



ΜΥΕ οο2 Μηχανική Μάθηση

ΣΕΤ ΑΣΚΗΣΕΩΝ 1

Νικόλαος Γιαννόπουλος ΑΜ 5199
Γεώργιος Στρούγγης ΑΜ 5357

Μια σύντομη περιγραφή

Η εργασία αυτή στοχεύει στην μελέτη και κατανόηση αλγορίθμων Μηχανικής Μάθησης αξιοποώντας σύγχρονες μεθόδους όπως το Jupyter Notebook και το PyTorch. Συγκεκριμένα, ζητούνται:

- I. Max-pooling and Array
- II. Nearest Neighbor Classifier
- III. Decision Tree and Random Forest
- IV. Linear SVM Classifier and SVM with RBF kernel
- V. Feed Forward Neural Network
- VI. Convolutional Neural Network

ΒΙΒΛΙΟΘΗΚΕΣ:

Οι βιβλιοθήκες/imports που χρησιμοποιήθηκαν είναι τα εξής:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
import torch as t
import torch.nn as nn
import torch.optim as optim
```

DATA IMPORT:

Πρώτο βήμα πριν μπορέσουμε να εκτελέσουμε οποιαδήποτε μέθοδο είναι να φορτώσουμε τα δεδομένα που θα χρησιμοποιηθούν. Αρχικά φορτώνουμε ξεχωριστά τα δεδομένα εκπεδεύσης και ελέγχου ως train_data και test_data πριν δούμε την μορφή τους μέσω των εντολών .shape και .head. Στη συνέχεια σπάμε τα δεδομένα σε δύο κατηγορίες, τα x_train και x_test περιέχουν τις πληροφορίες για τις εικόνες ενώ τα y_train και y_test τα labels που τους αντιστοιχούν. Όστε 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 καθώ και πως μοιάζουν.

```
#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 from 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())

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

#Pictures of 28x28
x_train = X_train_scaled.reshape((-1, 28, 28))
x_test = X_test_scaled.reshape((-1, 28, 28))

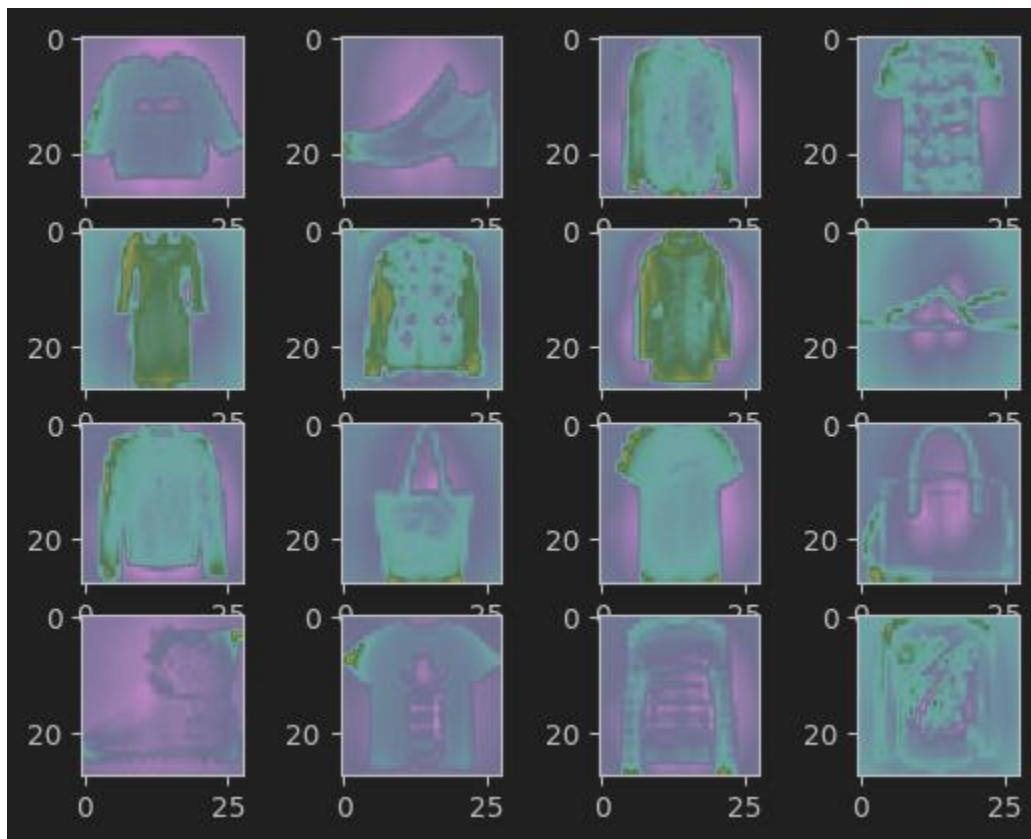
#View Train Data as pictures
for i in range(16):
    plt.subplot(4, 4, i+1)
    plt.imshow(x_train[i], cmap='viridis')
plt.show()
```

OUTPUT AND VISUALIZATION:

Training data shape: (60000, 785)

Testing data shape: (10000, 785)

Αναπαράσταση των δεδομένων εκπαίδευσης:



Max-pooling and Array Transformation

MAX-POOLING:

Για να πραγματοποιηθεί η μέθοδος maxpooling τα οποία παράγει στο πρώτο βήμα πρέπει για αρχή τα δεδομένα μας να μεταμορφωθούν σε tensors πριν η εντολή .unsqueeze τα φέρει στο κατάλληλο μέγεθος. Εφόσον οι tensors μας είναι στο κατάλληλο μέγεθος για να τους δεχτεί η εντολή .max_pool2d, τότε αυτή η εντολή θα εκτελεστεί για παράθυρο διάστασης 4x4. Τέλος, για τις κατάλληλες διαστάσεις εκτελούμε .squeeze ώστε φέρουμε τα δεδομένα μας σε μορφή numpy και να τα εκτυπώσουμε.

