Έχει μοναδιαίο μήκος: $\| \boldsymbol{v_{k+1}} \|^2 = 1$.

Το πρόβλημα βελτιστοποίησης για το $\kappa + 1$-οστό διάνυσμα είναι:

$$\max_{v_{k+1}} v_{k+1}^T S\, v_{k+1} \quad subject\ to: \| \boldsymbol{v_{k+1}} \|^2 = 1$$
$$and\ \boldsymbol{v_{k+1}^T v_i} = 0, \quad \forall i = 1,2, \dots, k$$

Η λύση είναι το ιδιοδιάνυσμα του S που αντιστοιχεί στην $k + 1$-οστή μεγαλύτερη ιδιοτιμή λ_{k+1}.

Γενική Διατύπωση για $L = L$

Για οποιοδήποτε L, το συνολικό πρόβλημα βελτιστοποίησης μπορεί να διατυπωθεί ως εξής:

$$\max_{\{v_1, v_2, \ldots, v_L\}} \sum_{i=1}^{L} \boldsymbol{v_i^T S\, v_i} \;\; subject\ to: \| \boldsymbol{v_i} \|^2 = 1$$
$$and\ \boldsymbol{v_i^T v_j} = 0, \quad \forall i \neq j$$

Οι περιορισμοί $v_i^T v_j = 0$ διασφαλίζουν ότι τα κύρια διανύσματα είναι ορθογώνια, γεγονός που εξασφαλίζει τη γραμμική ασυσχέτιστη διασπορά στις διαφορετικές διαστάσεις.

Η αντικειμενική συνάρτηση $\sum_{i=1}^{L} v_i^T S\, v_i$ διασφαλίζει ότι η διασπορά που εξηγείται από τον L-διάστατο υπόχωρο είναι η μέγιστη δυνατή.

Λόγω της ιδιοτιμής λ που συνδέεται με κάθε ιδιοδιάνυσμα v, το πρόβλημα οδηγεί στην επιλογή των L πρώτων ιδιοδιανυσμάτων του S, τα οποία αντιστοιχούν στις L μεγαλύτερες ιδιοτιμές.

- Το πρόβλημα ελαχιστοποίησης:

$$\min_{A} |S - A|_F^2 \quad \text{subject to } \operatorname{rank}(A) = L$$

έχει λύση: $A^* = V_L \Sigma_L V_L^T$

όπου:

- V_L περιέχει τις πρώτες L στήλες του ορθοκανονικού πίνακα ιδιοδιανυσμάτων V,
- Σ_L είναι το $L \times L$ διαγώνιο μητρώο με τις L πρώτες τιμές της διαγωνίου του μητρώου Σ_0

Απόδειξη

SVD του S: Έστω ότι S έχει $S = U\, \Sigma_0\, V^T$ με U, V ορθοκανονικούς και $\Sigma_0 = \operatorname{diag}(\sigma_1, \ldots, \sigma_r)$, όπου $r = rank(S)$

Αν S είναι συμμετρικός, τότε συνήθως γράφουμε $\mathrm{S} = \mathrm{V}\Sigma_0\, V^T$ (ιδιοδιάσπαση), αλλά η ουσία δεν αλλάζει.
Η νόρμα Frobenius είναι **αναλλοίωτη** (invariant) κάτω από ορθοκανονικούς μετασχηματισμούς. Συγκεκριμένα,

$$|S - A|_F = |U^T (S - A) V|_F,$$

αφού $|Q_1\, M\, Q_2|_F = |M|_F$ για ορθοκανονικούς Q_1, Q_2.

Για οποιονδήποτε πίνακα A με rank(A)=L, θέτουμε $B = U^T A V$.

Τότε $rank\ (B) = rank\ (A) = L$. Άρα

$$|S - A|_F^2 = |U \Sigma_0 V^T - A|_F^2 = |U^T (U \Sigma_0 V^T - A) V|_F^2 = |\Sigma_0 - B|_F^2.$$

Έτσι, το αρχικό πρόβλημα ισοδυναμεί με:

$$\min_{rank(B)=L} |\Sigma_0 - B|_F^2 \ .$$

Η Σ_0 είναι **διαγώνιος** πίνακας $diag(\sigma 1, \dots, \sigma r, 0, \dots, 0)$.
Ο πίνακας B μπορεί να είναι οποιοσδήποτε (περιορισμός: rank(B)=L), αλλά για να προσεγγίσει βέλτιστα μια διαγώνια μορφή, θα επιδιώξει να είναι κι εκείνος διαγώνιος.

Στην πράξη, μπορούμε να αποδείξουμε ότι τα στοιχεία εκτός διαγωνίου απλώς αυξάνουν τη $|\Sigma_0 - B|_F^2$ χωρίς να βελτιώνουν κάτι, οπότε η βέλτιστη λύση είναι διαγώνια.

Αν υποθέσουμε $B = diag(b1, \dots, bn)$, τότε $rank\ (B) = L$ σημαίνει ότι **τουλάχιστον n–L από τα bi είναι μηδενικά** (μόνο L μπορούν να είναι μη μηδενικά). Θέλουμε να ελαχιστοποιήσουμε:

$$|\Sigma_0 - B|_F^2 = \sum_{i=1}^{n} (\sigma_i - b_i)^2$$

για να ελαχιστοποιήσουμε το $\sum(\sigma_i - b_i)^2$, επιλέγουμε:

$$b_i = \begin{cases} \sigma_{i,} & \mathrm{i} = 1, \dots, \mathrm{L} \\ 0, & \mathrm{i} = \mathrm{L}+1, \dots, \mathrm{n} \end{cases}$$

Με αυτόν τον τρόπο, τα bi αντιγράφουν τις **L** μεγαλύτερες $\sigma_{i,}$, και μηδενίζουν τις υπόλοιπες. Αυτό δίνει rank L και ελάχιστο σφάλμα.

Επιστρέφοντας στον πίνακα A, η παραπάνω επιλογή $B = diag(\sigma 1, \dots, \sigma L, 0, \dots)$ αντιστοιχεί στον πίνακα:

$$A^* = U B V^T = U \begin{pmatrix} \sigma_1 & & 0 \\ & \ddots & \\ 0 & & \sigma_L \\ & & 0 \end{pmatrix} V^T.$$

Αν S είναι συμμετρικός και γράψουμε $\mathrm{S} = \mathrm{V}\Sigma_0 V^T$, τότε $U = V$ και παίρνουμε:

$$A^* = V_L \Sigma_L V_L^T,$$

όπου V_L είναι οι πρώτες L στήλες του V, Σ_L είναι η διαγώνιος με $\sigma 1, \dots, \sigma L$.

Διαδικασία PCA

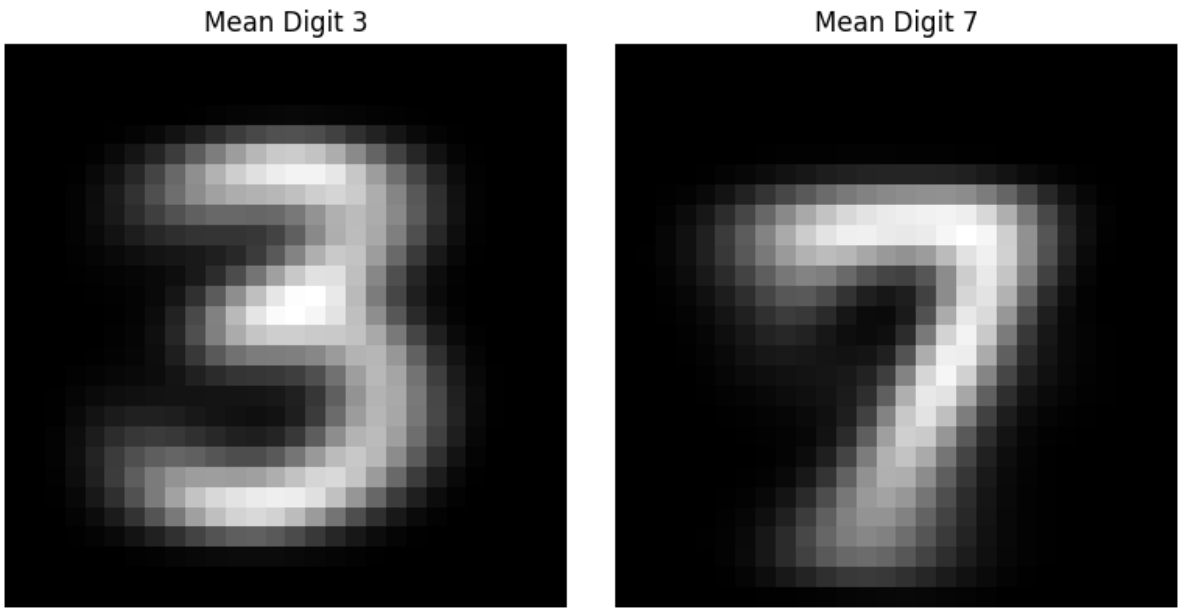
Ερώτημα 1

Επιλέγουμε μόνο δύο ψηφία, τα 3 και τα 7. Χρησιμοποιούμε το np.isin(labels, digits) για να φτιάξουμε μια μάσκα (boolean πίνακα) που είναι True μόνο στις θέσεις όπου η ετικέτα είναι 3 ή 7. Στη συνέχεια, εφαρμόζουμε τη μάσκα τόσο στα δεδομένα (digit_data) όσο και στις ετικέτες (digit_labels), κρατώντας έτσι μόνο τις εικόνες και τις ετικέτες των ψηφίων που μας ενδιαφέρουν.

```
digits = [3, 7]
mask = np.isin(labels, digits)
digit_data = data[mask]
digit_labels = labels[mask]
```

Υπολογίζουμε το μέσο ψηφίο για κάθε ένα από τα επιλεγμένα ψηφία και την μέση εικόνα για όλο το μέρος του dataset που κρατήσαμε.

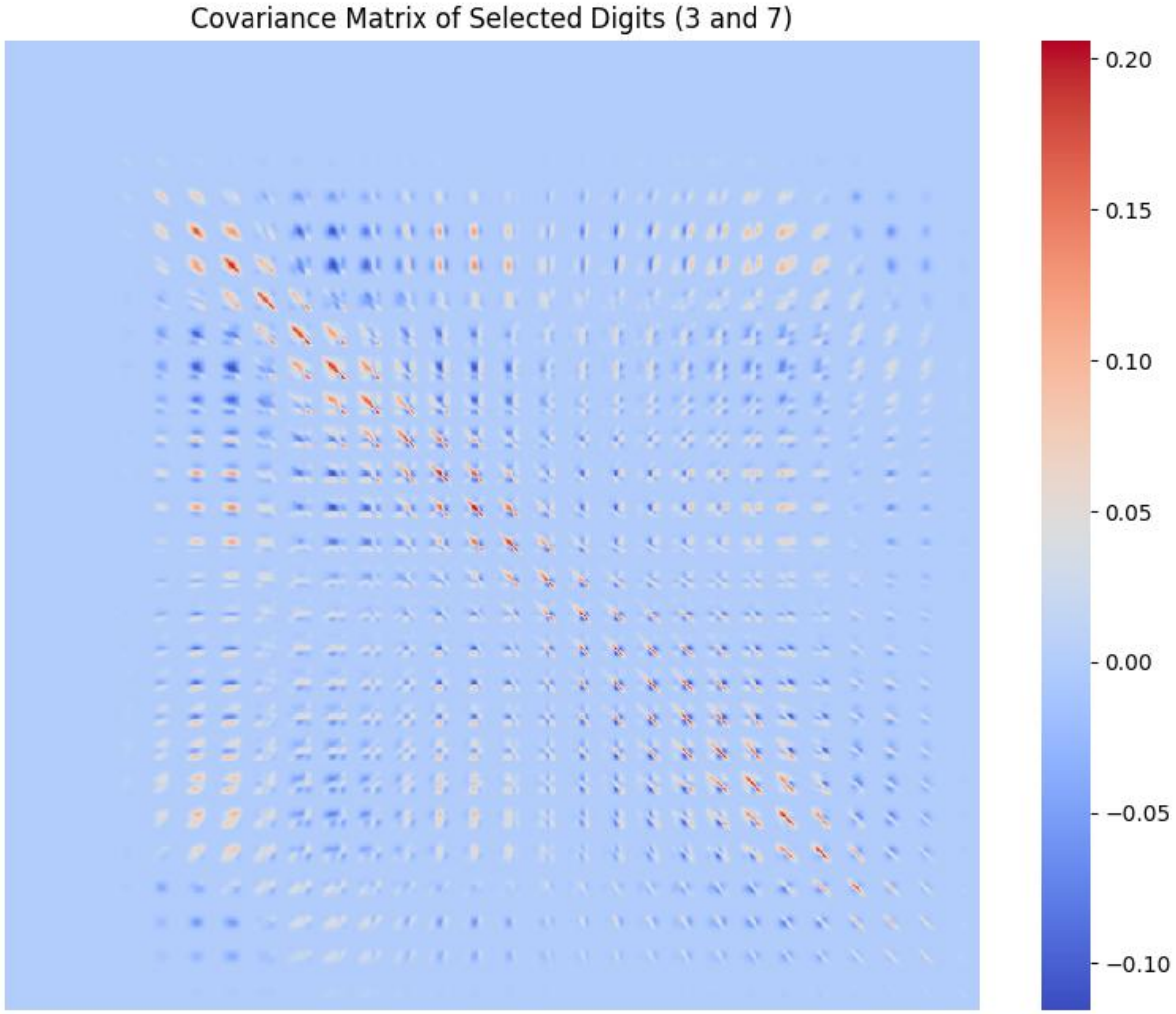
```
mean_images_per_digit = {digit: np.mean(digit_data[digit_labels ==
digit], axis=0) for digit in digits}
mean_image = np.mean(digit_data, axis=0)
```



Υπολογίζουμε την συνδιασπορά μεταξύ των δυο αυτών ψηφίων.

```
mean_centered = digit_data - mean_image
cov_matrix = np.cov(mean_centered, rowvar=False)
```

Αρχικά, αφαιρούμε τη συνολική μέση εικόνα (mean_image) από κάθε δείγμα (mean centering). Έπειτα, υπολογίζουμε τον πίνακα συνδιασποράς (covariance matrix) με τη συνάρτηση np.cov(). Το rowvar=False δηλώνει ότι οι μεταβλητές βρίσκονται στις στήλες (κάθε pixel είναι μία "μεταβλητή"), ενώ κάθε γραμμή είναι ένα δείγμα εικόνας.



Ορίζουμε συνάρτηση που υλοποιεί PCA μέσω SVD και στη συνέχεια ανακατασκευάζει:

```
def PCA(data, mean_image, components_list):
    mean_centered = data - mean_image
    U, Sigma, Vt = np.linalg.svd(mean_centered, full_matrices=False)
    reconstructions = {}
    mse_list = []

    for L in components_list:
        U_L = U[:, :L]
        Sigma_L = np.diag(Sigma[:L])
        Vt_L = Vt[:L, :]
        A_star = U_L @ Sigma_L @ Vt_L

        # Reconstruct the original data using A_star and adding the
mean back
        reconstructed = A_star + mean_image
        reconstructions[L] = reconstructed

        # Compute MSE
        mse = np.mean((data - reconstructed) ** 2)
        mse_list.append(mse)

    return reconstructions, Vt, mse_list
```

Υπολογισμός SVD: `U, Sigma, Vt = np.linalg.svd(...).`,

,όπου

- U περιέχει τα αριστερά ιδιοδιανύσματα (ποσοστά παραλλαγής ανά δείγμα),
- Sigma περιέχει τις ιδιοτιμές (ως μοναδιαίο διάνυσμα),
- Vt είναι οι δεξιοί ιδιοδιανύσματα (οι κύριες συνιστώσες - principal components).

Για κάθε αριθμό συνιστωσών **L** στη λίστα components_list, κρατάμε τα πρώτα **L** στοιχεία (U_L, Sigma_L, Vt_L). Στη συνέχεια, κάνουμε μερική ανακατασκευή της μήτρας (A_star) και προσθέτουμε πίσω τη μέση εικόνα.

