ΜΥΥ 002 Μηχανική Μάθηση

ΣΕΤ ΑΣΚΗΣΕΩΝ 2

Νικόλαος Γιαννόπουλος Α.Μ. 5199
Γεώργιος Στρούγγης Α.Μ. 5357

Μία σύντομη περιγραφή

Σε αυτό το ΣΕΤ ασκήσεων αναλαμβάνουμε να πραγματοποιήσουμε:

- Μείωση διάστασης μέσω PCA και ENCODER
- Ομαδοποίηση δεδομένων μέσω k-means, αναζητώντας:
    - Silhouette score
    - Cluster center
    - Putity και F-measure

ΒΙΒΛΙΟΘΗΚΕΣ:

Οι βιβλιοθήκες/imports που χρησιμοποιήθηκαν είναι οι εξής:

```
#All the imports happen here
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import LabelEncoder
import torch as t
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
```

DATA IMPORT:

Πρώτο βήμα πριν μπορέσουμε να εκτελέσουμε οποιαδήποτε μέθοδο είναι να φορτώσουμε τα δεδομένα που θα χρησιμοποιηθούν. Αρχικά φορτώνουμε ξεχωριστά τα δεδομένα εκπαίδευσης και ελέγχου ως train_data και test_data πριν δούμε την μορφή τους μέσω των εντολών .shape και .head. Αρχικοποιούνται οι τιμές d=784 και n=1000 που συμβολίζουν την τιμή διάστασης των αρχικών εικόνων και τον αριθμό εικόνων που θα χρησιμοποιηθεί από κάθε κατηγορία των αρχικών δεδομένων εισόδου αντίστοιχα.

Στη συνέχεια μέσω for-loop καταφέρνουμε να φτιάξουμε ένα δείγμα από τα αρχικά δεδομένα σύμφωνα με την τιμή που προηγουμένως ορίσαμε. Χρησιμοποιώντας αυτό το δείγμα για τα δεδομένα μας τα σπάμε σε δύο κατηγορίες, τα x_train και x_test που περιέχουν τις πληροφορίες για τις εικόνες ενώ τα y_train και y_test τα labels που τους αντιστοιχούν. Ώστε τα δεδομένα μας να μπορούν να χρησιμοποιηθούν γίνεται χρήση StandardScaler και αργότερα .reshape για να πάρουν την μορφή 28x28 την οποία χρειαζόμαστε. Τέλος εκτυπώνονται οι φωτογραφίες στην τελική τους μορφή ώστε να δούμε αν έχει επιτύχει η διαδικασία καθώς και πως μοιάζουν.

```
#1 Dimensional Reduction

d = 784

#Load the data
train_data = pd.read_csv('fashion-mnist_train.csv')
test_data = pd.read_csv('fashion-mnist_test.csv')

#Make sure it got loaded correctly
print(f"Training data shape: {train_data.shape}")
print(f"Testing data shape: {test_data.shape}")
#Print a few data form each data set
print("\nFirst few rows of training data:")
print(train_data.head())
print("\nFirst few rows of testing data:")
print(test_data.head())

#Initialize number of samples for each category
n = 1000

#Sample n images for each category from the training data
sampled_data_list = []
for label in train_data['label'].unique():
    sampled_data = train_data[train_data['label'] == label].sample(n=n,
random_state=42)
    sampled_data_list.append(sampled_data)
train_data_sample = pd.concat(sampled_data_list).reset_index(drop=True)

#Split images and labels as X and Y
X_train = train_data_sample.drop(columns=['label'], axis=1).values
X_test = test_data.drop(columns=['label'], axis=1).values
Y_train = train_data_sample['label'].values
Y_test = test_data['label'].values
```

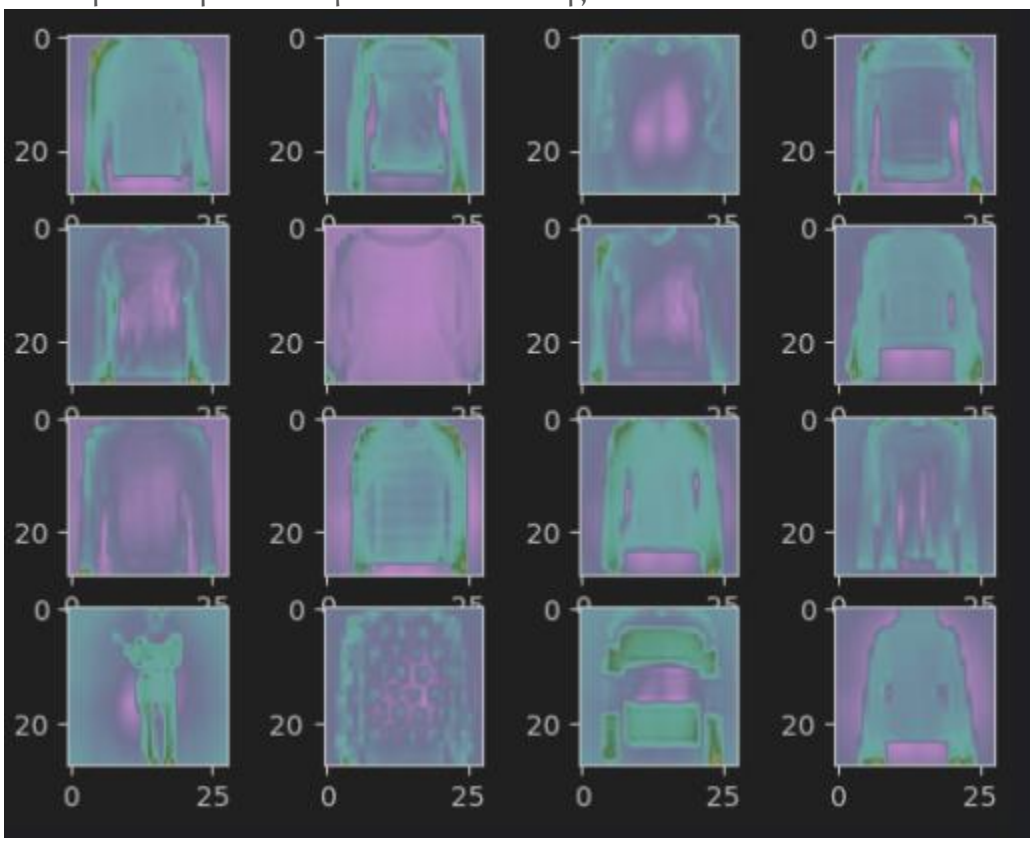
```
#Use Standard Scaler for the images
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

#Finalize Y
y_train = Y_train
y_test = Y_test

#Resize to 28x28
x_train = X_train_scaled.reshape((-1, 28, 28))
x_test = X_test_scaled.reshape((-1, 28, 28))

#View Train Data as pictures
for i in range(16):
    plt.subplot(4, 4, i + 1)
    plt.imshow(x_train[i], cmap='viridis')
plt.show()
```