```
#Convert the data from numpy to tensors
x_train_tensor = t.tensor(x_train, dtype=t.float32)
x_test_tensor = t.tensor(x_test, dtype=t.float32)

#Reshape the tensors to match the expected input shape
x_train_tensor = x_train_tensor.unsqueeze(1)
x_test_tensor = x_test_tensor.unsqueeze(1)

#Apply max polling to train/test data
x_train_pooled = t.max_pool2d(x_train_tensor, kernel_size=4)
x_test_pooled = t.max_pool2d(x_test_tensor, kernel_size=4)

#Convert the pooled tensors back to numpy arrays
x_train_np = x_train_pooled.squeeze(1).numpy()
x_test_np = x_test_pooled.squeeze(1).numpy()

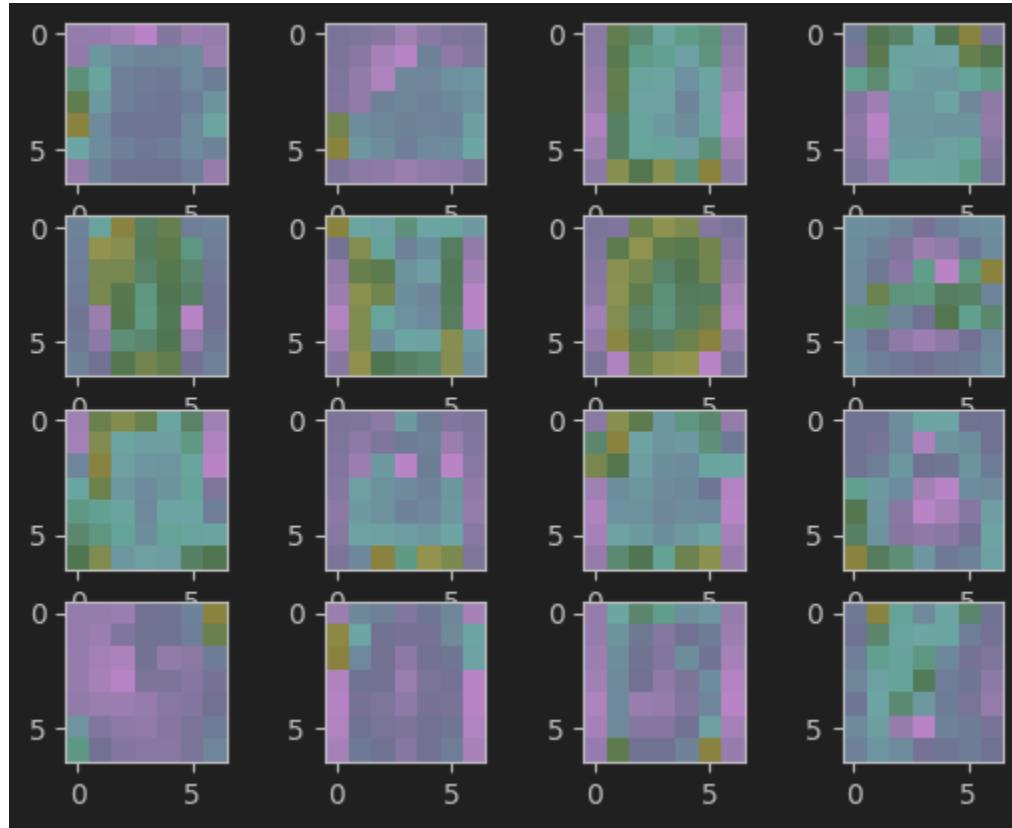
#print to test if all went accordingly
print(f"Train data shape: {x_train_np.shape}")
print(f"Test data shape: {x_test_np.shape}")
#Print of the pooled images to certify the max polling
for i in range(16):
    plt.subplot(4, 4, i+1)
    plt.imshow(x_train_np[i], cmap='viridis')
plt.show()
```

Output for MAX-POOLING:

Train data shape: (60000, 7, 7)

Test data shape: (10000, 7, 7)

Αναπαράσταση των δεδομένων εκπαίδευσης μετά το max-pooling:



ARRAY TRANSFORMATION:

Στη συνέχεια αφότου εκτελεστεί .squeeze για να φέρουμε τα δεδομένα μας πίσω σε μορφή numpy, εκτελούμε reshape από εικόνες της μορφής 7×7 , όπως φαίνεται από τις αντίστοιχες εκτυπώσεις πριν η εντολή .reshape, να καταλήξουμε σε arrays με διανυσματική διάσταση 49.