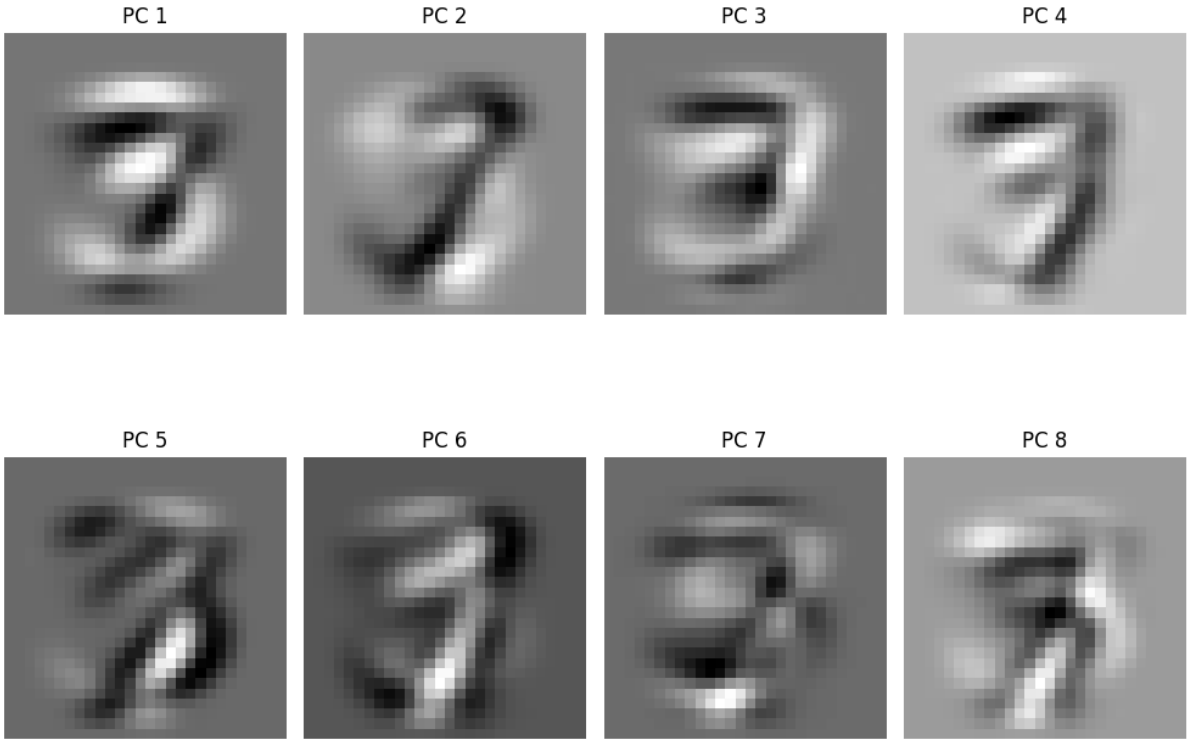
Εφαρμόζουμε PCA στο υποσύνολο του MNIST που κρατήσαμε:

```
# Apply PCA on the full dataset
components_list = [1, 8, 16, 64, 256]
reconstructions, Vt, mse = PCA(digit_data, mean_image, components_list)
```

Η συνάρτηση επιστρέφει όλες τις πιθανές ανακατασκευές για τα συγκεκριμένα L, τον πίνακα Vt (κύριες συνιστώσες) και τα αντίστοιχα σφάλματα MSE.

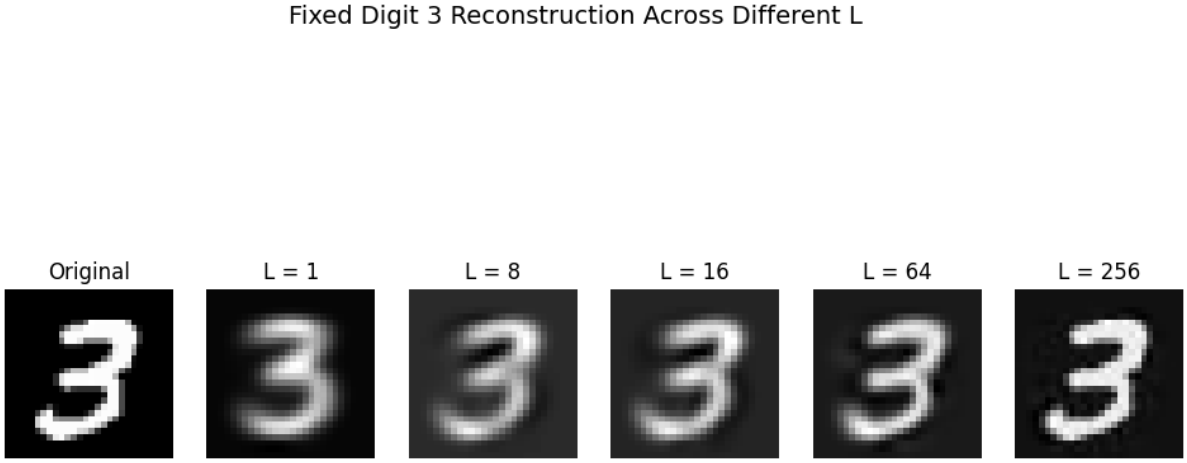
Μέσω του μητρώου V που προκύπτει από το SVD τυπώνουμε τις 8 πρώτες κύριες συνιστώσες των ψηφίων που επιλέχθηκαν:

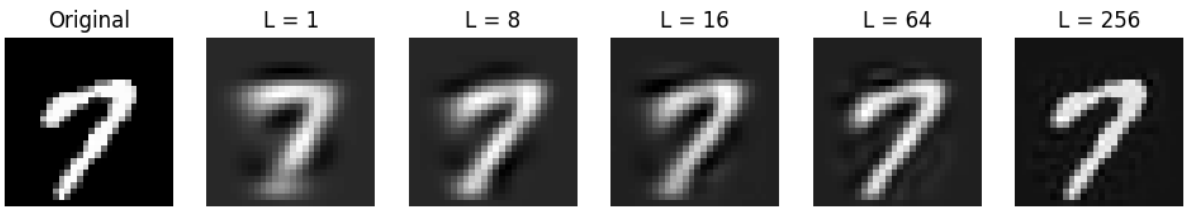
```
def plot_pca_components(Vt)
```



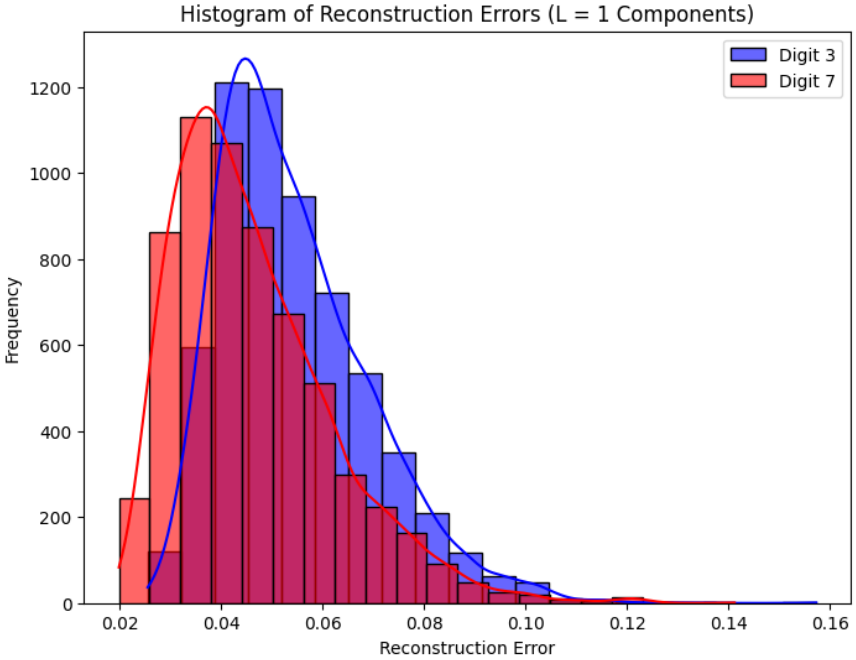
Και οι ανακατασκευές των ψηφίων αυτών για διαφορετικές τιμές του L:

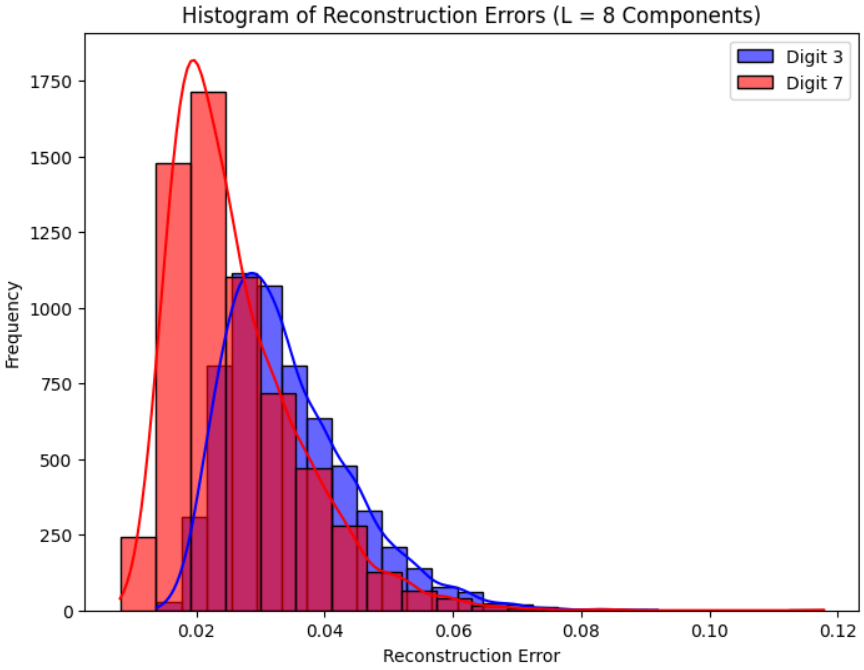
```
plot_fixed_reconstructions(digit_samples, reconstructions,
components_list)
```

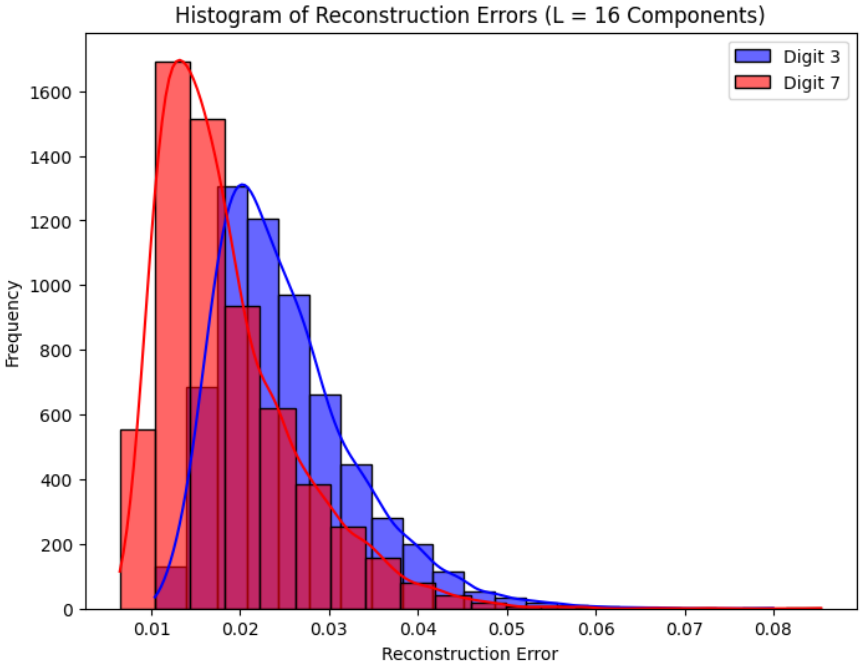


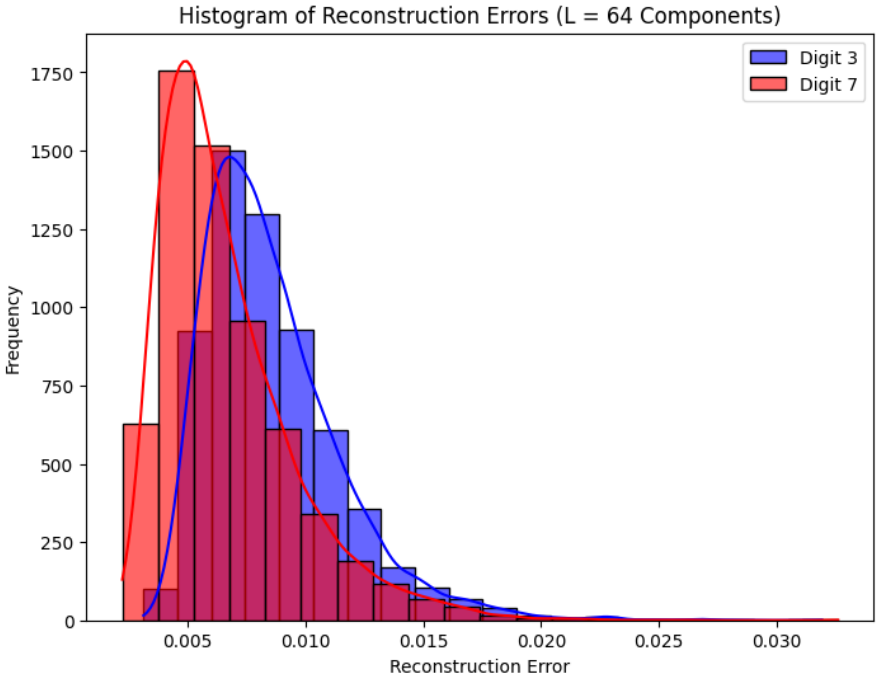


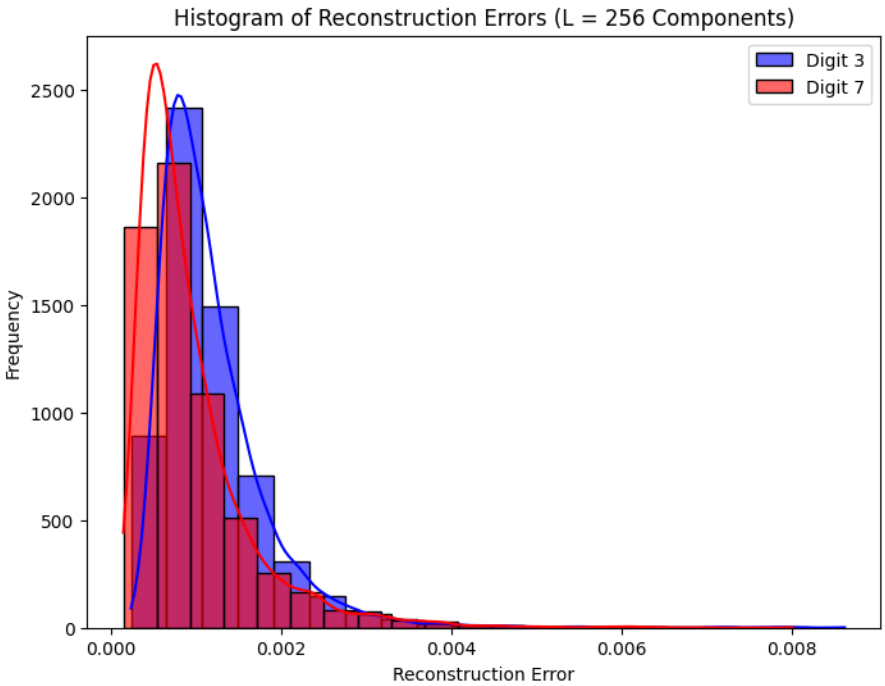
Σχεδιάζονται τα ιστογράμματα των σφαλμάτων καθώς και το σφάλμα ανακατασκευής για κάθε ένα ψηφίο από αυτά που επιλέχθηκαν.