OUTPUT AND VISUALIZATION:

Training data shape: (60000, 785)

Testing data shape: (10000, 785)

Αναπαράσταση των δεδομένων εκπαίδευσης



Μείωση διάστασης

PCA:

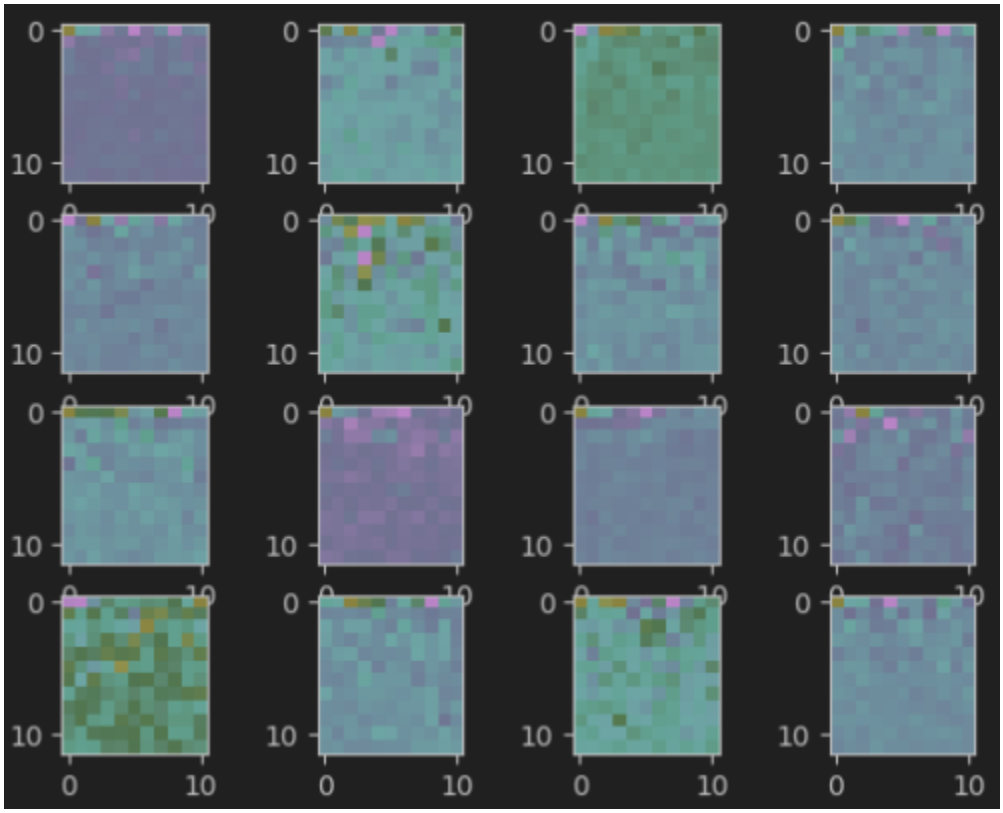
Για να μειωθεί η διάσταση δεδομένων μέσω PCA καλούμε την αντίστοιχη μέθοδο PCA() με είσοδο 0.90 η οποία θα δατηρήσει το 90% της διακύμανσης. Για τα παραπάνω δεδομένα X_train_scaled και X_test_scaled εκτελούνται pca.fit_transform και pca.transform αντίστοιχα ώστε να εκτελεσθεί pca σε αυτά και να αποτκήσουμε τα δεδομένα μειωμένων διαστάσεων X_train_pca και X_test_pca. Από το πρώτο θα βρούμε την διάσταση τους M = 132 μέσω .shape. Στη συνέχεια για να μπορέσουμε να δούμε την εμφάνιση αυτών των δεδομένων τα μετατρέψουμε σε μορφή εικόνας διαστάσεων 12 επί 11, καθώς 12x11=132 και τέλος ένα for-loop τα εκτυπώνει.

```
# Reduce dimensionality through PCA while keeping 90% covariance
pca = PCA(n_components=0.90)
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)

# Print the dimensionality
M = X_train_pca.shape[1]
print(f"Dimensionality: {M}")

#Resize to 12x11. These dimensions were chosen because the square root
of 132 is a floating point number, so instead we need to use values for
its dimensions that are not equals, such as 12x11=132.
x_train_pca = X_train_pca.reshape((-1, 12, 11))
x_test_pca = X_test_pca.reshape((-1, 12, 11))

# View Train Data after PCA as pictures
for i in range(16):
    plt.subplot(4, 4, i + 1)
    plt.imshow(x_train_pca[i], cmap='viridis')
plt.show()
```