```
#Array of polled train/test data
x_train_array = x_train_np.reshape(x_train_np.shape[0], -1)
x_test_array = x_test_np.reshape(x_test_np.shape[0], -1)

#Print to test that the conversion to array worked
print(f"Train data array shape: {x_train_array.shape}")
print(f"Test data array shape: {x_test_array.shape}")
```

Output for Array Transformation:

Train data array shape: (60000, 49)

Test data array shape: (10000, 49)

Nearest Neighbor Classifier

Για τις τιμές που δίνονται στην εκφώνηση εκτελείται η εντολή KNeighborsClassifier με ευκλείδεια απόσταση. Μέσω εντολής .fit εκπαιδεύεται με τα δεδομένα στα arrays και τα αντίστοιχα label πριν δοθεί η accuracy. Τέλος εκτυπώνονται τα αποτελέσματα του ταξινομητή κοντινότερων γειτόνων για όλες τις τιμές, συμπεριλαμβανομένω το precision. Βλέπουμε ότι η τιμή του accuracy ανεβαίνει για μεγαλύτερες τιμές του K αλλά είναι για μικρό ποσοστό.

```
#Define different values for K
k_values = [1, 3, 5]

#Loop for different K values, then create classifier, train it and
finally test it
for k in k_values:
    knn_classifier = KNeighborsClassifier(n_neighbors=k,
metric='euclidean')
    knn_classifier.fit(x_train_array, y_train)
    accuracy = knn_classifier.score(x_test_array, y_test)

#Prints of report for each value
    print(f"Accuracy with K={k}: {accuracy}\n")
    print(f"Classification Report: {classification_report(y_test,
y_pred=knn_classifier.predict(x_test_array))}\n")
    print("-----")
```

OUTPUT:

```
Accuracy with K=1: 0.7813
Classification Report: precision recall f1-score support
          0    0.77    0.77    0.77   1000
          1    0.89    0.93    0.91   1000
          2    0.67    0.69    0.68   1000
          3    0.78    0.75    0.76   1000
          4    0.64    0.65    0.64   1000
          5    0.90    0.81    0.85   1000
          6    0.54    0.53    0.53   1000
          7    0.83    0.85    0.84   1000
          8    0.97    0.93    0.95   1000
          9    0.84    0.91    0.87   1000