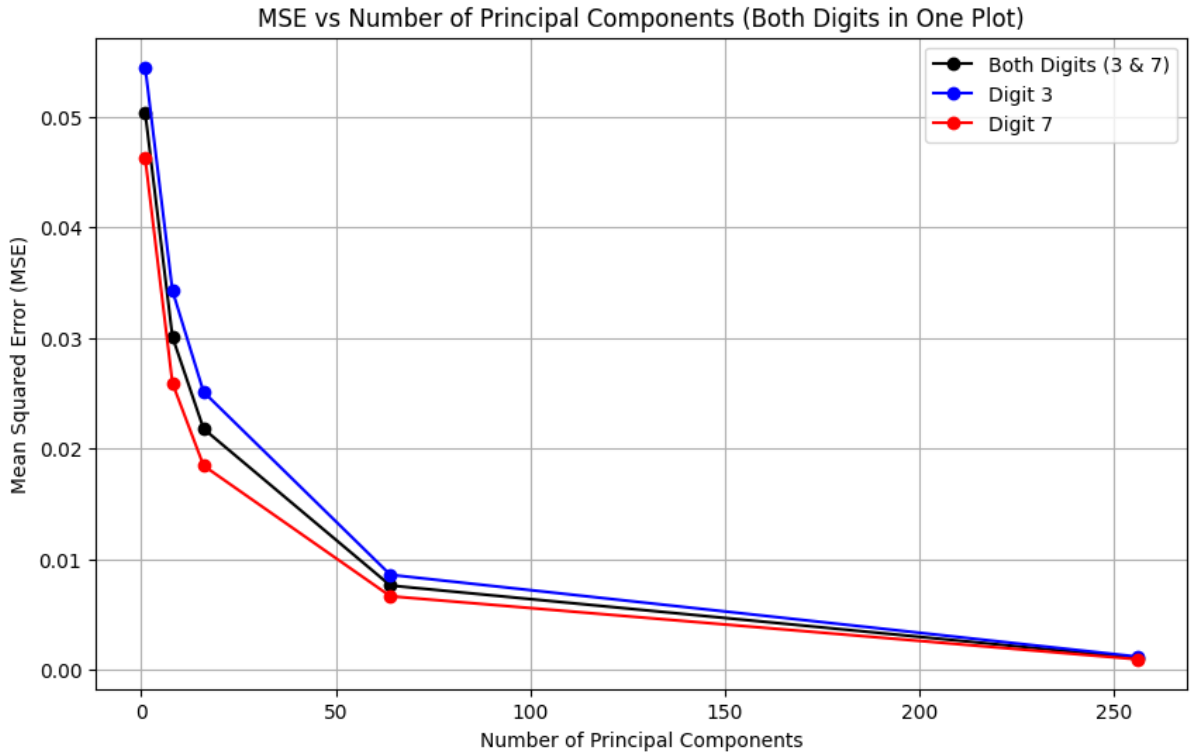



Histogram of Reconstruction Errors (L = 8 Components)
Digit 3
Digit 7
Frequency
Reconstruction Error
0
250
500
750
1000
1250
1500
1750
0.02
0.04
0.06
0.08
0.10
0.12


Histogram of Reconstruction Errors (L = 16 Components)
Digit 3
Digit 7
Frequency
Reconstruction Error
0
200
400
600
800
1000
1200
1400
1600
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08


Histogram of Reconstruction Errors (L = 64 Components)
Digit 3
Digit 7
Frequency
Reconstruction Error
0
250
500
750
1000
1250
1500
1750
0.005
0.010
0.015
0.020
0.025
0.030


Histogram of Reconstruction Errors (L = 256 Components)
Digit 3
Digit 7
Frequency
Reconstruction Error
0
500
1000
1500
2000
2500
0.000
0.002
0.004
0.006
0.008



	3	7
1	0.054449	0.046220
8	0.034271	0.025898
16	0.025106	0.018443
64	0.008580	0.006642
256	0.001180	0.000940

Επιβεβαιώνεται μέσω των γραφικών αυτών η οπτική παρατήρηση της βελτίωσης της λεπτομέρειας της εικόνας καθώς αυξάνεται το L.

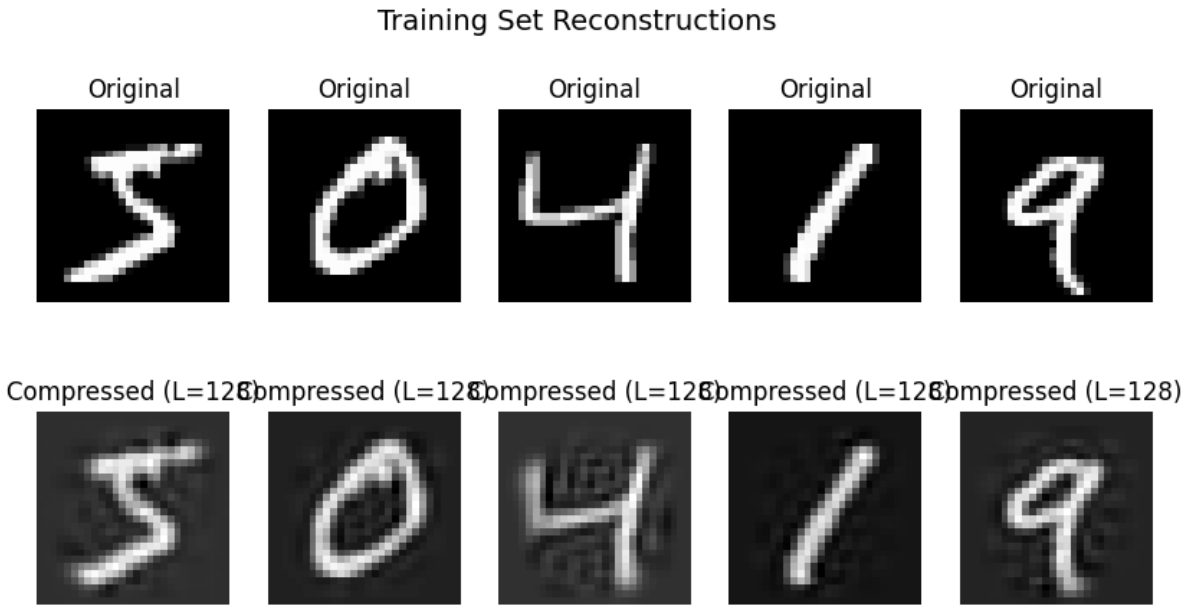
Ερώτημα 2

Εφαρμογή PCA και υπολογισμός του μητρώου VL με L =128

```
# PCA using SVD (Compute V_L for L=128)
L = 128  # Number of principal components
mean_centered_train = train_data - mean_image  # Centering training
data
U, Sigma, Vt = np.linalg.svd(mean_centered_train, full_matrices=False)

V_L = Vt[:L, :]  # Select first 128 principal components (V_L)
```

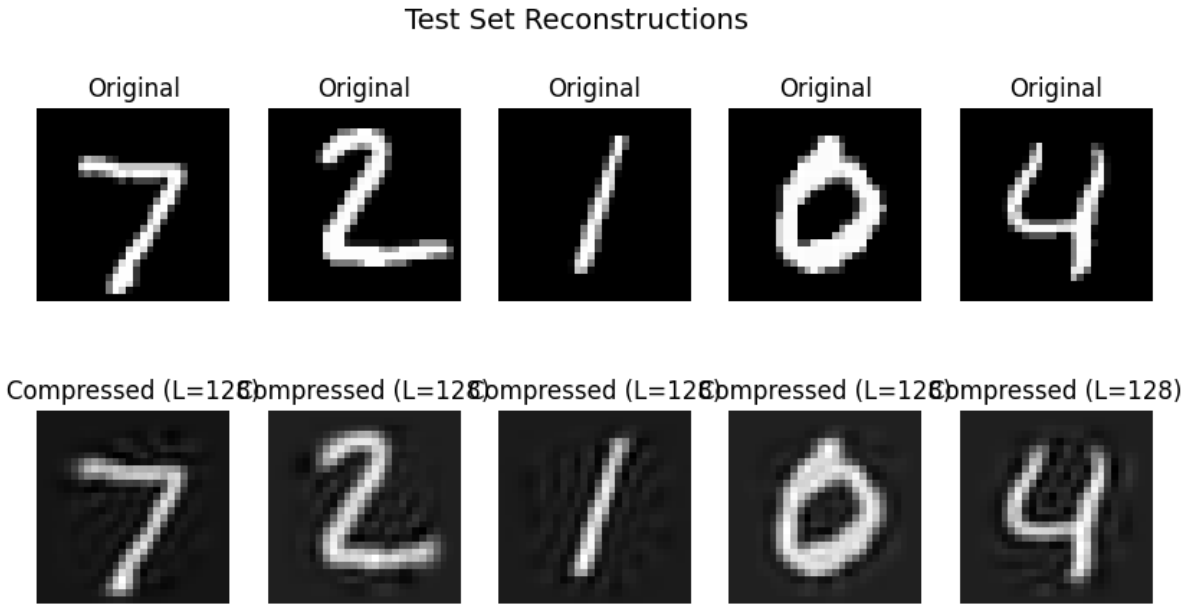
Αρχικά ανακατασκευάζουμε ψηφία από το training set.

```
compressed_train = mean_centered_train @ V_L.T  # Project training data
onto PCA space
reconstructed_train = (compressed_train @ V_L) + mean_image  #
Reconstruct
```



Και στην συνέχεια ψηφία από το testing set.

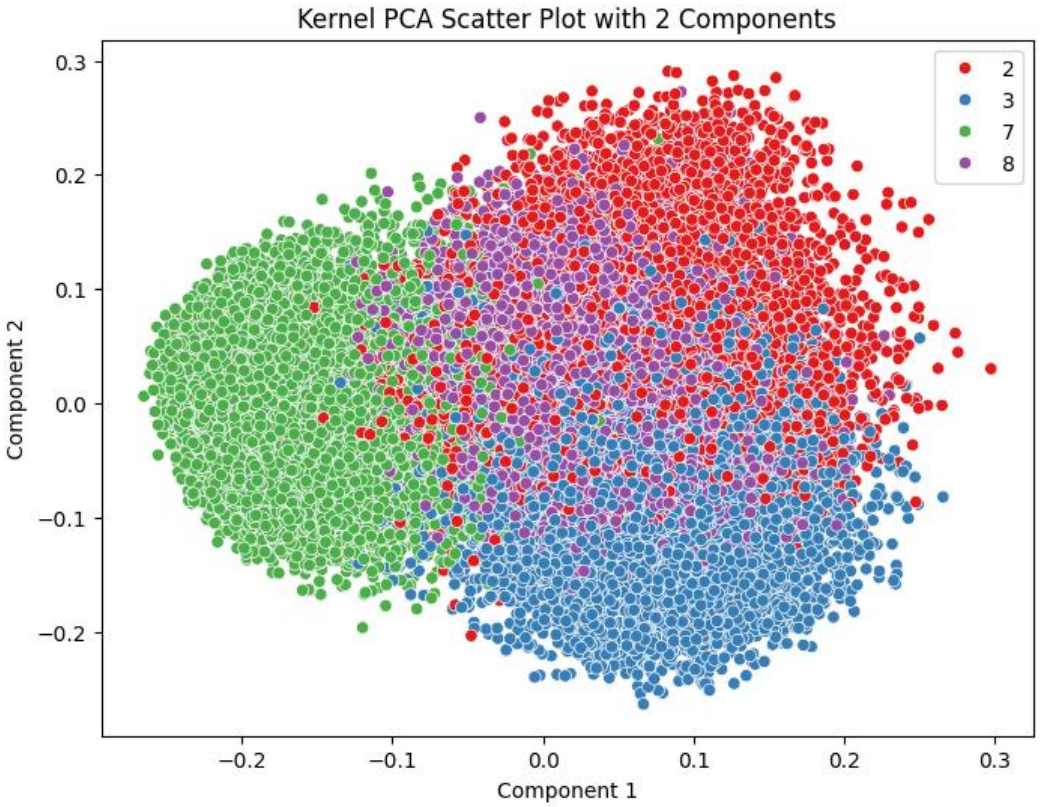
```
mean_centered_test = test_data - mean_image  # Centering test data
using training mean
compressed_test = mean_centered_test @ V_L.T  # Project test data onto
PCA space
reconstructed_test = (compressed_test @ V_L) + mean_image  #
Reconstruct test images
```



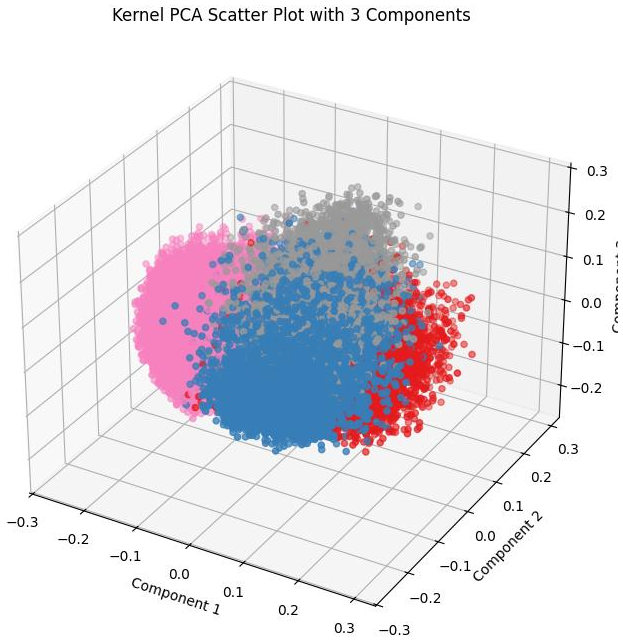
Διαδικασία kernel-PCA

Για την υλοποίηση αυτού του ερωτήματος έχουν επιλεγεί τα ψηφία 2,3,7,8.

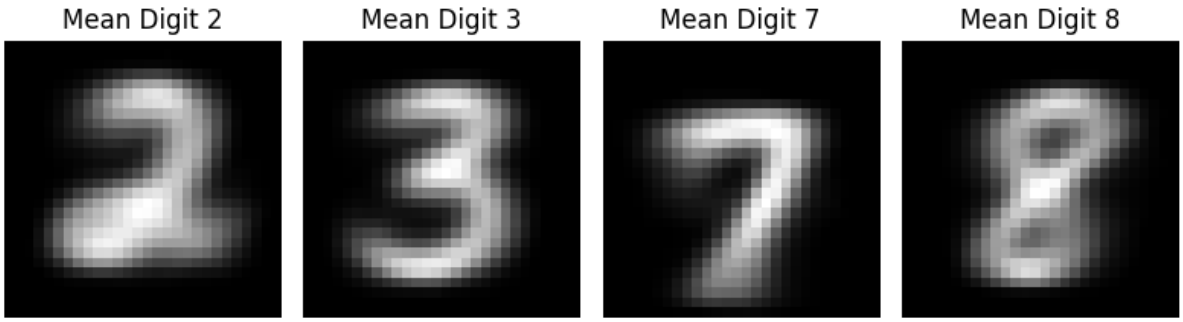
Αναπαράσταση των ψηφίων στον δυσδιάστατο χώρο κρατώντας τα 2 πρώτα components:



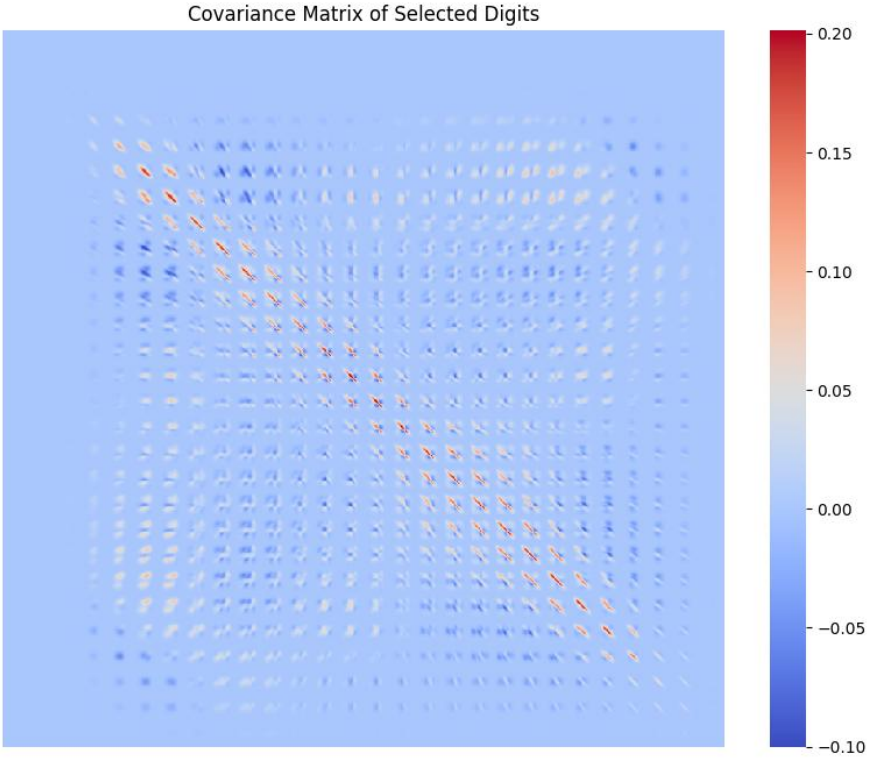
Αναπαράσταση των ψηφίων στον τρισδιάστατο χώρο κρατώντας τα 3 πρώτα components:



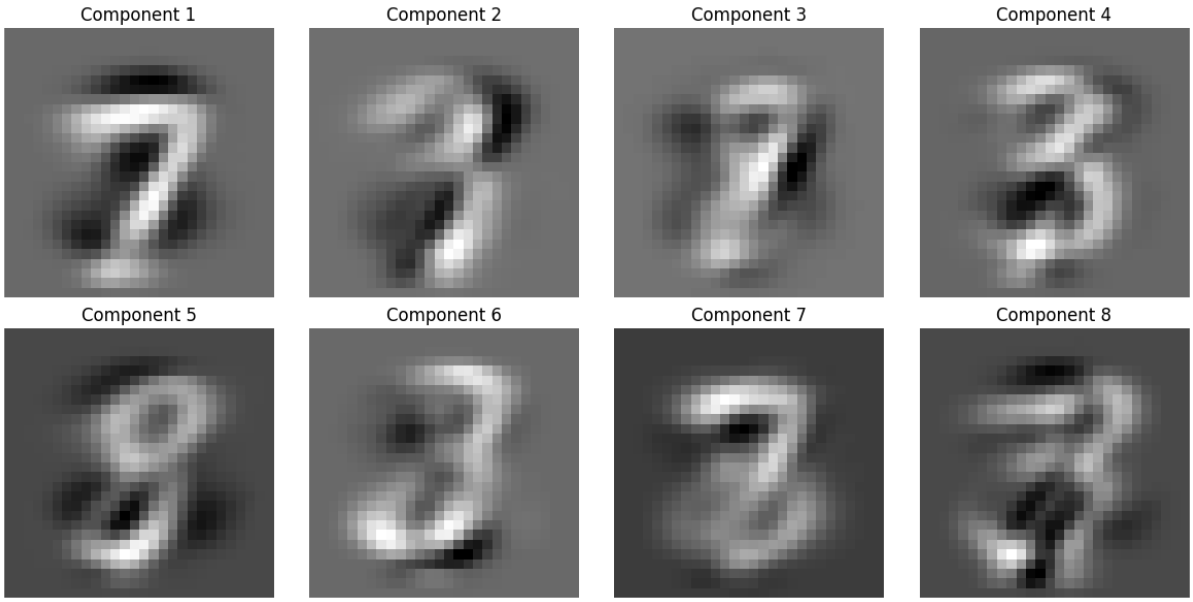
Το μέσο ψηφίο



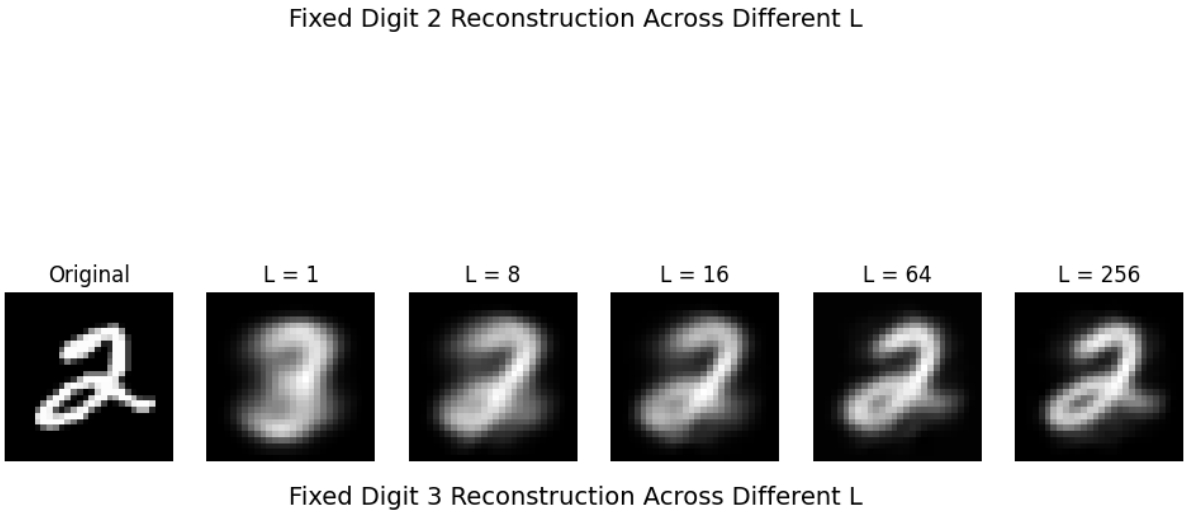
Το μητρώο συνδιασπορών

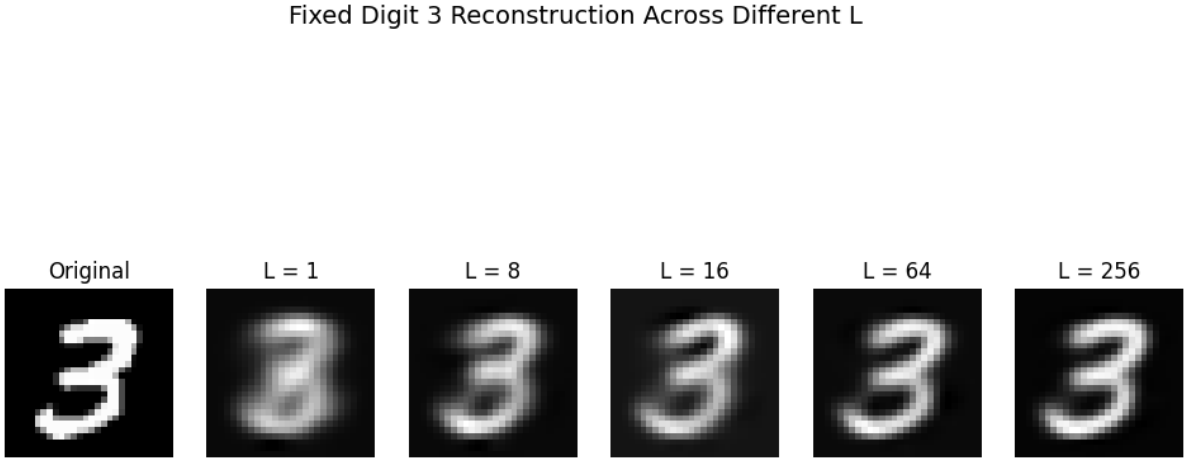


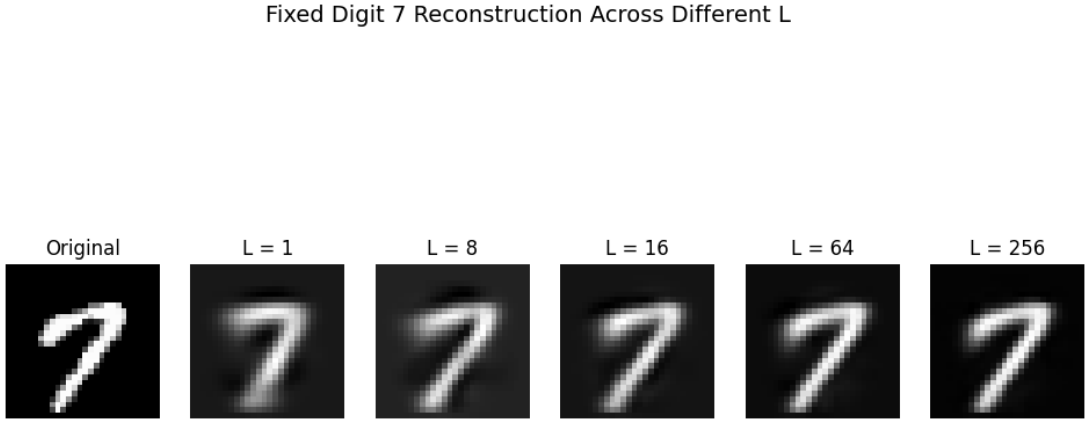
Οι οκτώ (8) πρώτες κύριες συνιστώσες

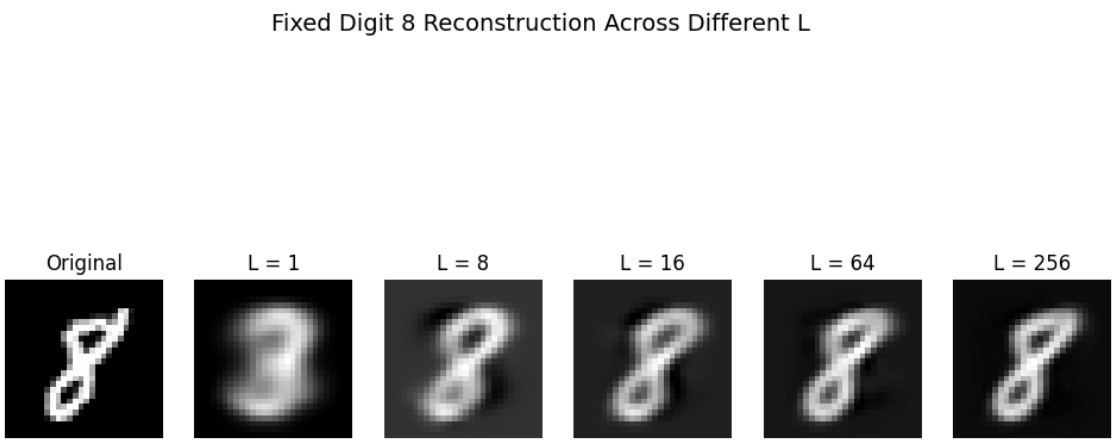


Οι ανακατασκευές του κάθε ψηφίου για L = 1, 8, 16, 64 και 256 αντίστοιχα.