OUTPUTS AND VISUALIZATION:

Dimensionality: 132

Παρατηρούμε ότι η εμφάνιση των εικόνων αποτελείται από μια μάζα από pixel, κάτι λογικό και αναμενόμνο αν και όχι όμορφο στο μάτι.



AUTOENCODER:

Ο Autoencoder λειτουργεί μέσω PyTorch. Στην ουσία, ορίζουμε ένα model βάση της αρχιτεκτονικής: «d – d/4 - M – d/4 – d»:

```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential( #Encoding
            nn.Linear(d, d//4),
            nn.ReLU(),
            nn.Linear(d//4, M),
            nn.ReLU()
        )
        self.decoder = nn.Sequential( #Decoding
            nn.Linear(M, d//4),
            nn.ReLU(),
            nn.Linear(d//4, d),
            nn.Sigmoid()
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded
```

Στην συνέχεια, πραγματοποιούμε απαραίτητες αλλαγές ώστε να μπορέσουμε να χρησιμοποιήσουμε τα δεδομένα μας κάνοντας ομαλοποίηση μέσω διαίρεσης με το 255(ώστε τα pixel να έχουν τιμές από 0 έως 1 αντί για 255), και στη συνέχεια, μετατροπή σε tensors:

```
#Normalize data
X_train_new = X_train.astype(np.float32) / 255.0
X_test_new = X_test.astype(np.float32) / 255.0

#Transform to tensors
x_train_torch = t.tensor(X_train_new, dtype=t.float32)
x_test_torch = t.tensor(X_test_new, dtype=t.float32)
```

Οριστικοποιούμε το model με batch size 50(από προηγούμενη άσκηση), την συνάρτηση σφάλματος(επιλέγουμε την MSELoss) και τον optimizer(Adam, όπως και στην προηγούμενη άσηση). Τα epochs θα εξηγηθούν στη συνέχεια:

```
#Create DataLoader. Last time we did it hard coded and had many
challenges. Datasets and Loader seem to be more efficient
#Batch size 50 like last time
train_dataset = TensorDataset(x_train_torch, x_test_torch)
train_loader = DataLoader(train_dataset, batch_size=50, shuffle=True)
```

```
#Model, criterion and optimizer
model = Autoencoder()
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

#Set number of epochs, the more epochs, the better the outcome
#epochs = 5
#epochs = 10
#epochs = 20
epochs = 10
```

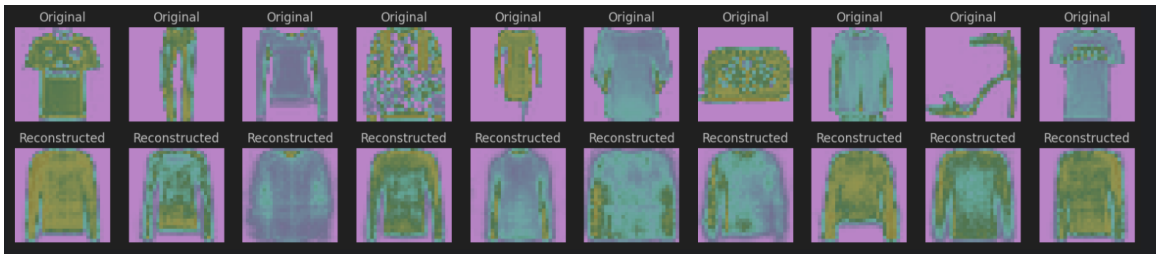
Τέλος, ξεκινάμε το training loop με τη γνωστή διαδικασία, με μόνη διαφορά το evaluation καθώς μας ενδιαφέρει να κάνουμε encode και decode τα δεδομένα και να δούμε πόσο κοντά είναι στις αρχικές εικόνες:

```
#Start of training loop
for epoch in range(epochs):
    for data in train_loader:
        inputs, _ = data

        outputs = model(inputs)
        loss = criterion(outputs, inputs)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')

#Encode and decode images
model.eval()
with t.no_grad():
    encoded_data = model.encoder(x_train_torch)
    decoded_data = model.decoder(encoded_data).numpy()

#Visualize original and reconstructed images
r = 10
plt.figure(figsize=(20, 4))
for i in range(r):
    #Display original
    plot = plt.subplot(2, r, i + 1)
    plt.imshow(X_test[i].reshape(28, 28), cmap='viridis')
    plt.title("Original")
    plt.axis('off')

    #Display reconstructed
    plot = plt.subplot(2, r, i + 1 + r)
    plt.imshow(decoded_data[i].reshape(28, 28), cmap='viridis')
    plt.title("Reconstructed")
    plt.axis('off')
plt.show()
```