accuracy      0.78
macro avg     0.78
weighted avg  0.78

-----
```

```
Accuracy with K=3: 0.7998
Classification Report: precision recall f1-score support
          0    0.73    0.83    0.78   1000
          1    0.90    0.93    0.92   1000
          2    0.67    0.74    0.70   1000
          3    0.79    0.77    0.78   1000
          4    0.68    0.67    0.68   1000
          5    0.92    0.80    0.86   1000
          6    0.64    0.52    0.57   1000
          7    0.84    0.90    0.87   1000
          8    0.97    0.92    0.95   1000
          9    0.86    0.92    0.89   1000

accuracy      0.80
macro avg     0.80
weighted avg  0.80

-----
```

```
Accuracy with K=5: 0.8056
Classification Report: precision recall f1-score support
          0    0.76    0.83    0.79   1000
          1    0.89    0.94    0.92   1000
          2    0.69    0.73    0.71   1000
          3    0.78    0.78    0.78   1000
          4    0.69    0.72    0.70   1000
          5    0.92    0.81    0.86   1000
          6    0.65    0.51    0.57   1000
          7    0.84    0.90    0.86   1000
          8    0.97    0.92    0.95   1000
          9    0.86    0.92    0.89   1000

accuracy      0.81
macro avg     0.81
weighted avg  0.81
```

Decision Tree and Random Forest

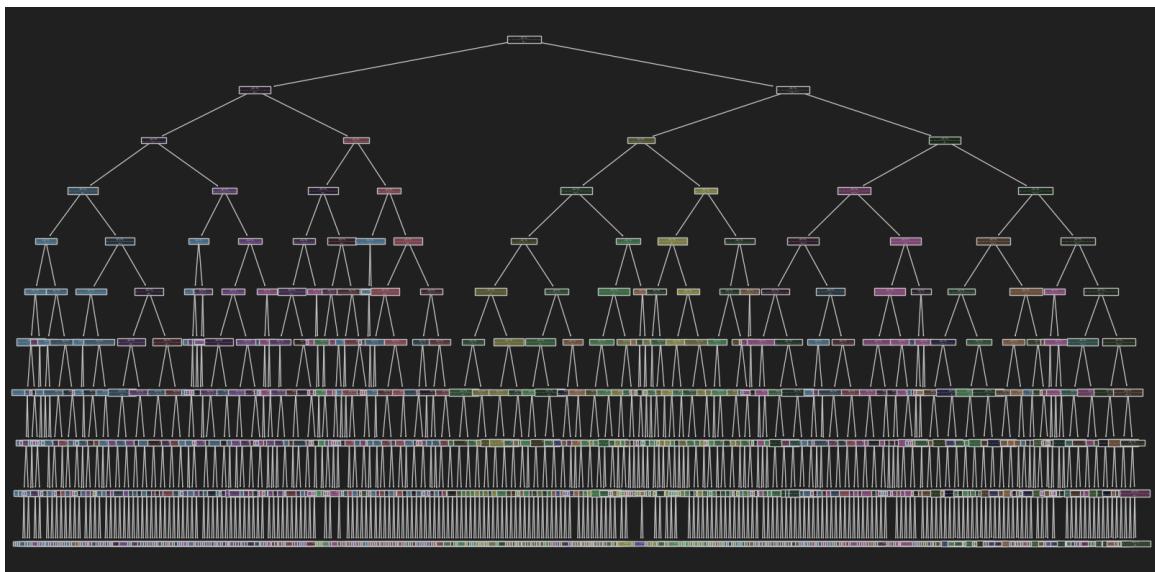
DECISION TREE:

Μέσω εντολής DecisionTreeClassifier και εισόδου ιο δημιουργείται δέντρο απόφασης επιθυμητού μεγέθους το οποίο εκπαιδευεται με τα δεδομένα μας μέσω εντολής .fit(x_train_array, y_train) όπως και παραπάνω. Τέλος εκτυπώνεται το δέντρο το οποίο δημιουργήσαμε με τα χαρακτηριστικά του.

```
#Create and train the Decision Tree with max depth of 10
decision_tree = DecisionTreeClassifier(max_depth=10)
decision_tree.fit(x_train_array, y_train)
#Plot creator for the Decision Tree
print("The Decision Tree:\n")
plt.figure(figsize=(20, 10))
plot_tree(decision_tree, filled=True, feature_names=[str(i) for i in range(49)], class_names=[str(i) for i in range(10)])
plt.show()
```

Output for Decision Tree:

The Decision Tree:



RANDOM FOREST:

Παρομοίως δημιουργείται random forest με 100 estimators μέσω εντολής RandomForestClassifier για τιμή 100 και εκπαιδεύεται μέσω .fit(x_train_array, y_train) για τα κατάλληλα δεδομένα. Σε ένα Random Forest αυτό που μας νοιάζει είναι η επίδοση και εξού ο λόγος που τυπώνεται.