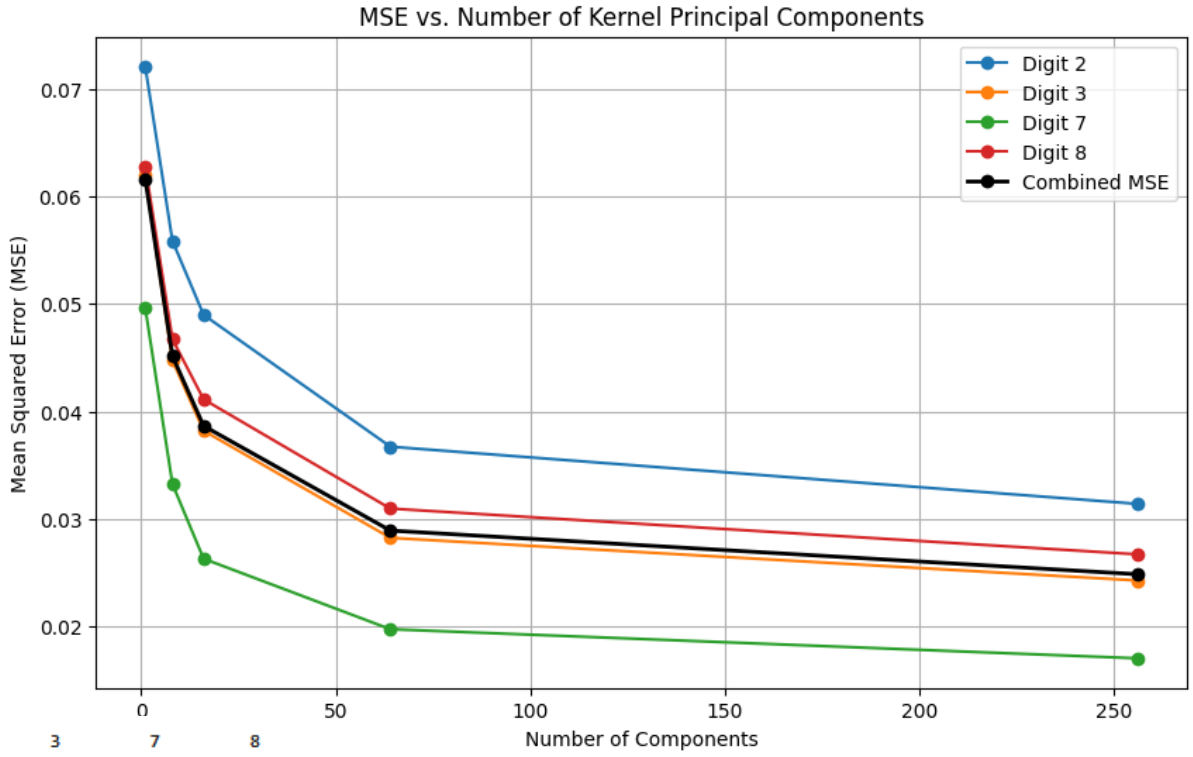




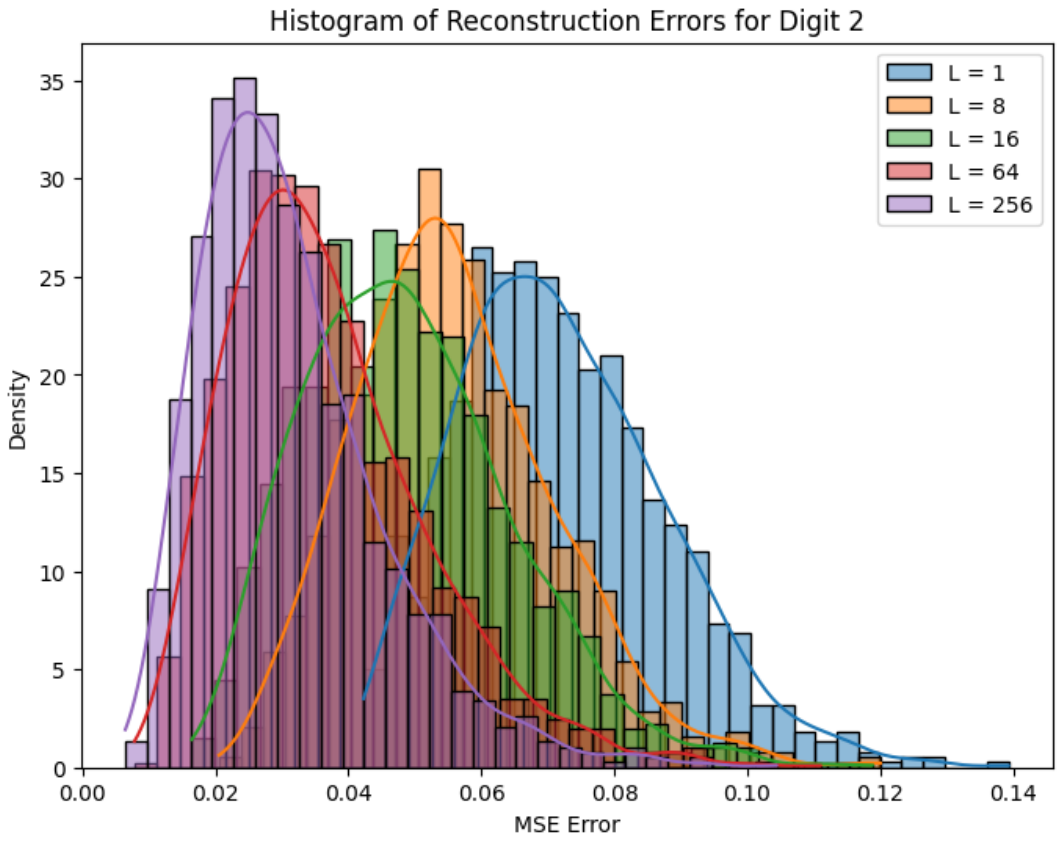


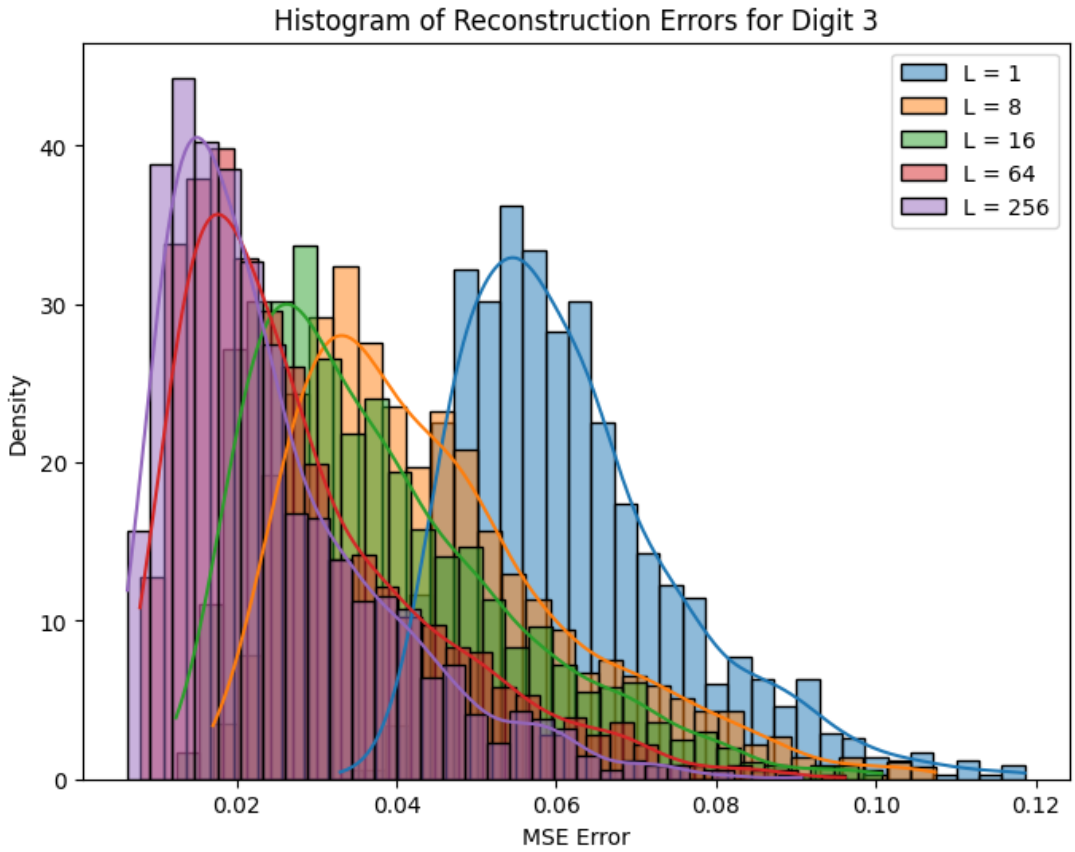


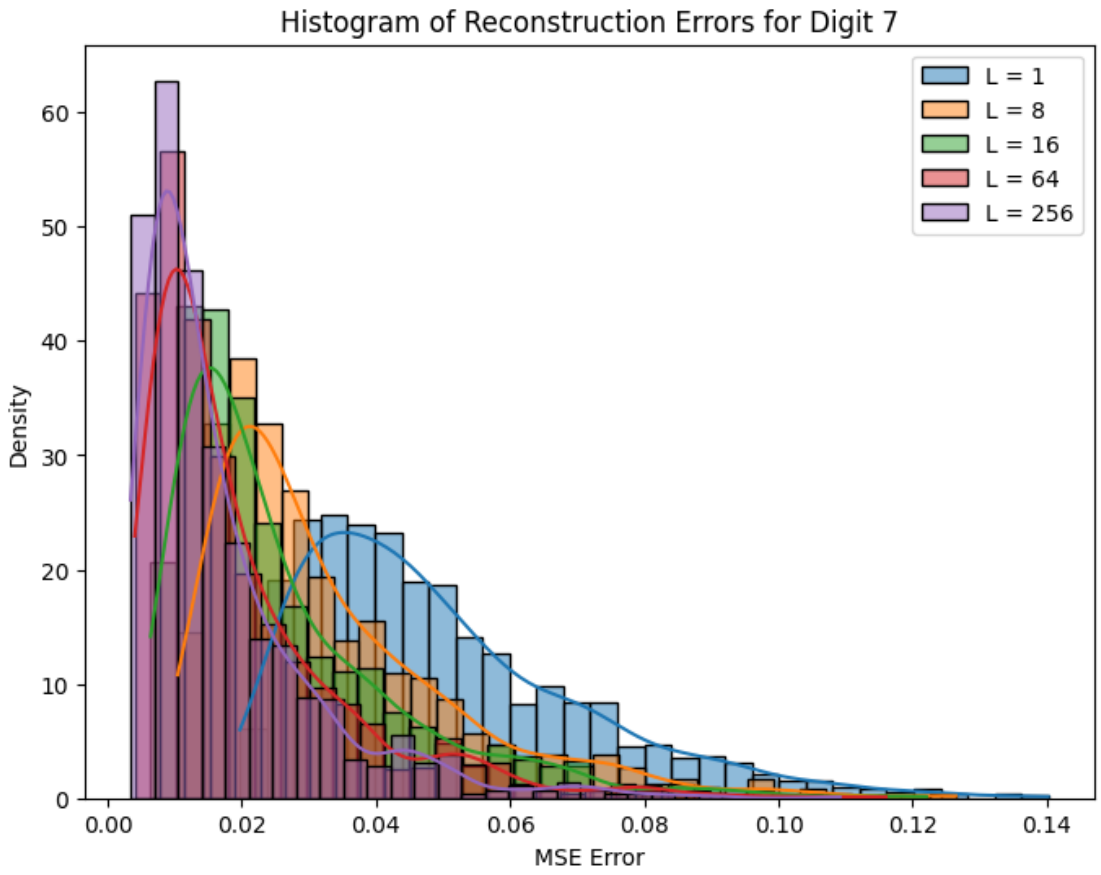
Τα ιστογράμματα των σφαλμάτων καθώς και το σφάλμα ανακατασκευής για κάθε ένα ψηφίο από τα επιλεγμενα.

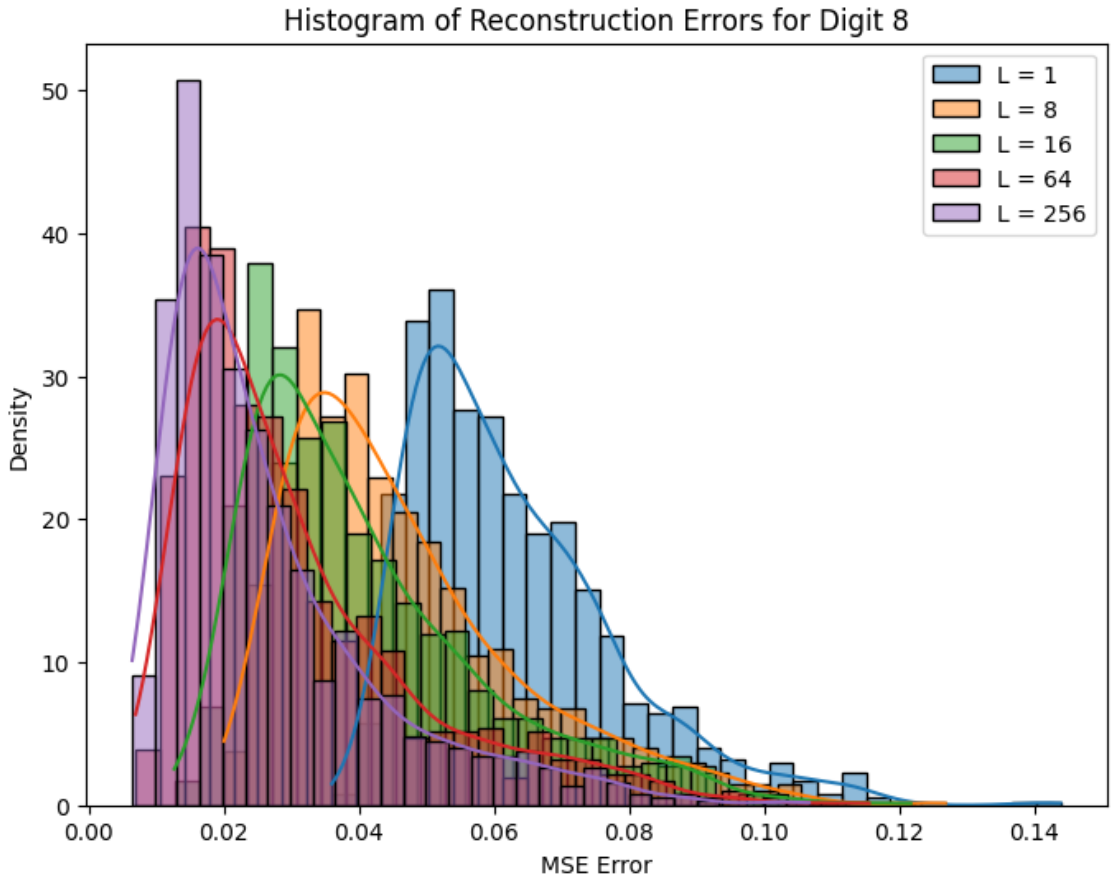


	2	3	7	8
1	0.072096	0.061957	0.049661	0.062819
8	0.055862	0.044758	0.033259	0.046721
16	0.048990	0.038213	0.026316	0.041135
64	0.036733	0.028254	0.019759	0.030978
256	0.031414	0.024290	0.017054	0.026723


Histogram of Reconstruction Errors for Digit 2
L = 1
L = 8
L = 16
L = 64
L = 256
Density
MSE Error
0
5
10
15
20
25
30
35
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14


Histogram of Reconstruction Errors for Digit 3
L = 1
L = 8
L = 16
L = 64
L = 256
Density
MSE Error
0
10
20
30
40
0.02
0.04
0.06
0.08
0.10
0.12


Histogram of Reconstruction Errors for Digit 7
L = 1
L = 8
L = 16
L = 64
L = 256
Density
MSE Error
0
10
20
30
40
50
60
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14


Histogram of Reconstruction Errors for Digit 8
L = 1
L = 8
L = 16
L = 64
L = 256
Density
MSE Error
0
10
20
30
40
50
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14

Διαδικασία Autoencoders

Ερώτημα 1

Ορίζεται ο παρακάτω Γραμμικός Autoencoder:

```
class LinearAutoencoder(nn.Module):
    def __init__(self, input_size=784, latent_size=128):
        super(LinearAutoencoder, self).__init__()
        # In PyTorch, nn.Linear(in_features, out_features) has weight
shape (out_features, in_features)
        self.encoder = nn.Linear(input_size, latent_size, bias=False)
        self.decoder = nn.Linear(latent_size, input_size, bias=False)

    def forward(self, x):
        z = self.encoder(x)
        x_recon = self.decoder(z)
        return x_recon
```

Εκπαιδεύεται για 40 epoch και το αποτέλεσμα της εκτέλεσης φαίνεται στην διπλανή εικόνα.

```
Training on device: cuda
Epoch [1/40], Loss: 0.000221
Epoch [2/40], Loss: 0.000127
Epoch [3/40], Loss: 0.000098
Epoch [4/40], Loss: 0.000085
Epoch [5/40], Loss: 0.000080
Epoch [6/40], Loss: 0.000077
Epoch [7/40], Loss: 0.000076
Epoch [8/40], Loss: 0.000075
Epoch [9/40], Loss: 0.000074
Epoch [10/40], Loss: 0.000074
Epoch [11/40], Loss: 0.000074
Epoch [12/40], Loss: 0.000074
Epoch [13/40], Loss: 0.000074
Epoch [14/40], Loss: 0.000074
Epoch [15/40], Loss: 0.000074
Epoch [16/40], Loss: 0.000074
Epoch [17/40], Loss: 0.000074
Epoch [18/40], Loss: 0.000074
Epoch [19/40], Loss: 0.000074
Epoch [20/40], Loss: 0.000074
Epoch [21/40], Loss: 0.000074
Epoch [22/40], Loss: 0.000074
Epoch [23/40], Loss: 0.000074
Epoch [24/40], Loss: 0.000074
Epoch [25/40], Loss: 0.000074
Epoch [26/40], Loss: 0.000074
Epoch [27/40], Loss: 0.000074
Epoch [28/40], Loss: 0.000074
Epoch [29/40], Loss: 0.000074
Epoch [30/40], Loss: 0.000074
Epoch [31/40], Loss: 0.000074
Epoch [32/40], Loss: 0.000074
Epoch [33/40], Loss: 0.000074
Epoch [34/40], Loss: 0.000074
Epoch [35/40], Loss: 0.000074
Epoch [36/40], Loss: 0.000074
Epoch [37/40], Loss: 0.000074
Epoch [38/40], Loss: 0.000074
Epoch [39/40], Loss: 0.000074
Epoch [40/40], Loss: 0.000074
```

Για την σύγκριση των βαρών με αυτά του PCA στο ερώτημα 2 της διαδικασίας PCA, μετά το τέλος της εκπαίδευσης εξάγουμε τα βάρη του αποκωδικοποιητή του δικτύου και τα αποθηκεύουμε στο w2.

```
w2 = model.decoder.weight.data.cpu().numpy()  # shape: (784, 128)
w2_transposed = w2.T  # shape: (128, 784)
```

Η σύγκριση βασίζεται στην μέθοδο που προτείνεται στο [1].

Έστω ότι διαθέτουμε δεδομένα $Y_0 \in R^{n \times N}$, όπου n είναι η διάσταση κάθε δείγματος (π.χ. αριθμός χαρακτηριστικών) και N ο αριθμός των δειγμάτων. Ένας γραμμικός autoencoder αποτελείται από:
Ένα γραμμικό επίπεδο "encoder" με βάρη $W_1 \in R^{m \times n}$. Αυτό μετασχηματίζει κάθε δείγμα $y \in R^n$ σε έναν κρυφό χώρο διάστασης m:

$$z = W_1 y \quad (\text{όπου } z \in R^m).$$

Ένα γραμμικό επίπεδο "decoder" με βάρη $W_2 \in R^{n \times m}$, το οποίο μετασχηματίζει πίσω από τον κρυφό χώρο z στον αρχικό χώρο R^n:

$$\hat{y} = W_2\, z = W_2\, W_1\, y.$$

Συλλογικά, για όλα τα δεδομένα Y_0,η έξοδος του autoencoder είναι:

$$\hat{Y} = W_2\, W_1\, Y_0.$$

Ο στόχος του autoencoder είναι να "μάθει" τους πίνακες W_1 και W_2 που ελαχιστοποιούν το σφάλμα ανακατασκευής (συνήθως μετρώντας το σφάλμα κατά Frobenius):

$$\min_{W_1, W_2} |Y_0 - W_2\, W_1\, Y_0\, 1|_F^2$$

Όταν ο autoencoder είναι αυστηρά γραμμικός, μπορεί να φανεί ότι ο βέλτιστος W_1και W_2 σχετίζονται με την SVD του Y_0, καθώς το ζητούμενο είναι να βρεθεί ένας υποχώρος διάστασης m που να "εξηγεί" όσο το δυνατόν μεγαλύτερη πληροφορία των δεδομένων.

Μπορούμε να επικεντρωθούμε κυρίως στον πίνακα W2, κάνοντας την παρατήρηση πως ο βέλτιστος W1 μπορεί να εκφραστεί ως συνάρτηση του W2.

Συγκεκριμένα, αν ορίσουμε

$$Z = W_1\, Y_0 \quad (\text{οι "κρυφές αναπαραστάσεις"}),$$

τότε το βέλτιστο W_1 που ελαχιστοποιεί το σφάλμα για δεδομένο W_2 θα είναι ο ψευδοαντίστροφος του W_2 πολλαπλασιασμένος κατάλληλα με τα δεδομένα:

$$W_1 = W_2^{\dagger} = (W_2^T W_2)^{-1} W_2^T.$$

Έτσι, το μοντέλο μπορεί να γραφτεί ως:

$$\hat{Y} = W_2\left(W_2^{\dagger}\, Y_0\right) = W_2\, W_2^{\dagger}\, Y_0.$$

Τελικά, η βελτιστοποίηση περιορίζεται σε:

$$\min_{W_2} |Y_0 - W_2\, W_2^{\dagger}\, Y_0\,|_F^2$$

Όταν οι στήλες του W_2 δεν είναι ορθοκανονικές, ο πίνακας $W_2 W_2^{\dagger}$ ισοδυναμεί με τον ορθογώνιο projection πάνω στον χώρο που ορίζεται από τις στήλες του W_2.

- **Εάν επιβάλουμε ορθοκανονικότητα** στις στήλες του W_2, τότε W_2 γίνεται ο πίνακας που περιέχει τα πρώτα m αριστερά μοναδιαία διανύσματα της SVD του Y_0.
- **Εάν δεν επιβάλουμε ορθοκανονικότητα**, το αποτέλεσμα είναι ο ίδιος υποχώρος, αλλά οι στήλες του W_2 μπορεί να έχουν διαφορετικές κλίμακες μεταξύ τους. Παρ' όλα αυτά, το projection $W_2 W_2^{\dagger}$ παραμένει ίδιο (ως υποχώρος), οπότε το σφάλμα ανακατασκευής είναι το ίδιο.

Αυτό δείχνει ότι η βέλτιστη λύση του γραμμικού autoencoder προσεγγίζει ουσιαστικά την ίδια βέλτιστη προβολή που θα προέκυπτε από την PCA.

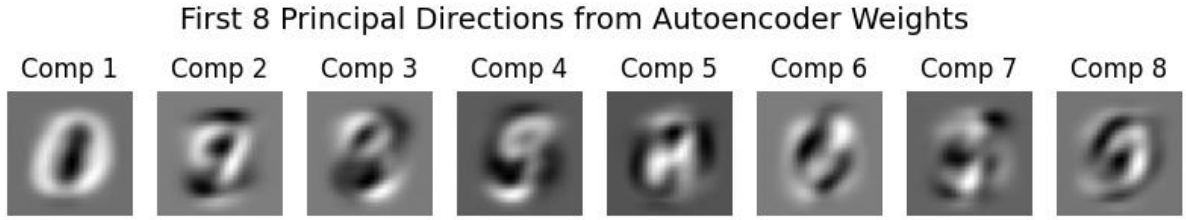
Η βασική Υπόθεση: Οι πρώτοι mmm "loading vectors" του Y (δηλαδή τα πρώτα m αριστερά ιδιοδιανύσματα του SVD) αντιστοιχούν στα πρώτα m αριστερά ιδιοδιανύσματα του πίνακα W_2 που ελαχιστοποιεί την εξίσωση $\min_{W_2} |Y_0 - W_2 W_2^{\dagger} Y_0|_F^2$.