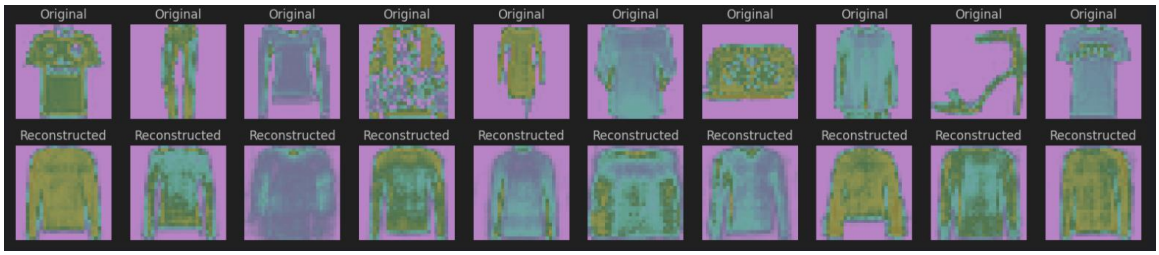
OUTPUTS AND VISUALIZATION:

Αρχικά να πούμε ότι περιμένουμε τα reconstructed images να είναι κοντά στις αρχικές εικόνες. Αυτό είναι κάτι που επηρεάζεται άμεσα από τα epochs, διότι, όσες πιο πολλές επαναλήψεις, τόσο πιο κοντά θα είναι μεταξύ τους(αλλά πο΄τε ίδια).

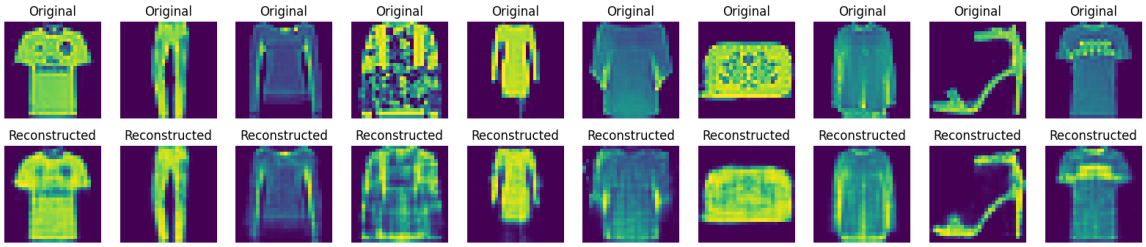
Αποτελέσματα για epochs = 10:



Αποτελέσματα για epochs = 20:



Αποτελέσματα για epochs = 1000:



(η φωτογραφία φαίνεται αλλιώς διότι ήταν αποθηκευμένη καθώς για 1000 epochs, λογικό και αναμενόμενο, πέρνει και απαιτεί αρκετή ώρα)

COMPARISON WITH DATA CLASSIFICATION METHODS:

Η καλύτερη μέθοδος Data Classification από το ΣΕΤ ασκήσεων 1 ήταν το FeedForwardNeuralNetwork για εμάς. Δυστυχώς όμως, δεν καταφέραμε να το κάνουμε να λειτουργήσει, συνεπώς αρκεστήκαμε στην δεύτερη καλύτερη μέθοδο η οποία ήταν το Random Forest Classifier. Αρχικά, κάνουμε encode τα δεδομένα για το train και test, και στη συνέχεια βλέπουμε το accuracy. Παρομοίως και για το PCA.

```
#Second best from last exercise because best didnt work well here
#First we need the data to be encoded so we use them
autoencoder = Autoencoder()
autoencoder.eval()
with t.no_grad():
    train_tensor_encoded = autoencoder.encoder(t.tensor(X_train_scaled,
dtype=t.float32))
    test_tensor_encoded = autoencoder.encoder(t.tensor(X_test_scaled,
dtype=t.float32))

#We save them on the array, although they are not array just to
differentiate them
x_train_array = train_tensor_encoded.numpy()
x_test_array = test_tensor_encoded.numpy()

#Create and train the Random Forest with the 100 estimators
random_forest = RandomForestClassifier(n_estimators=100)
random_forest.fit(x_train_array, y_train)
#Forest accuracy
accuracy = random_forest.score(x_test_array, y_test)
print(f"Accuracy with Random Forest on Autoencoder: {accuracy}\n")

#Create and train the Random Forest with the 100 estimators
random_forest2 = RandomForestClassifier(n_estimators=100)
random_forest2.fit(X_train_pca, y_train)
#Forest accuracy
accuracy = random_forest2.score(X_test_pca, y_test)
print(f"Accuracy with Random Forest on PCA: {accuracy}\n")
```

OUTPUTS AND OBSERVATIONS:

Accuracy with Random Forest on Autoencoder: 0.7928

Accuracy with Random Forest on PCA: 0.8428

Παρατηρούμε ότι, το accuracy, μειώθηκε για τον Autoencoder αλλά αυξήθηκε, αν και ελάχιστα για το PCA. Για παραπομπή, στην προηγούμενη άσκηση, το accuracy score ήταν:

Accuracy with Random Forest: 0.8373

Ομαδοποίηση

Οι μέθοδοι ομαδοποίησης υλοποιήθηκαν με παρόμοιο τρόπο στο PCA και Autoencoder με μόνη διαφορά η είσοδος X_train_pca και encoded_data αντίστοιχα σε συγκεκριμένες εντολές που φαίνονται από τον παρακάτω κώδικα, γι' αυτό τον λόγο θα δοθεί μία εξήγηση καθώς ισχύι και για τα δύο. Η υλοποίηση για τα ερωτήματα 2α και 2β έγινε σε ένα cell κώδικα ενώ το 2γ σε ξεχωριστό, συνολικά δηλαδή τέσσερα, δύο για το PCA και δύο για Autoencoder.