```
#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: {accuracy}\n")
```

Output for Random forest:

Accuracy with Random Forest: 0.8373

Linear SVM Classifier and SVM with RBF Kernel

SUBSET FORMATION:

Εξαιτίας του μεγέθους και της πολυπλοκότητας των δεδομένων απαιτείται μεγάλος χρόνος για την εκτέλεση των ταξινομητών SVM, για αυτό τον λόγο αναγκαζόμαστε να χρησιμοποιήσουμε ένα subset δεδομένων. Ο κώδικας που εφαρμόζουμε χρησιμοποιεί ένα ισορροπημένο subset από δεδομένα εκπαίδευσης διαλέγοντας τυχαία από κάθε κατηγορία. Για κάθε μοναδική ετικέτα ανακατεύει τα αντίστοιχα δείγματα και κρατάει ένα τμήμα τους στο subset. Ως τελευταίο βήμα αυτής της διαδικασίας ορίζει τα δεδομένα και labels που χρησιμοποιούμε ως x_train_subset και y_train_subset.

Μέσω χρήσης MinMaxScaler μεταμορφώνουμε τα δεδομένα για εκπαίδευση και έλεγχο x_train_subset και x_test_array στο κατάλληλο μέγεθος και εκτυπώνουμε τον αριθμό δειγμάτων του subset μας.

```
#Due to the complexity we are forced to use a subset of data
subset = []
for label in np.unique(y_train):
    label_values = np.where(y_train == label)[0]
    np.random.shuffle(label_values)
    num_train = int(len(label_values) * 1)
    subset.extend(label_values[:num_train])
x_train_subset = x_train_array[subset]
y_train_subset = y_train[subset]

#To improve data convergence we will use a scaler to preprocess it
scaler = MinMaxScaler()
x_train_scaled = scaler.fit_transform(x_train_subset)
x_test_scaled = scaler.transform(x_test_array)

print("Number of samples in subsampled training set:",
      x_train_subset.shape)
```

LINEAR SVM CLASSIFIER:

Για κάθε τιμή του C (1, 10, 100) δημουργείται ένα linear SVM classifier μέσω της εντολής SVC για εισόδους linear (που επιλέγει την επιθυμητή μορφή), την τιμή του C και το μέγιστο μέγεθος επαναλήψεων 500. Στη συνέχεια εκπαιδεύεται με τα δεδομένα στο κατάλληλο μέγεθος και τα αντίστοιχα labels μέσω της .fit πριν εκτυπωθεί η accuracy. Παρατηρούμε ότι για το linear SVM classifier ότι, όσο μεγαλώνει η τιμή του C, τόσο μικραίνει και η ακρίβεια, που σημαίνει ότι αυτή η μέθοδος ευνοεί μικρές εισόδους.

```

#Start of loop for C values for Linear and RBF kernel
for c in C_values:
    linear_svm = SVC(kernel='linear', C=c, max_iter=500) #Classifier
for linear SVM
    linear_svm.fit(x_train_scaled, y_train_subset) #Train linear SVM

    #See the accuracy of linear SVM
    accuracy = linear_svm.score(x_test_scaled, y_test)
print(f"Accuracy with Linear SVM with C= {c}: {accuracy}\n")