Χρησιμοποιώντας αυτή λοιπόν την υπόθεση:

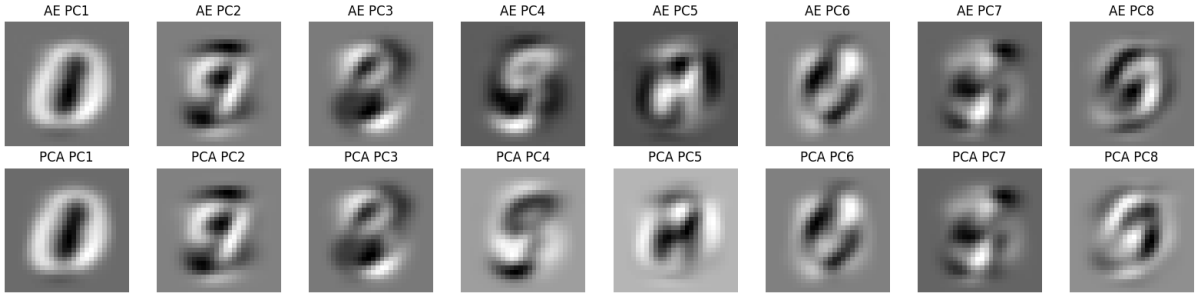
```
U, S, Vt = np.linalg.svd(W2, full_matrices=False)
pca_components_from_ae = U  # shape [784, 128]
```

Ο κώδικας παίρνει τα βάρη του decoder (που είναι μια εκμάθηση του υποχώρου μέσω του autoencoder), κάνει την SVD για να απομονώσει τις κύριες κατευθύνσεις (principal components) και στη συνέχεια χρησιμοποιεί τις στήλες του U ως τα "loading vectors"

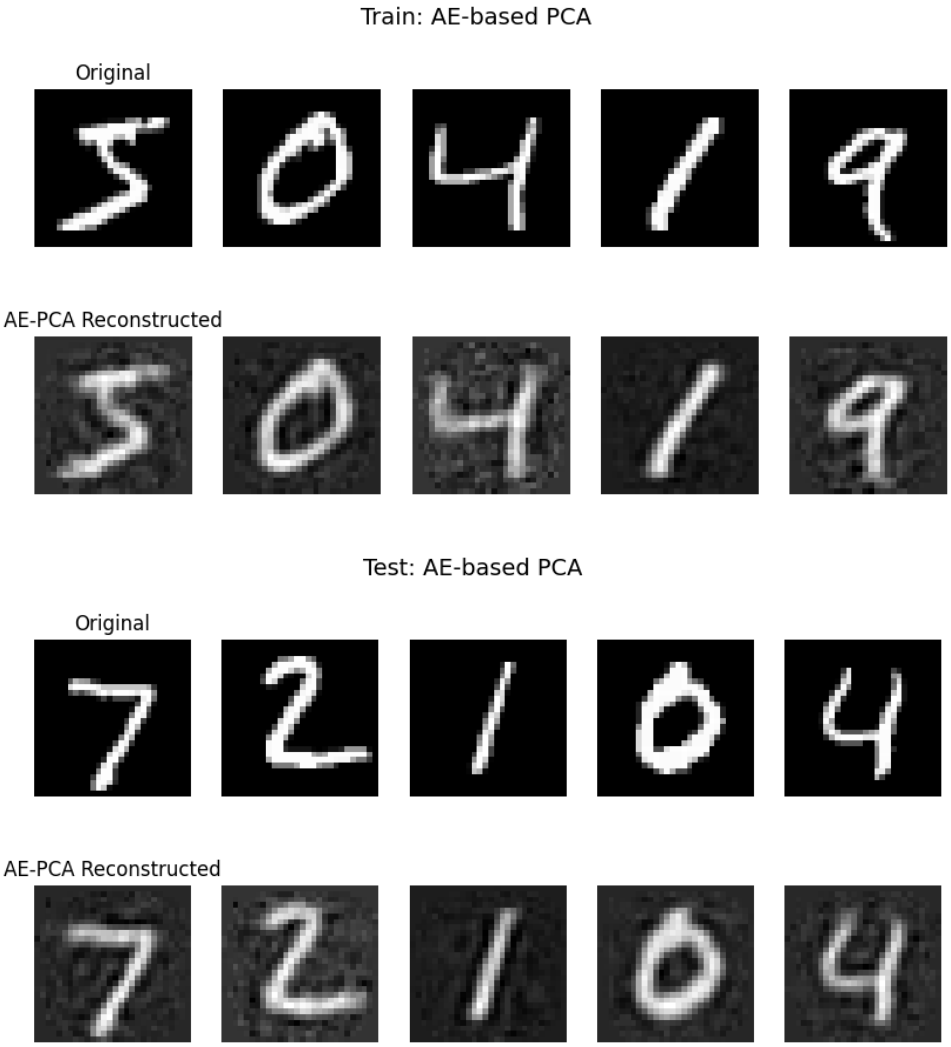
Τυπώνουμε τις 8 πρώτες κύριες συνιστώσες που προκύπτουν από το SVD των βαρών:



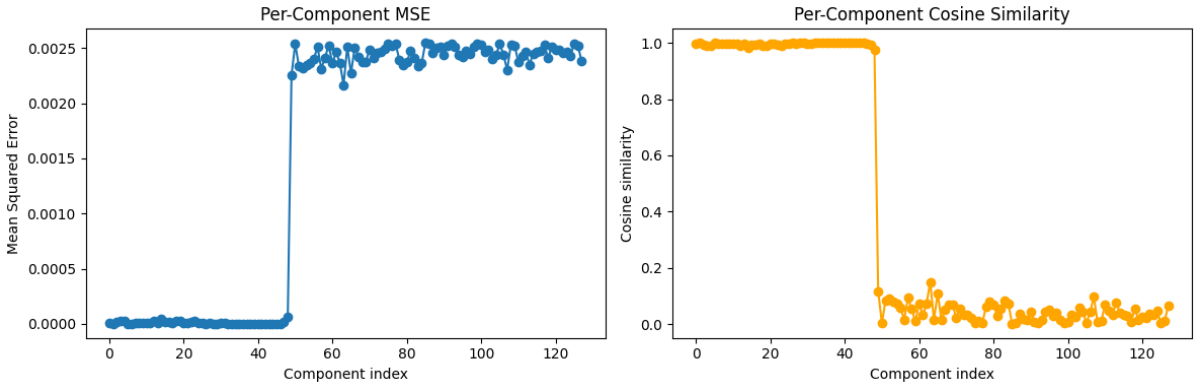
Παρατηρούμε ότι οι κύριες συνιστώσες είναι ίδιες με αυτές του PCA. Μπορεί να διαφέρουν μόνο σε κάποια σημεία ως προς το πρόσιμο.



Παίρνουμε επίσης μερικές ανακατασκευές βάση αυτών των βαρών και στο test και στο train dataset.



Για την σύγκριση των βαρών παίρνουμε την μετρική του Cosine Similarity και MSE:



Παρατηρούμε ότι πράγματι το MSE συγκλίνει στο 0 για τις πρώτες κύριες συνιστώσες του PCA και AE, ενώ στη σύγκριση με το Cosine Similarity τα διανύσματα τείνουν συνευθειακά και πάλι για τις πρώτες κύριες συνιστώσες τους.

Ερώτημα 2

Ορίζεται ο παρακάτω Μη γραμμικός Autoencoder

```
class NonLinearAutoencoder(nn.Module):
    def __init__(self):
        super(NonLinearAutoencoder, self).__init__()
        # Encoder: 784 -> 512 -> 128
        self.encoder = nn.Sequential(
            nn.Linear(784, 512),
            nn.ReLU(),
            nn.Linear(512, 128),
            nn.ReLU()
        )
        # Decoder: 128 -> 512 -> 784
        self.decoder = nn.Sequential(
            nn.Linear(128, 512),
            nn.ReLU(),
            nn.Linear(512, 784)
        )
    def forward(self, x):
        z = self.encoder(x)
        x_recon = self.decoder(z)
        return x_recon
```

Εισάγεται η μη γραμμικότητα μέσω της ReLU.
Το αποτέλεσμα της εκπαίδευσης του δικτύου φαίνεται στην διπλανή εικόνα.

```
Total parameters in the 3-layer Nonlinear AE: 935824
[Nonlinear AE] Epoch [1/40], Train Loss: 0.0193, Test Loss: 0.0097
[Nonlinear AE] Epoch [2/40], Train Loss: 0.0084, Test Loss: 0.0073
[Nonlinear AE] Epoch [3/40], Train Loss: 0.0069, Test Loss: 0.0063
[Nonlinear AE] Epoch [4/40], Train Loss: 0.0061, Test Loss: 0.0058
[Nonlinear AE] Epoch [5/40], Train Loss: 0.0056, Test Loss: 0.0053
[Nonlinear AE] Epoch [6/40], Train Loss: 0.0054, Test Loss: 0.0052
[Nonlinear AE] Epoch [7/40], Train Loss: 0.0052, Test Loss: 0.0051
[Nonlinear AE] Epoch [8/40], Train Loss: 0.0051, Test Loss: 0.0050
[Nonlinear AE] Epoch [9/40], Train Loss: 0.0050, Test Loss: 0.0049
[Nonlinear AE] Epoch [10/40], Train Loss: 0.0050, Test Loss: 0.0048
[Nonlinear AE] Epoch [11/40], Train Loss: 0.0049, Test Loss: 0.0048
[Nonlinear AE] Epoch [12/40], Train Loss: 0.0049, Test Loss: 0.0047
[Nonlinear AE] Epoch [13/40], Train Loss: 0.0048, Test Loss: 0.0048
[Nonlinear AE] Epoch [14/40], Train Loss: 0.0048, Test Loss: 0.0047
[Nonlinear AE] Epoch [15/40], Train Loss: 0.0048, Test Loss: 0.0047
[Nonlinear AE] Epoch [16/40], Train Loss: 0.0048, Test Loss: 0.0049
[Nonlinear AE] Epoch [17/40], Train Loss: 0.0047, Test Loss: 0.0046
[Nonlinear AE] Epoch [18/40], Train Loss: 0.0047, Test Loss: 0.0046
[Nonlinear AE] Epoch [19/40], Train Loss: 0.0047, Test Loss: 0.0046
[Nonlinear AE] Epoch [20/40], Train Loss: 0.0047, Test Loss: 0.0048
[Nonlinear AE] Epoch [21/40], Train Loss: 0.0047, Test Loss: 0.0046
[Nonlinear AE] Epoch [22/40], Train Loss: 0.0047, Test Loss: 0.0046
[Nonlinear AE] Epoch [23/40], Train Loss: 0.0046, Test Loss: 0.0046
[Nonlinear AE] Epoch [24/40], Train Loss: 0.0046, Test Loss: 0.0046
[Nonlinear AE] Epoch [25/40], Train Loss: 0.0046, Test Loss: 0.0046
[Nonlinear AE] Epoch [26/40], Train Loss: 0.0046, Test Loss: 0.0044
[Nonlinear AE] Epoch [27/40], Train Loss: 0.0046, Test Loss: 0.0045
[Nonlinear AE] Epoch [28/40], Train Loss: 0.0045, Test Loss: 0.0044
[Nonlinear AE] Epoch [29/40], Train Loss: 0.0045, Test Loss: 0.0044
[Nonlinear AE] Epoch [30/40], Train Loss: 0.0045, Test Loss: 0.0044
[Nonlinear AE] Epoch [31/40], Train Loss: 0.0045, Test Loss: 0.0045
[Nonlinear AE] Epoch [32/40], Train Loss: 0.0045, Test Loss: 0.0043
[Nonlinear AE] Epoch [33/40], Train Loss: 0.0044, Test Loss: 0.0043
[Nonlinear AE] Epoch [34/40], Train Loss: 0.0044, Test Loss: 0.0043
[Nonlinear AE] Epoch [35/40], Train Loss: 0.0044, Test Loss: 0.0044
[Nonlinear AE] Epoch [36/40], Train Loss: 0.0044, Test Loss: 0.0043
[Nonlinear AE] Epoch [37/40], Train Loss: 0.0044, Test Loss: 0.0043
[Nonlinear AE] Epoch [38/40], Train Loss: 0.0043, Test Loss: 0.0043
[Nonlinear AE] Epoch [39/40], Train Loss: 0.0043, Test Loss: 0.0042
[Nonlinear AE] Epoch [40/40], Train Loss: 0.0043, Test Loss: 0.0042
```

Το δίκτυο έχει `935824` παραμέτρους.
Για κάθε επίπεδο $Linear(n_{\text{in}}, n_{out})$, το πλήθος των παραμέτρων είναι:

$$(weights) = n_{\text{in}} \times n_{\text{out}}, (biases) = n_{\text{out}}$$

Το σύνολο λοιπόν σε κάθε επίπεδο είναι nin×nout+nout.

1ο επίπεδο: Linear(784, 512) 784×512+512 = 401,408+512 = 401,920

2ο επίπεδο: Linear(512, 128) 512×128+128 = 65,536+128 = 65,664

3ο επίπεδο: Linear(128, 512) 128×512+512 = 65,536+512 = 66,048

4ο επίπεδο: Linear(512, 784) 512×784+784 = 401,408+784 = 402,192

Συνολικό πλήθος παραμέτρων στο δίκτυο: 401,920+65,664+66,048+402,192=935,824.

Ερώτημα 3

Ορίζεται το δίκτυο που χρησιμοποιεί ακριβώς τις μισές παραμέτρους.

```
class TiedNonlinearAutoencoder(nn.Module):
    def __init__(self, enc_neg_slope=0.2):
        super().__init__()
        # "Encoder" layers:
        # εδώ ορίζουμε MONO τις μήτρες βαρών για 784->512 και 512->128
        self.fc1 = nn.Linear(784, 512, bias=False)
        self.fc2 = nn.Linear(512, 128, bias=False)

        # ορίζουμε κλίση Leaky ReLU στον encoder
        self.enc_neg_slope = enc_neg_slope
        # κλίση = 1 / enc_neg_slope, δηλ. 5 αν enc_neg_slope=0.2
        self.dec_neg_slope = 1.0 / enc_neg_slope

    def encode(self, x):
        x = F.leaky_relu(self.fc1(x), # x: [batch_size, 784]
        negative_slope=self.enc_neg_slope)
        x = F.leaky_relu(self.fc2(x), negative_slope=self.enc_neg_slope)
        return x  # [batch_size, 128]

    def decode(self, z):
        # Χρησιμοποιούμε τα transpose των βαρών αντί για ανεξάρτητες
        z = torch.matmul(z, self.fc2.weight)  # (batch, 512)
        z = F.leaky_relu(z, negative_slope=self.dec_neg_slope)
        z = torch.matmul(z, self.fc1.weight)  # (batch, 784)
        return z

    def forward(self, x):
        z = self.encode(x)
        x_recon = self.decode(z)
        return x_recon
```

Σε ένα κλασικό fully-connected autoencoder με encoder/decoder ανεξάρτητο, αν τα βάρη στον κωδικοποιητή είναι $W_e \in R^{n\times m}$ και στον αποκωδικοποιητή $W_d \in R^{m\times n}$, τότε ο συνολικός αριθμός παραμέτρων (χωρίς bias) είναι: $n \times m \ + \ m \times n \ = \ 2nm$.

Για να γίνει "στα πρότυπα της PCA" και να έχει ακριβώς τις μισές παραμέτρους, ορίζουμε

$$W_d = W_e^{\mathsf{T}}.$$

Έτσι τα βάρη του αποκωδικοποιητή *δεν* είναι ανεξάρτητα, αλλά "τάσσονται" ίσα με το ανάστροφο μητρώο του encoder. Συνεπώς ο αριθμός παραμέτρων είναι πια μόνον $n \times m$ (συν bias αν υπάρχει). Στη βιβλιογραφία, αυτό λέγεται **weight tying** και είναι ο γνωστός τρόπος να μιμηθεί κανείς την "συμμετρική" μορφή ενός PCA (linearly) με τη μισή χωρητικότητα.

Αποτέλεσμα εκτέλεσης για υπολογισμό παραμέτρων:

Συνολικοί παράμετροι (tied AE): 466944

Που είναι πράγματι οι μισοί από εκείνους στο προηγούμενο ερώτημα.

Leaky ReLU στον encoder

Ας πούμε ότι στον encoder εφαρμόζεται:

$$LeakyReLU_{\alpha=0.2}(x) = \begin{cases} x, & x \geq 0 \\ 0.2\,x, & x < 0 \end{cases}$$

για κάθε επίπεδο του encoder.