SILHUETTE SCORE:

Για τις τιμές k=10 μέχρι 20 πραγματοποιήθηκε η εντολή KMeans που αποθηκεύτηκε ως kmeans. Από αυτή βρέθηκαν τα labels των cluster μέσω .fit_predict() για X_train_pca ή encoded_data και μέσω silhouette_score γα ευκλείδια απόσταση βρήσκουμε το σκορ για αυτή την τιμή του k. Στο τέλος κάθε loop αποθηκεύοντα οι τιμές score και kmeans σε λίστες silhouette_scores και k_list πριν εκτυπωθεί το σκορ για αυτή την τιμή του k.

Στη συνέχεια μέσω συγκρίσεων βρίσκουμε ποια τιμή που αποθηκεύσαμε αντιστοιχεί στο καλύτερο σκορ, την θέση της και την ομάδα στην οποία ανήκει. Τέλος εκτυπώνουμε το καλύτερο σκορ καθώς και τον αριθμό τον clusters για τον οποίο παρουσιάζεται μαζί με ένα διάγραμμα το οποίο θα δείχνει την τιμή του σε σύγκριση με τις άλλες τιμές του k.

- PCA:

```
#2 Clustering
#k-means for PCA
silhouette_scores = []
k_list = []

#Loop for k values with random_state equals 42 after research on which
should be
for k in range(10, 21):
    kmeans = KMeans(n_clusters=k, random_state=42)
    cluster_labels = kmeans.fit_predict(X_train_pca)
    score = silhouette_score(X_train_pca, cluster_labels,
metric='euclidean')
    silhouette_scores.append(score)
    k_list.append(kmeans)
    print(f'Cluster: {k}, Score: {score}')

#Best value for k and score
k_best = k_list[np.argmax(silhouette_scores)]
k_best_score = 0
k_best_pos = 0
kmeans_best = 0
for i in range(10):
    if silhouette_scores[i] > k_best_score:
        k_best_score = silhouette_scores[i]
        k_best_pos = i + 10
        kmeans_best = k_list[i]
```

```
print(f"Optimal number of clusters: {k_best_pos} with a score of
{k_best_score}")
center = k_best.cluster_centers_

#Plot the silhouette scores
plt.figure(figsize=(10, 6))
plt.plot(range(10, 21), silhouette_scores, marker='o')
plt.title('Silhouette Scores for different K values')
plt.xlabel('Number of Clusters')
plt.ylabel('Silhouette Score')
plt.show()
```

Results:

Cluster: 10, Score: 0.144992068805032 32

Cluster: 11, Score: 0.136904433937442 67

Cluster: 12, Score: 0.128176536462448 48

Cluster: 13, Score: 0.124323538394925 15

Cluster: 14, Score: 0.129271950708007 37

Cluster: 15, Score: 0.123905165503560 93

Cluster: 16, Score: 0.126068591306950 72

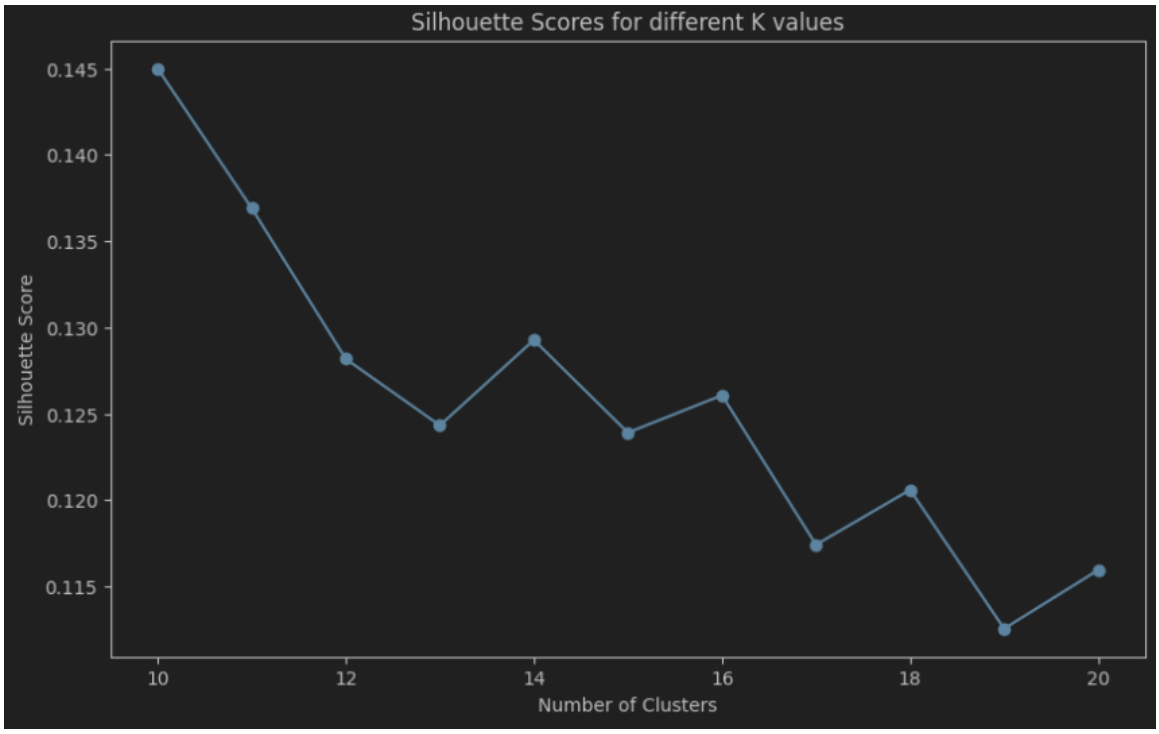
Cluster: 17, Score: 0.117412116769808 6

Cluster: 18, Score: 0.120593342465663 14

Cluster: 19, Score: 0.112554012441822 49

Cluster: 20, Score: 0.115947086335277 4

Optimal number of clusters: 10 with a score of 0.144992068805032 32



- Autoencoder:

```
#k-means for Autoencoder
silhouette_scores = []
k_list = []


for k in range(10, 21):
    kmeans = KMeans(n_clusters=k, random_state=42)
    cluster_labels = kmeans.fit_predict(encoded_data)
    score = silhouette_score(encoded_data, cluster_labels,
metric='euclidean')
    silhouette_scores.append(score)
    k_list.append(kmeans)
    print(f'Cluster: {k}, Score: {score}')

#Best value for k and score
k_best = k_list[np.argmax(silhouette_scores)]
k_best_score = 0
k_best_pos = 0
kmeans_best = 0
for i in range(10):
    if silhouette_scores[i] > k_best_score:
        k_best_score = silhouette_scores[i]
        k_best_pos = i + 10
        kmeans_best = k_list[i]
print(f"Optimal number of clusters: {k_best_pos} with a score of
{k_best_score}")
```

```
center = k_best.cluster_centers_

#Plot the silhouette scores
plt.figure(figsize=(10, 6))
plt.plot(range(10, 21), silhouette_scores, marker='o')
plt.title('Silhouette Scores for different K values')
plt.xlabel('Number of Clusters')
plt.ylabel('Silhouette Score')
plt.show()
```

Results:

Cluster: 10, Score: 0.17278625071048737

Cluster: 11, Score: 0.16966243088245392

Cluster: 12, Score: 0.15842421352863312

Cluster: 13, Score: 0.15820235013961792

Cluster: 14, Score: 0.1614837348461151

Cluster: 15, Score: 0.15799270570278168

Cluster: 16, Score: 0.1560397893190384

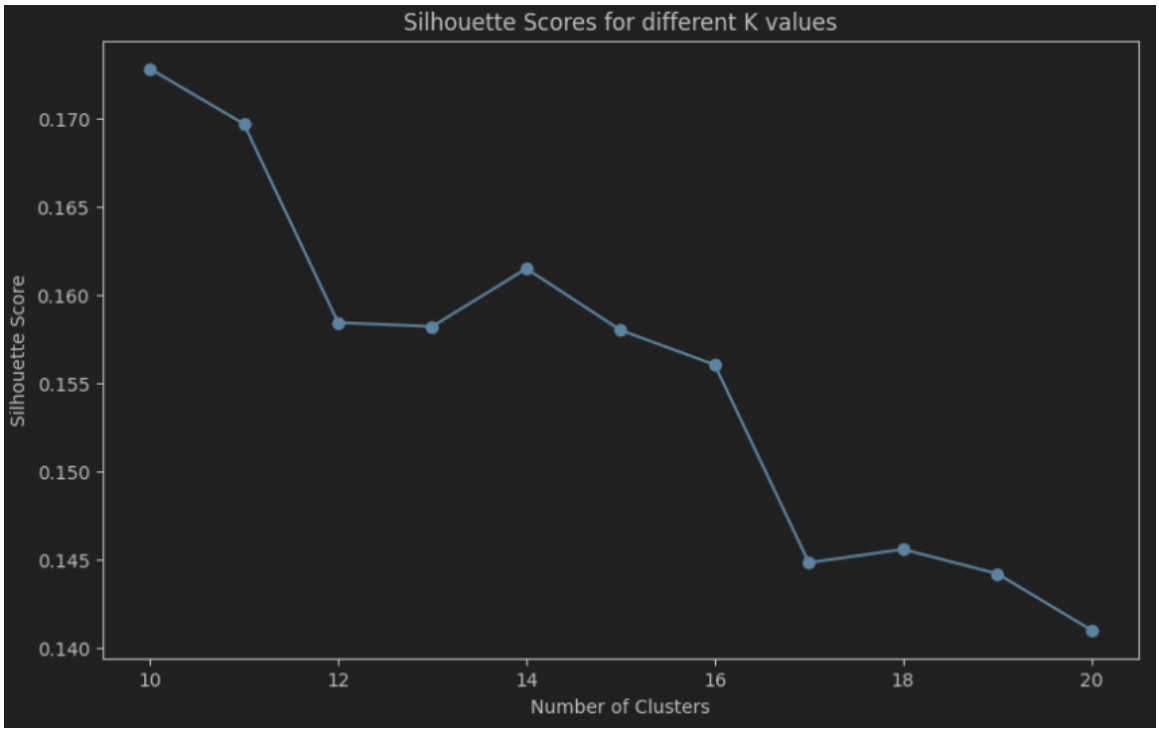
Cluster: 17, Score: 0.1448439210653305

Cluster: 18, Score: 0.14560063183307648

Cluster: 19, Score: 0.14422273635864258

Cluster: 20, Score: 0.1410309076309204

Optimal number of clusters: 10 with a score of 0.17278625071048737



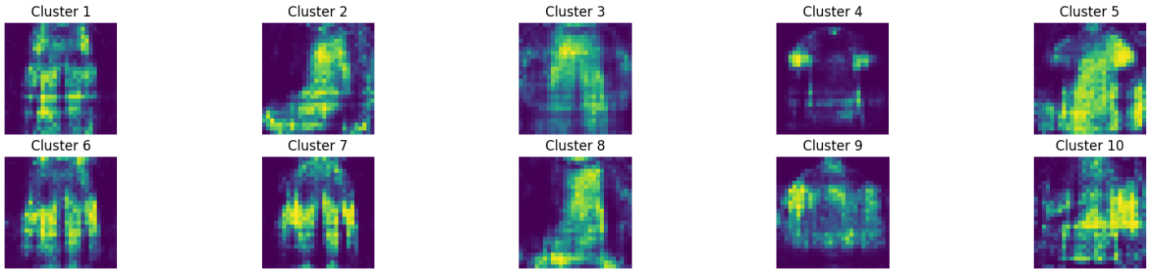
CLUSTER CENTER

Αφότου έχουμε βρεί την βέλτιστη τιμή των clusters θα προχωρήσουμε να βρούμε τα κέντρα τους. Παίρνουμε το k_best που είναι το kmeans με τον βέλτιστο αριθμό (συνήθως το 10 αλλά έχει τύχει και άλλους αριθμούς) και εκτελούμε .cluster_centers το οποίο βρήσκει τα κέντρα και τα αποθηκεύει στο centers. Στη συνέχεια μετατρέπουμε τα κέντρα σε tensors και τα αποκωδικοποιούμε ώστε να έρθουν σε μορφή η οποία μπορεί να εκτυπωθεί ως εικόνα, κάτι το οποίο κάνουμε ώστε να τις δούμε μέσω ενός for-loop.

- PCA

```
#Decode the center and reconstruct the images to their original
dimensions
center_tensor = t.tensor(center, dtype=t.float32)
with t.no_grad():
    reconstructed_center = model.decoder(center_tensor).numpy()

#Plot the centers
plt.figure(figsize=(20, 4))
for j in range(k_best_pos):
    plt.subplot(2, k_best_pos // 2, j + 1)
    plt.imshow(reconstructed_center[j].reshape(28, 28), cmap='viridis')
    plt.title(f'Cluster {j+1}')
    plt.axis('off')
plt.show()
```

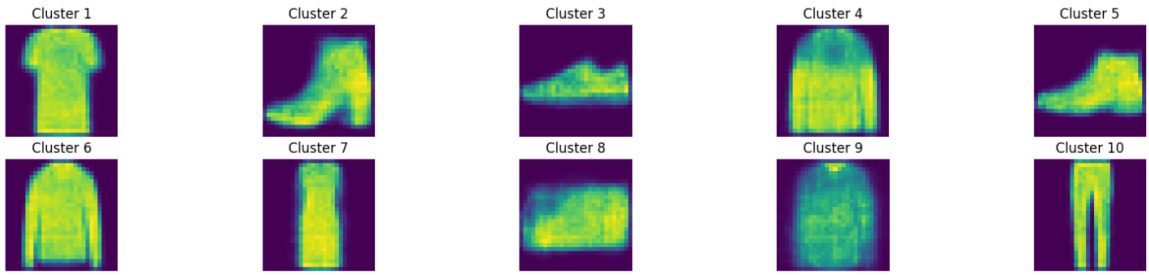
Results:



- ## Autoencoder

```
#Decode the center and reconstruct the images to their original
dimensions
center_tensor = t.tensor(center, dtype=t.float32)
with t.no_grad():
    reconstructed_center = model.decoder(center_tensor).numpy()

#Plot the centers
plt.figure(figsize=(20, 4))
for j in range(k_best_pos):
    plt.subplot(2, k_best_pos // 2, j + 1)
    plt.imshow(reconstructed_center[j].reshape(28, 28), cmap='viridis')
    plt.title(f'Cluster {j+1}')
    plt.axis('off')
plt.show()
```

Results:



PURITY AND F-MEASURE

Purity

Αρχικά κωδικοποιούμε τα labels ώστε να μετατρέψουμε τα δεδομένα του Y_test σε αριθμούς. Σύμφωνα με την καλύτερη τιμή του kmeans που βρήκαμε παραπάνω και τα X_train_pca, encoded_data αντίστοιχα στις δύο μεθόδους μπορούμε να βρούμε τα labels μέσα από τα οποία θα υπολογίσουμε τον συνολικό αριθμό από clusters. Στη συνέχεια αρχικοποιούμε έναν πίνακα majority_classes με βάση τον αριθμό των clusters ο οποίος αρικά παίρνει τιμές 0, σκοπός του θα είναι να κρατήσει την πλειοψηφούσα πραγματική κατηγορία από κάθε cluster.

Σε ένα for-loop βρίσκουμε τους δείκτες που αντιστοιχούν στο κάθε cluster και τα αποθηκεύουμε στο cluster_indices και μέσω αυτούς τους δείκτες βρίσκουμε τα labels των clusters και τα αποθηκεύουμε στο cluster_labels_true. Εφόσον υπάρχουν clusters τότε μέσω .argmax() βρίσκουμε την πλειοψηφούσα πραγματική κατηγορία και προσθέτουμε το άθροισμα των δεδομένων της στο purity. Τέλος διαιρώντας το purity με το Y_test βρίσκουμε τον επιθυμητό μέσω όρο.