best_p = 0
best_accuracy = 0

```

Output for Linear SVM Classifier:

Accuracy with Linear SVM with C= 1: 0.5635

Accuracy with Linear SVM with C= 10: 0.4937

Accuracy with Linear SVM with C= 100: 0.3436

RBF KERNEL CLASSIFIER:

Για τις ίδια τιμές του C εκτελείται και ένα SVM με RBF kernel το οποίο παίρνει είσοδο rbf για να τρέξει στην επιθυμητή μορφή. Επιπλέον όμως αυτή η μορφή δέχεται μία ακόμα είσοδο p η οποία παίρνει τις τρεις διαφορετικές τιμές 0.02, 0.1 και 1. Σύμφωνα με την εκφώνηση μας ζητείται να βρούμε για ποιά τιμή από αυτές παρουσιάζεται καλύτερη ακρίβεια, για αυτό για κάθε τιμή του C εκτελείται ένα for loop για τις τρεις τιμές του p. Όπως παραπάνω εκπαιδεύεται και κραταει την μεγαλύτερη ακρίβεια, αντικαθιστώντας την αν κάποιο άλλο p παρουσιάζει μεγαλύτερη τιμή ώστε στο τέλος της λουπας να εκτυπώνει την καλύτερη απόδοση καθώς και για ποιά τιμή του p παρουσιάζεται. Για όλες τις τιμές του C βρίσκουμε ότι η τιμή p = 1 πάντα παρουσιάζει την καλύτερη ακρίβεια, το οποίο μας δείχνει ότι αυτή η μέθοδος ευνοεί μεγάλες εισόδους.

Τέλος, για κάθε εκτέλεση της εντολής SVC μας παρουσιάζεται την ακόλουθη προειδοποίηση το οποίο θα σχεδιάσουμε αν και δεν επηρεάζει το πρόγραμμα: ConvergenceWarning: Solver terminated early (max_iter=500). Consider pre-processing your data with StandardScaler or MinMaxScaler. Στην ουσία απλά μας προειδοποιεί ότι η εντολή εξετελέσθει πριν προλάβει να φτάσει τον μέγιστο αριθμό επαναλήψεων, που σημαίνει ότι η εκτέλεση ήταν επιτυχής.

```

#Start of loop for Parameter values for the RBF kernel
for p in param_values:
    rbf_kernel_svm = SVC(kernel='rbf', gamma=p, C=c, max_iter=500)
    rbf_kernel_svm.fit(x_train_scaled, y_train_subset)
    rbf_accuracy = rbf_kernel_svm.score(x_test_scaled, y_test)
    if rbf_accuracy>best_accuracy:
        best_accuracy = rbf_accuracy
        best_p = p
    print(f"Accuracy with RBF SVM with C= {c} and P={p}:
{rbf_accuracy}\n")
    print("-----")
print(f"The best accuracy belongs to P={best_p}: {best_accuracy}")
print("-----")

```

Output for RBF Kernel Classifier:

Accuracy with RBF SVM with C= 1 and P=0.02: 0.5132

Accuracy with RBF SVM with C= 1 and P=0.1: 0.4914

Accuracy with RBF SVM with C= 1 and P=1: 0.7009

The best accuracy belongs to P=1: 0.7009

Accuracy with RBF SVM with C= 10 and P=0.02: 0.4645

Accuracy with RBF SVM with C= 10 and P=0.1: 0.544

Accuracy with RBF SVM with C= 10 and P=1: 0.7381

The best accuracy belongs to P=1: 0.7381

Accuracy with RBF SVM with C= 100 and P=0.02: 0.5743

Accuracy with RBF SVM with C= 100 and P=0.1: 0.5383

Accuracy with RBF SVM with C= 100 and P=1: 0.6498

The best accuracy belongs to P=1: 0.6498

Feed Forward Neural Network

A SMALL DESCRIPTION:

Για την δημιουργία του νευρωνικού δικτύου(Feed Forward Neural Network) πρώτο πράγμα ήταν ο ορισμός του βάση των οδηγιών που μας δόθηκαν. Στη συνέχεια μετατρέπουμε τα δεδομένα σε tensors, ορίζουμε το μοντέλο και το αριθμό της εισόδου(όσο τα labels).

Εκτυπώνουμε τον αριθμό των παραμέτρων, και πριν το train, οριστικούμε το loss function(criterion) και τον optimizer που βασίζεται στο μοντέλο Adam με learning rate 0.01(προέκυψε ως βέλτιστο μετά από δοκιμές). Τέλος, πραγματοποιύμε τα loops εκπαίδευσης και testing ακολουθώντας παρόμοια παραδείγματα στις διαφάνειες και εκτυπώνουμε τα Accuracy και Loss για τα Train και Test. Υπάρχουν κάποια warnings αλλά τα αγνοούμε. Όσον αφορά τα αποτελέσματα, παρατηρούμε ότι όσο αυξάνονται epochs(loop) τόσο μειώνεται το loss και επομένως αυξάνεται το accuracy.

Υποσημείωση: Όπως θα παρατηρείται στον κώδικα, για το loss χρησιμοποιήθηκε το categorial-cross-entropy(crossEntropyLoss()) αντί για το τετραγωνικό σφάλμα ως συνάρτηση απώλειας(MSELoss()). Αυτό συναίβει διότι στα test με το MSELoss τα αποτελέσματα εμφάνιζαν accuracy άνω του ενός(δεν θα έπερεπε) και loss πολύ υψηλό. Δεν καταφέραμε να επιλύσουμε τα προβλήματα γι' αυτό παρέχουμε και τους 2 κώδικες, και CrossEntropyLoss και MSELoss με τα γραφήματα να ανήκουν στον CrossEntropyLoss.

CROSS ENTROPY LOSS:

```
class FeedForwardNeuralNetwork(nn.Module):
    def __init__(self, input_size, num_classes):
        super(FeedForwardNeuralNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, 100) #First layer with 100
        self.fc2 = nn.Linear(100, 100) #Second layer
        self.fc3 = nn.Linear(100, 50) #Third layer
        self.fc4 = nn.Linear(50, num_classes) #Output layer
        self.relu = nn.LeakyReLU() #LeakyReLU function
        self.softmax = nn.Softmax(dim=1) #Softmax function

    def forward(self, x):
        out = self.relu(self.fc1(x))
        out = self.relu(self.fc2(out))
        out = self.relu(self.fc3(out))
        out = self.fc4(out)
        return self.softmax(out)

#Transform to Tensors
x_train_tensor = t.tensor(x_train_array, dtype=t.float32)
x_test_tensor = t.tensor(x_test_array, dtype=t.float32)
y_train_tensor = t.tensor(y_train, dtype=t.long)
y_test_tensor = t.tensor(y_test, dtype=t.long)

#Define the input size and num classes and then create the
```

```

FeedForwardNeuralNetwork
input_size = x_train_tensor.shape[1]
num_classes = len(t.unique(y_train_tensor))
model = FeedForwardNeuralNetwork(input_size, num_classes)

#Parameters print
num_param = sum(p.numel() for p in model.parameters() if
p.requires_grad)
print(f"Number of parameters: {num_param}\n")

#Loss and optimizer function
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

#Batch and epochs initializer with the loss and accuracy arrays
epochs = 100
batch_size = 50
train_losses = []
train_accuracies = []
test_losses = []
test_accuracies = []

#Define the scalers for features and target variable
scaler_x = MinMaxScaler()
scaler_y = MinMaxScaler()
#Perform grid search
best_score = float('inf')
best_params = {}

#Start of training loop
for epoch in range(epochs):
    epoch_train_loss = 0.0
    epoch_train_correct = 0.0
    epoch_test_loss = 0.0
    epoch_test_correct = 0.0

    model.train()
    for i in range(0, len(x_train_tensor), batch_size):
        batch_x = t.tensor(x_train_tensor[i:i+batch_size],
dtype=t.float32)
        batch_y = t.tensor(y_train_tensor[i:i+batch_size],
dtype=t.long)
        optimizer.zero_grad()
        outputs = model(batch_x) #Trainer definition
        loss = criterion(outputs, batch_y) #Loss definition
        loss.backward()
        optimizer.step()
        epoch_train_loss += loss.item() #Loss counter

        _, predicted = t.max(outputs, 1) #Accuracy counter
        epoch_train_correct += (predicted == batch_y).sum().item()

    train_losses.append(epoch_train_loss / len(x_train_tensor))
    train_accuracies.append(epoch_train_correct / len(x_train_tensor))

    epoch_test_loss = 0.0
    epoch_test_correct = 0.0

    for i in range(0, len(x_test_tensor), batch_size):
        batch_x = t.tensor(x_test_tensor[i:i+batch_size],
dtype=t.float32)
        batch_y = t.tensor(y_test_tensor[i:i+batch_size],
dtype=t.long)
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)
        epoch_test_loss += loss.item()