Για να αντιστρέψουμε αυτή την επίδραση στον αποκωδικοποιητή—δηλαδή να "ανορθώσουμε" τις τιμές που είχαν μειωθεί—χρειαζόμαστε μια συνάρτηση ενεργοποίησης που θα έχει αντίστροφη συμπεριφορά. Συγκεκριμένα, ορίζουμε ότι στον αποκωδικοποιητή θα χρησιμοποιήσουμε LeakyReLU με αρνητική κλίση β, όπου:

$$LeakyReLU_{\beta}(x) = \begin{cases} z, & z \geq 0 \\ \beta\,z, & z < 0 \end{cases}$$

και για να είναι ακριβής τοπική αντίστροφη ως προς τα αρνητικά, ισχύει $\beta = \frac{1}{0.2} = 5$

Όταν εφαρμόζουμε πρώτα το LeakyReLU(0.2) στον encoder και στη συνέχεια το LeakyReLU(5) στον decoder, η σύνθεση αυτών (για τις αρνητικές τιμές) προσεγγίζει την ταυτότητα, δηλαδή αναστρέφει την αρχική συμπίεση.

Ερώτημα 4

Σε έναν κανονικό autoencoder με πλήρως συνδεδεμένα στρώματα, ο κωδικοποιητής (encoder) και ο αποκωδικοποιητής (decoder) έχουν ανεξάρτητες μήτρες βαρών. Έτσι, ο συνολικός αριθμός παραμέτρων είναι το άθροισμα των παραμέτρων και στα δύο μέρη.

Για να μειώσουμε τις παραμέτρους (δηλαδή να έχουμε «ακριβώς τις μισές παραμέτρους»), μπορούμε να χρησιμοποιήσουμε το λεγόμενο **weight tying**. Στην παραδοσιακή μέθοδο, ο αποκωδικοποιητής έχει βάρη ίσα με το απλό transpose των βαρών του κωδικοποιητή, δηλαδή:

$$W_{decoder} = W_{encoder}^{T}$$

Έτσι, δεν μαθαίνουμε ξεχωριστά τα βάρη για τον αποκωδικοποιητή, αλλά τα "αποδίδουμε" ως τη μεταθέση των βαρών του encoder.

Ωστόσο, για να ακολουθήσουμε πιο στενά τη λογική της ανακατασκευής που βασίζεται στην PCA (όπου η ανακατασκευή γίνεται μέσω της απόσβεσης της μετατροπής των δεδομένων με τα loading vectors), μπορούμε να χρησιμοποιήσουμε τους **ψευδοαντίστροφους** των βαρών του κωδικοποιητή.

Δηλαδή, αντί για το απλό transpose, ορίζουμε:

$$W^{\dagger} = (W\,W^{T})^{-1}W.$$

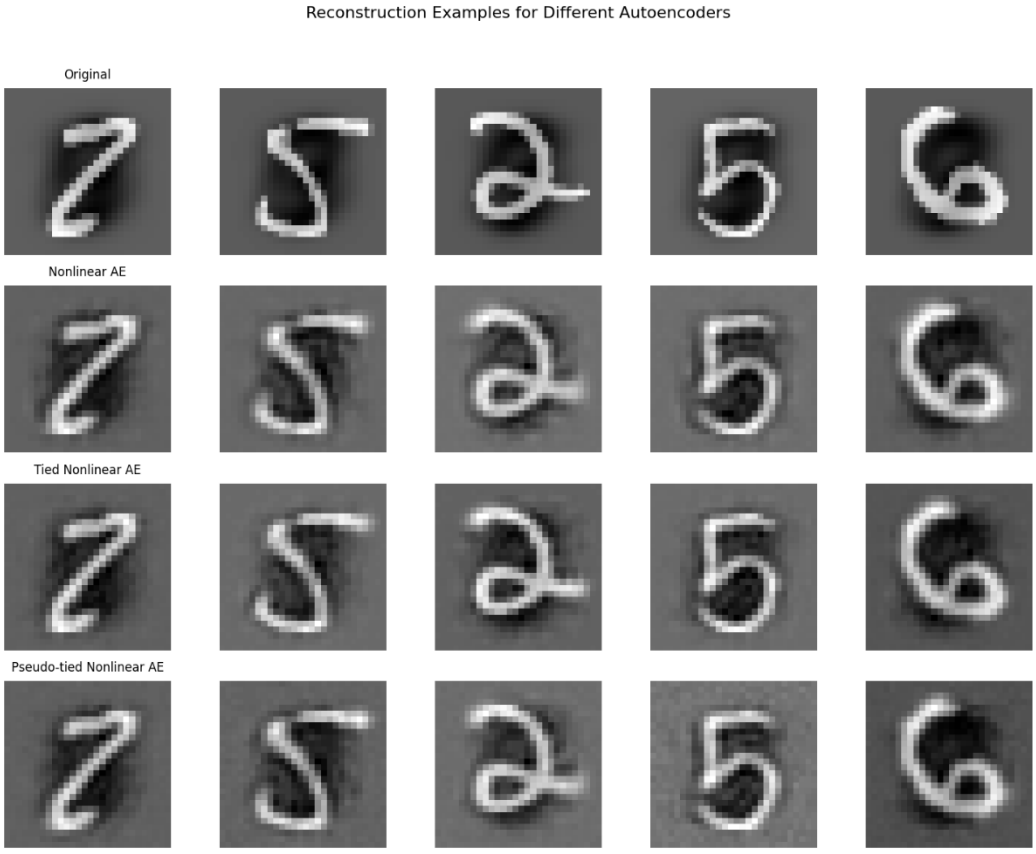
Αυτός ο ψευδοαντίστροφος είναι ο Moore–Penrose ψευδοαντίστροφος και, υπό κατάλληλες προϋποθέσεις, αντιστρέφει το αποτέλεσμα του W. Με αυτόν τον τρόπο, όταν πολλαπλασιάζουμε το κωδικοποιημένο διάνυσμα (latent representation) με τον ψευδοαντίστροφο του W, λαμβάνουμε μια ανακατασκευή του αρχικού διανύσματος με λογική που μιμείται την ανακατασκευή της PCA.

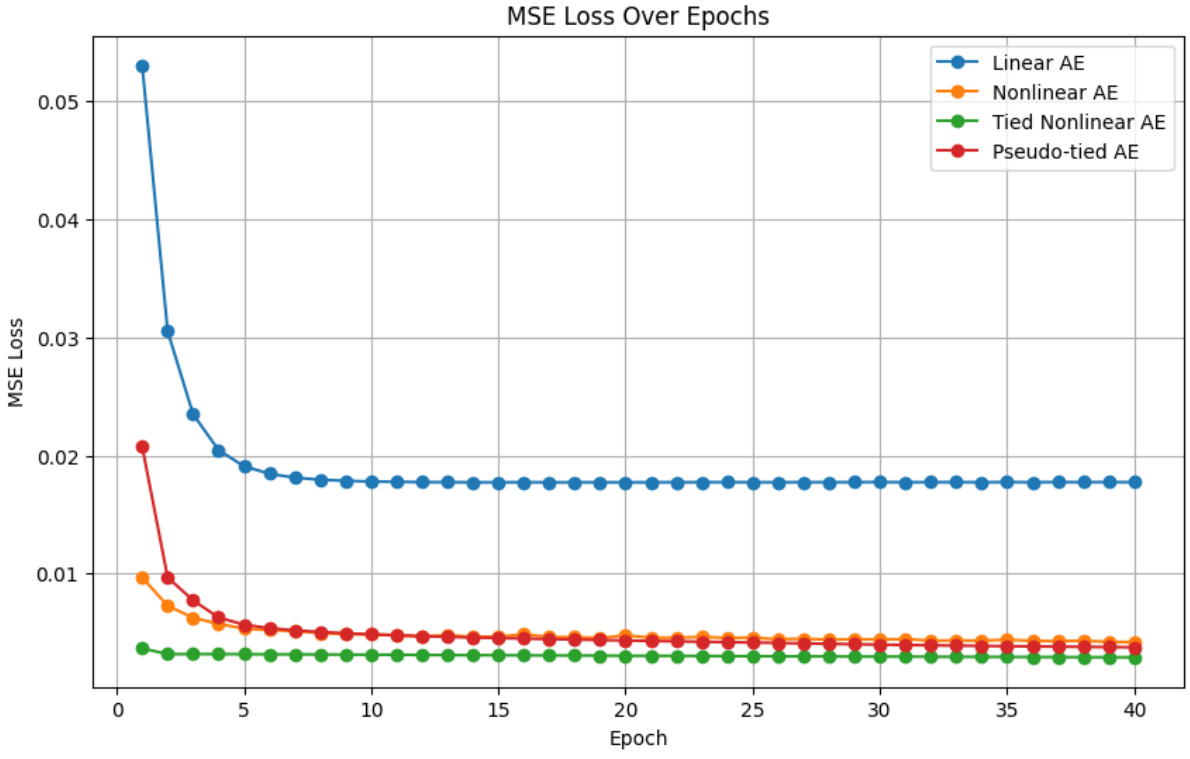
Η λογική εφαρμογής LeakyReLU(0.2) στον encoder και στη συνέχεια το LeakyReLU(5) στον decoder παραμένει η ίδια με αυτή στο προηγούμενο ερώτημα.

```
class PseudoTiedNonlinearAutoencoder(nn.Module):
    def __init__(self, enc_neg_slope=0.2):
        super().__init__()
        self.fc1 = nn.Linear(784, 512, bias=False)
        self.fc2 = nn.Linear(512, 128, bias=False)

        self.enc_neg_slope = enc_neg_slope
        self.dec_neg_slope = 1.0 / enc_neg_slope

    def encode(self, x):
        x = F.leaky_relu(self.fc1(x), # Apply encoder layers with
        negative_slope=self.enc_neg_slope)
        x = F.leaky_relu(self.fc2(x), negative_slope=self.enc_neg_slope)
        return x  # latent representation of shape (batch, 128)

    def decode(self, z):
        pseudo_inv_fc2 = torch.inverse(self.fc2.weight @ self.fc2.weight.T) @
self.fc2.weight
        z = torch.matmul(z, pseudo_inv_fc2)
        z = F.leaky_relu(z, negative_slope=self.dec_neg_slope)

        pseudo_inv_fc1 = torch.inverse(self.fc1.weight @ self.fc1.weight.T) @
self.fc1.weight
        x_recon = torch.matmul(z, pseudo_inv_fc1)
        return x_recon

    def forward(self, x):
        z = self.encode(x)
        x_recon = self.decode(z)
        return x_recon
```

Ερώτημα 5

Παραδείγματα ανακατασκευής των AE που ορίστηκαν:





Διαδικασία Πιθανοτική PCA

Απόδειξη

Ας υποθέσουμε τώρα ότι θέλουμε να υπολογίσουμε την ακόλουθη υπό συνθήκη σππ: $f_{Z|X}(z \mid x)$ η οποία είναι γνωστή ως εκ των υστέρων (posterior) σππ.

Αν γνωρίζουμε την p.d.f. $f_{X(x)}$, τότε από το θεώρημα του Bayes εύκολα βρίσκουμε ότι:

$$f_{X|Z}(x \mid z) = \frac{f_{Z|X}(z \mid x)\, f_X(x)}{f_Z(z)},$$

όπου $f_{Z|X}(z \mid x)$ η συνάρτηση πιθανοφάνειας (likelihood).

Αποδείξτε ότι η παραπάνω υπό συνθήκη p.d.f., με δεδομένα τα παραπάνω, δίνεται από την ακόλουθη σχέση:

$$f_{z|x}(Z|X) = N\left(z; \Sigma_1^{-1} W^T (x-\mu), \sigma^2 \Sigma_1^{-1}\right),$$

όπου $\Sigma_1 = W^T W + \sigma^2 \mathrm{I}$.

Likelihood: $f_{x|z}(X|Z) = \frac{1}{(2\pi\sigma^2)^{\frac{d}{2}}} \exp!\left(-1/2\sigma^2 \mid x-\mu-Wz|^2\right)$

όπου $x,\ \mu \in R^{\mathrm{d}}, z \in R^m$, και W είναι πίνακας διαστάσεων $d \times m$.

Prior για το z (συνήθως υποθέτουμε N(0,I)):

$$f_z(z) = \frac{1}{(2\pi)^{m/2}} \exp!\left(-\frac{1}{2}|z|^2\right).$$

Συνεπώς, $f_{z|x}(Z|X) \propto \exp!\left(-\frac{1}{2\sigma^2}|x-\mu-Wz|^2\right) \exp!\left(-\frac{1}{2}|z|^2\right)$.

Για να βρούμε την ακριβή μορφή (μέση τιμή και συνδιασπορά) της κατανομής $f_{Z|X}(z \mid x)$, αρκεί να αναπτύξουμε και να ομαδοποιήσουμε τους όρους στο εκθετικό μέρος (δηλαδή τον *εκθέτη* του e):

$$-\frac{1}{2\sigma^2}|x-\mu-Wz|^2 - \frac{1}{2}|z|^2.$$

Έστω $|y|^2 = y^T y$. Τότε:

$$|x-\mu-Wz|^2 = (x-\mu-Wz)^T (x-\mu-Wz).$$

Αναπτύσσοντας: $|x-\mu-Wz|^2 = (x-\mu)^T(x-\mu) - 2\, z^T W^T (x-\mu) + z^T W^T W\, z.$

Άρα,

$$-\frac{1}{2\sigma^2}|x-\mu-Wz|^2 = -\frac{1}{2\sigma^2}(x-\mu)^T(x-\mu) \; + \; \frac{1}{\sigma^2}\, z^T\, W^T\, (x-\mu) \; - \; \frac{1}{2\sigma^2}\, z^T\, W^T\, W\, z.$$

Συμπεριλαμβάνουμε και τον όρο του *prior*: $-\frac{1}{2}|z|^2 = \; -\frac{1}{2}\, z^T z.$

Έτσι, ο *συνολικός* εκθέτης γίνεται:

$$-\frac{1}{2\sigma^2}(x-\mu)^T(x-\mu) \; + \; \frac{1}{\sigma^2}\, z^T\, W^T\, (x-\mu) \; - \; \frac{1}{2\sigma^2}\, z^T\, W^T\, W\, z \; - \; \frac{1}{2}\, z^T z.$$

Παρατηρούμε ότι ο μοναδικός όρος που εξαρτάται *γραμμικά* από το z είναι $\frac{1}{\sigma^2} z^T W^T (x-\mu)$, ενώ οι *τετραγωνικοί* όροι ως προς το z είναι:

$$-\frac{1}{2\sigma^2}, z^T W^T W z - \frac{1}{2} z^T z \; = \; -\frac{1}{2} z^T \left(\frac{W^T W}{\sigma^2} + I\right) z$$

Ομαδοποίηση: Κανονικοποίηση σε Μορφή Gaussian
Η γενική μορφή ενός πολυδιάστατου κανονικού εκθέτη είναι:

$$-\frac{1}{2}\left(z-\mu_{\text{post}}\right)^T \Sigma_{\text{post}}^{-1}\left(z-\mu_{\text{post}}\right),$$

όπου μ_{post} και Σ_{post} είναι η μέση τιμή και ο πίνακας συνδιασποράς αντίστοιχα. Για να ταυτίσουμε τους όρους, «κλείνουμε το τετράγωνο» συγκρίνοντας:

$$-\frac{1}{2}\, z^T \left(\frac{W^T W}{\sigma^2} + I\right) z \; + \frac{1}{\sigma^2} z^T\, W^T\, (x-\mu)$$

Από τον τετραγωνικό όρο προκύπτει ότι $\Sigma_{post}^{-1} = \frac{W^T W}{\sigma^2} + I.$

Λύνοντας για Σ_{post}: $\Sigma_{\text{post}} = \left(\frac{W^T W}{\sigma^2} + I\right)^{(-1)}$

$\frac{W^T W}{\sigma^2} + I = \frac{1}{\sigma^2}(W^T W + \sigma^2 I) \Longrightarrow (\frac{W^T W}{\sigma^2} + I) - 1 = \sigma^2 (W^T W + \sigma^2 I)^{-1}.$

Ορίζουμε τον πίνακα $\Sigma_1 = W^T W + \sigma^2 I.$ Τότε $\Sigma_{\text{post}} = \sigma^2\, \Sigma_1^{-1}.$

Για να βρούμε τη μέση τιμή μ_{post}, συγκρίνουμε τον γραμμικό όρο $\frac{1}{\sigma^2} z^T\, W^T\, (x-\mu)$ με τη γενική μορφή $\mu_{\text{post}}^T \Sigma_{\text{post}}^{-1} z$. Συνεπώς,

$$\Sigma_{\text{post}}^{-1}\, \mu_{\text{post}} = \frac{W^T (x-\mu)}{\sigma^2}.$$

Άρα, $\mu_{\text{post}} = \Sigma_{\text{post}}\, \frac{W^T (x-\mu)}{\sigma^2} = \sigma^2\, \Sigma_1^{-1}\, \frac{W^T (x-\mu)}{\sigma^2} = \Sigma_1^{-1}\, W^T\, (x-\mu).$

Διαδικασία Variational Autoencoders

Ερώτημα 1

Υλοποιήθηκε ένα VAE τριών επιπέδων και εκπαιδεύει το δίκτυο για 100 εποχές με latent χώρο 2 διαστάσεων και batch size 250. Ο όρος ανακατασκευής στη συνάρτηση κόστους υπολογίζεται με τη δυαδική εντροπία, ενώ ο όρος τακτοποίησης βασίζεται στην KL divergence για να φέρει την κατανομή των latent παραμέτρων κοντά στην $N(0, I)$. Επιπλέον, ένας batch τυχαίου θορύβου από $N(0, I)$ δίνεται στον αποκωδικοποιητή μετά κάθε φάση, ώστε να παρακολουθείται η εξέλιξη της ανακατασκευής (με παραδείγματα στις εποχές 1, 50 και 100).