- PCA

```
#Encode labels
label_encoder = LabelEncoder()
Y_test_encoded = label_encoder.fit_transform(Y_test)

cluster_labels = kmeans_best.predict(X_train_pca)
num_clusters = len(np.unique(cluster_labels))

#Empty majority class array
majority_classes = np.zeros(num_clusters, dtype=int)

#Purity
purity = 0
for cluster in range(num_clusters):
    cluster_indices = np.where(cluster_labels == cluster)[0]
    cluster_labels_true = Y_test_encoded[cluster_indices]
    if len(cluster_labels_true) > 0:
        majority_class = np.argmax(np.bincount(cluster_labels_true))
        purity += np.sum(cluster_labels_true == majority_class)

purity /= len(Y_test)
print(f'Purity: {purity}')
```

- Autoencoder

```
#Encode labels
label_encoder = LabelEncoder()
Y_test_encoded = label_encoder.fit_transform(Y_test)

cluster_labels = kmeans_best.predict(encoded_data)
num_clusters = len(np.unique(cluster_labels))

#Empty majority class array
majority_classes = np.zeros(num_clusters, dtype=int)

#Purity
purity = 0
for cluster in range(num_clusters):
    cluster_indices = np.where(cluster_labels == cluster)[0]
    cluster_labels_true = Y_test_encoded[cluster_indices]
    if len(cluster_labels_true) > 0:
        majority_class = np.argmax(np.bincount(cluster_labels_true))
        purity += np.sum(cluster_labels_true == majority_class)

purity /= len(Y_test)
print(f'Purity: {purity}')
```

F-Measure

Αρχικά ορίζουμε τη λίστ απου θα αποθηκέυουμε τα αποτελέσματα και τα συνολικά samples. Στη συνέχεια ξεκινάει το loop που διατρέχει κάθε cluster, από τον πρώτο(1) έως και τον τελευταίο(10), βρίσκουμε τα δεδομένα που βρίσκονται μέσα στα clusters και ανακτάμε τα labels τους. Κρατάμε τα True Positives(TP), False Positives(FP) και False Negatives(FN) όπως απαιτείται για τον υπολογισμό του Precision και Recall που αποτελούν επιμέρος συναρτήσεις του F-measure. Υπολογίζουμε το F-measure όπως και στην εκφώνηση και βεβαιωνόμαστε ότι τα μεγαλύτερα clusters επηρεάζουν περισσότερο το F-measure. Τέλος, τυπόνουμε τα αποτελέσματα.

- PCA

```
#We set the loop
f_measures = []
total_samples = len(y_train)
#Loop start for each cluster
for cluster in range(num_clusters):
    cluster_indices = np.where(cluster_labels == cluster)[0]
    cluster_labels_true = y_train[cluster_indices]
    #Calculate TP, FP and FN
    if len(cluster_labels_true) > 0:
        majority_class = np.argmax(np.bincount(cluster_labels_true))
        TP = np.sum(cluster_labels_true == majority_class)
        FP = len(cluster_labels_true) - TP
```

```
        FN = np.sum(y_train == majority_class) - TP

        #Calculate precision recall and f-measure and save final
results
        precision = TP / (TP + FP) if (TP + FP) > 0 else 0
        recall = TP / (TP + FN) if (TP + FN) > 0 else 0
        f_measure = (1 + TP) * (precision * recall) / (TP * precision +
recall) if (TP * precision + recall) > 0 else 0
        weighted_f_measure = f_measure * (len(cluster_labels_true) /
total_samples)
        f_measures.append(weighted_f_measure)

#Calculate total and print final results
f_measure_total = np.sum(f_measures)
print(f'F-measure: {f_measure_total}')
```

- Autoencoder

```
#We set the loop
f_measures = []
total_samples = len(y_train)
#Loop start for each cluster
for cluster in range(num_clusters):
    cluster_indices = np.where(cluster_labels == cluster)[0]
    cluster_labels_true = y_train[cluster_indices]
    #Calculate TP, FP and FN
    if len(cluster_labels_true) > 0:
        majority_class = np.argmax(np.bincount(cluster_labels_true))
        TP = np.sum(cluster_labels_true == majority_class)
        FP = len(cluster_labels_true) - TP
        FN = np.sum(y_train == majority_class) - TP

        #Calculate precision recall and f-measure and save final
results
        precision = TP / (TP + FP) if (TP + FP) > 0 else 0
        recall = TP / (TP + FN) if (TP + FN) > 0 else 0
        f_measure = (1 + TP) * (precision * recall) / (TP * precision +
recall) if (TP * precision + recall) > 0 else 0
        weighted_f_measure = f_measure * (len(cluster_labels_true) /
total_samples)
        f_measures.append(weighted_f_measure)

#Calculate total and print final results
f_measure_total = np.sum(f_measures)
print(f'F-measure: {f_measure_total}')
```

RESULTS:

- PCA

Purity: 0.1171

F-measure: 0.6086535564197009

- Autoencoder

Purity: 0.1127

F-measure: 0.5287939231135816

Όπως παρατηρούμε το Purity και στις 2 περιπτώσεις είναι σχετικά μικρό. Μπορεί να είναι θέμα κώδικα ή και θέμα της Μείωσης Διάστασης. Το F-measure από την άλλη φαίνεται να μην είναι τέλειο αλλά είναι αρκετά καλό από τη στιγμή που είναι πάνω από 0.5 και στις 2 περιπτώσεις.