Δημιουργία του Fixed Noise Batch:
Στην αρχή της συνάρτησης εκπαίδευσης ορίζεται η μεταβλητή:

```
fixed_noise_sample =
torch.randn(16,model.decoder[0].in_features).to(device)
```

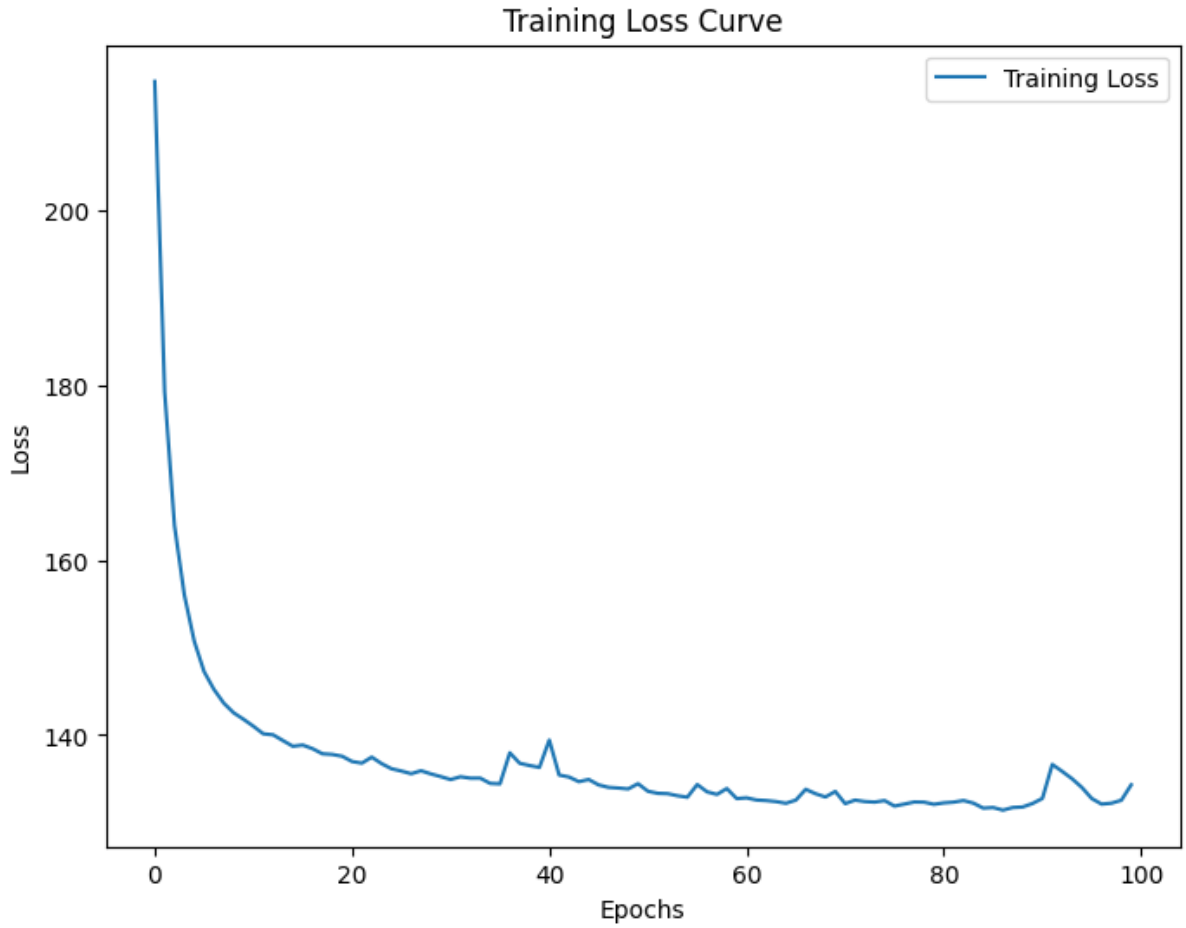
torch.randn(16, model.decoder[0].in_features): Δημιουργεί έναν τανυστή με 16 δείγματα, όπου κάθε δείγμα έχει διάσταση ίση με το `in_features` του πρώτου στρώματος του αποκωδικοποιητή. Δεδομένου ότι ο πρώτος αποκωδικοποιητής ορίζεται ως `nn.Linear(latent_size, hidden_size // 8)`, το `in_features` ισούται με το `latent_size` (δηλαδή 2 σε αυτήν την περίπτωση).

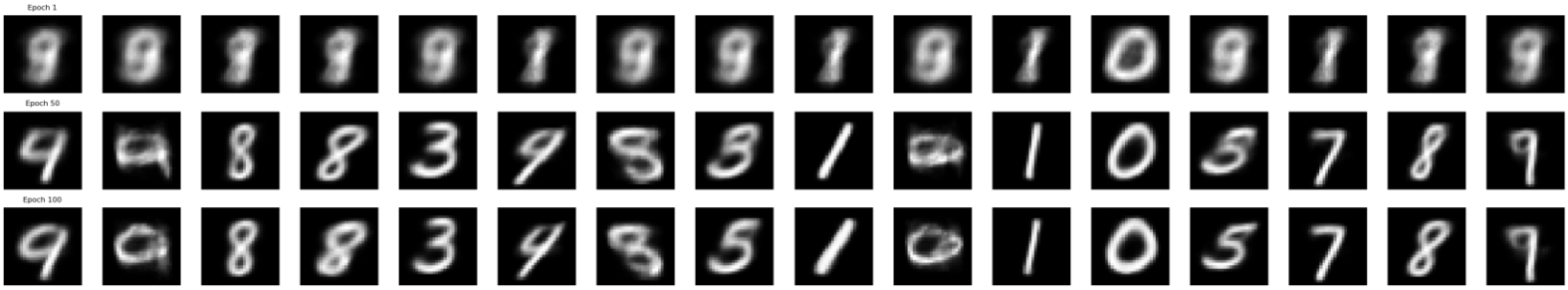
Κατά τη διάρκεια της εκπαίδευσης, σε συγκεκριμένες εποχές (1, 50 και 100), χρησιμοποιείται το ίδιο batch για να παραχθούν εικόνες από τον αποκωδικοποιητή:

```
if (epoch + 1) in epochs_to_visualize:
            with torch.no_grad():
                samples = model.decode(fixed_noise_sample).cpu().view(16, 1, 28, 28)
                generated_images_dict[epoch + 1] = samples
```

Με αυτόν τον τρόπο, το ίδιο batch θορύβου χρησιμοποιείται ως σταθερό σημείο αναφοράς για να μπορούμε να συγκρίνουμε τις ανακατασκευές σε διαφορετικές εποχές και να αξιολογήσουμε πώς βελτιώνεται ο αποκωδικοποιητής καθώς προχωρά η εκπαίδευση.

Εκπαίδευση και αποτελέσματα





Βλέπουμε μερικά δείγματα ανακατασκευής στις εποχές 1 (πρώτη γραμμή), 50 (δεύτερη γραμμή) και 100 (Τρίτη γραμμή).

Παρατηρώντας τις παραγόμενες εικόνες για τις εποχές 1, 50 και 100, βλέπουμε ξεκάθαρα πώς βελτιώνεται η ικανότητα του αποκωδικοποιητή να δημιουργεί πιο «καθαρές» και αναγνωρίσιμες εικόνες ψηφίων καθώς προχωρά η εκπαίδευση.

Ο λόγος που βλέπουμε διακριτή βελτίωση με το πέρασμα των εποχών είναι ότι το VAE:

Μαθαίνει να συμπιέζει (encode) τα δεδομένα σε έναν μικρών διαστάσεων latent χώρο (εδώ 2 διαστάσεις), όπου ο κώδικας (z) φέρει αρκετή πληροφορία για την ανακατασκευή.

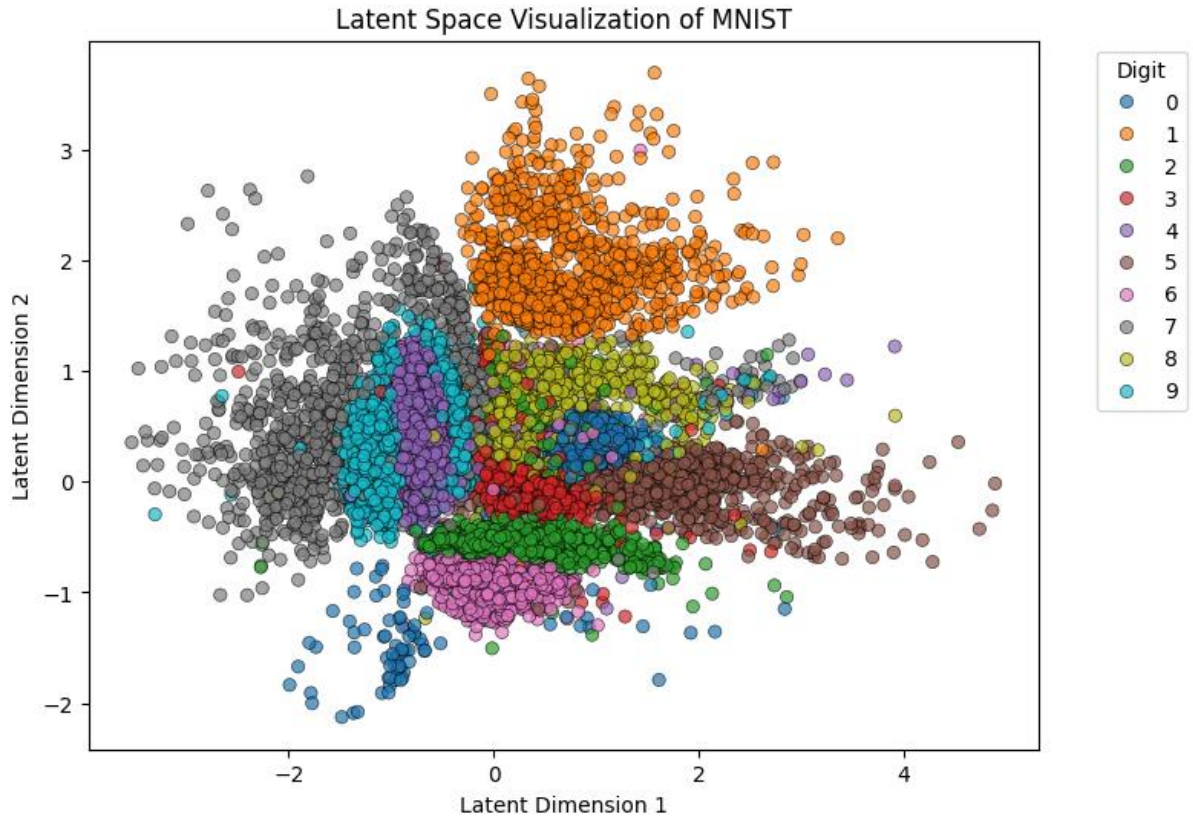
Εξασφαλίζει (με το KL-divergence) ότι η κατανομή των z δεν «ξεφεύγει» πολύ από την $N(0,I)$, ώστε να μπορεί να δέχεται θόρυβο από την ίδια κατανομή και να παράγει έγκυρες εικόνες.

Επαναφέρει (decode) το z πίσω στον χώρο των εικόνων, βελτιώνοντας σταδιακά την ποιότητα της ανακατασκευής μέσω της εκπαίδευσης.

Τέλος, το `fixed_noise_sample` (δηλαδή το ίδιο δείγμα θορύβου που χρησιμοποιείται σε κάθε εποχή) επιτρέπει να έχουμε μια σταθερή «αναφορά» για να συγκρίνουμε την ποιότητα των παραγόμενων εικόνων με την πάροδο των εποχών.

Ερώτημα 2

Λανθάνουσα αναπαράσταση όλων των δεδομένων ελέγχου.



Κάθε ψηφίο τείνει να συγκεντρώνεται σε μία «περιοχές» στον χώρο, αποδεικνύοντας ότι το μοντέλο έμαθε να διαχωρίζει τα διαφορετικά ψηφία στη λανθάνουσα αναπαράσταση. Παρότι σε ορισμένες περιπτώσεις οι περιοχές για διαφορετικά ψηφία επικαλύπτονται, γενικά υπάρχει αρκετά καλή ομαδοποίηση.

Ψηφία που μοιάζουν μεταξύ τους οπτικά (π.χ. το '3' και το '8', ή το '4' και το '9') μπορεί να εμφανίζουν κάποιες ζώνες επικάλυψης ή βρίσκονται κοντά στον λανθάνοντα χώρο, που είναι αναμενόμενο, καθώς οι «κοντινές» οπτικά κατηγορίες τείνουν να τοποθετούνται πιο κοντά μεταξύ τους σε έναν χώρο που μαθαίνει δομή (manifold) όπως το VAE.

Ο συγκεκριμένος λανθάνων χώρος επιτρέπει στο VAE να «περιηγείται» ανάμεσα στις διάφορες κατηγορίες, παράγοντας «ενδιάμεσες» μορφές ψηφίων. Αν μετακινηθούμε σταδιακά σε αυτόν τον 2D χώρο, μπορούμε να δούμε πώς το δίκτυο «μεταμορφώνει» ένα ψηφίο σε άλλο, γεγονός που υποδηλώνει ότι το VAE έχει μάθει μια «συνεχή» αναπαράσταση των δεδομένων.

Κώδικας και στο Google Colab

Διαδικασία PCA:
https://drive.google.com/file/d/1KT0yw4-gLm4hniPcPltchkNY-nozrJYu/view?usp=sharing
Διαδικασία kernel-PCA:
https://drive.google.com/file/d/1sK6wZR9kYS2W_lce0et9qrs0TwOaWrlA/view?usp=sharing
Διαδικασία Autoencoders:
https://drive.google.com/file/d/1iQfkJzd837LnTI618lYkBLMzPexP3TjT/view?usp=sharing
Διαδικασία Variational Autoencoders:
https://drive.google.com/file/d/11csBH5Flry1s7hlBFOlKkx9f9D5ZYybb/view?usp=sharing

References

1. E. Plaut, "From Principal Subspaces to Principal Components with Linear Autoencoders," *arXiv preprint*, arXiv:1804.10253, Dec. 2018. [Online]. Available: https://arxiv.org/pdf/1804.10253
2. B. Chakraborty, "Nonlinear Dimension Reduction, Kernel PCA (kPCA), and Multidimensional Scaling — An Easy Tutorial with Python," *Towards Data Science*, Medium, 2021. [Online]. Available: https://medium.com/towards-data-science/nonlinear-dimension-reduction-kernel-pca-kpca-and-multidimensional-scaling-an-easy-tutorial-63429ee9d0ae
3. K. Daniels, "Principal Component Analysis and the Autoencoder," *Machine Learning and Deep Learning Fundamentals*, 2018. [Online]. Available: https://kenndanielso.github.io/mlrefined/blog_posts/11_Linear_unsupervised_learning/11_2_Principal_Component_Analysis.html
4. Q. Fournier and D. Aloise, "Empirical Comparison Between Autoencoders and Traditional Dimensionality Reduction Methods," in *Proceedings of the IEEE Second International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*, Sardinia, Italy, 2019, pp. 211–214. [Online]. Available: https://arxiv.org/pdf/2103.04874
5. C. Doersch, "Tutorial on Variational Autoencoders," *arXiv preprint*, arXiv:1606.05908, Jan. 2021. [Online]. Available: https://arxiv.org/pdf/1606.05908
6. S. Sakurada and T. Yairi, "Autoencoder and Its Various Variants," in *Proceedings of the IEEE International Conference on Consumer Electronics (ICCE)*, Las Vegas, NV, USA, 2019, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/8616075
7. J. A. Richards, "Remote sensing digital image analysis," *Computers & Geosciences*, vol. 19, no. 6, pp. 836–836, 1993. [Online]. Available: https://www.sciencedirect.com/science/article/pii/009830049390090R