        _, predicted = t.max(outputs, 1)
        epoch_test_correct += (predicted == batch_y).sum().item()

    test_losses.append(epoch_test_loss / len(x_test_tensor))
    test_accuracies.append(epoch_test_correct / len(x_test_tensor))

    if epoch % 10 == 0:
        print(f'Epoch {epoch}: Train Loss: {epoch_train_loss}, Train Accuracy: {epoch_train_correct / len(x_train_tensor)}, Test Loss: {epoch_test_loss}, Test Accuracy: {epoch_test_correct / len(x_test_tensor)}')

    if epoch > 50 and epoch % 5 == 0:
        if best_score > epoch_train_loss:
            best_score = epoch_train_loss
            best_params = model.state_dict()

model.load_state_dict(best_params)

```

```

#Model evaluation
model.eval()
with t.no_grad():
    for j in range(0, len(x_test_tensor), batch_size):
        batch_test_x = t.tensor(x_test_tensor[j:j+batch_size],
dtype=t.float32)
        batch_test_y = t.tensor(y_test_tensor[j:j+batch_size],
dtype=t.long)
        y_pred = model(batch_test_x) #Trainer for test
        loss_test = criterion(y_pred, batch_test_y) #Loss for test
        epoch_test_loss += loss_test.item() #Loss counter
        _, predicted_test = t.max(y_pred, 1) #Accuracy counter
        epoch_test_correct += (predicted_test ==
batch_test_y).sum().item()

#Data classification to arrays for plot visualization
epoch_train_loss /= len(x_train_tensor)
epoch_train_accuracy = epoch_train_correct / len(x_train_tensor)
epoch_test_loss /= len(x_test_tensor)
epoch_test_accuracy = epoch_test_correct / len(x_test_tensor)

train_losses.append(epoch_train_loss)
train_accuracies.append(epoch_train_accuracy)
test_losses.append(epoch_test_loss)
test_accuracies.append(epoch_test_accuracy)

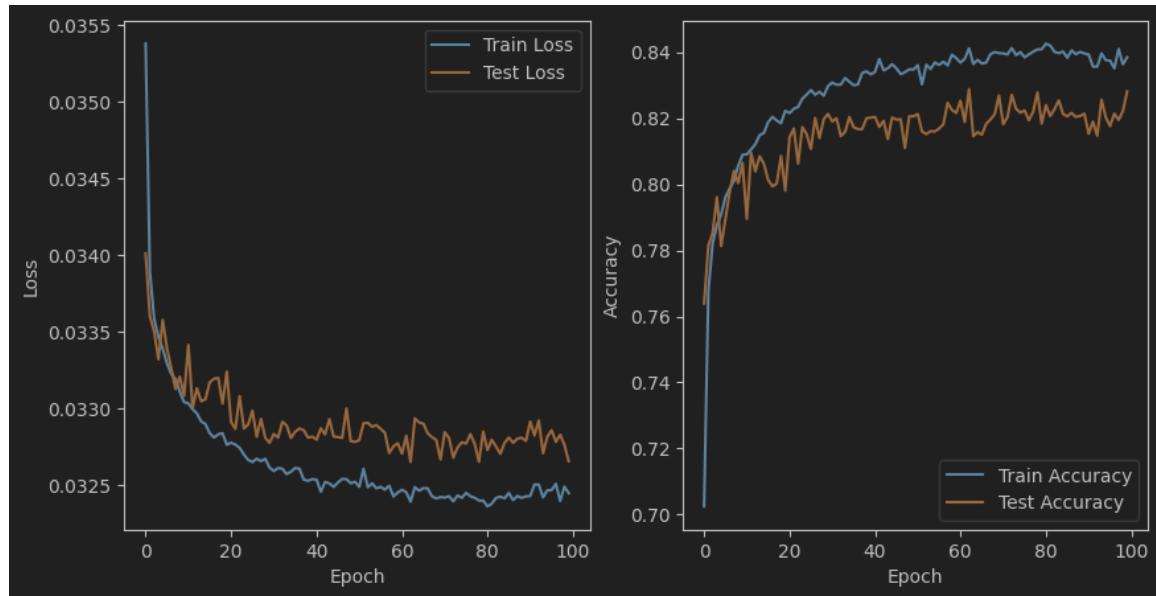
print(f"Epoch [{epoch+1}/{epochs}], Train Loss: {epoch_train_loss:.4f}, Train Acc: {epoch_train_accuracy:.4f}, Test Loss: {epoch_test_loss:.4f}, Test Acc: {epoch_test_accuracy:.4f}")

# Plot loss and accuracy
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(train_accuracies, label='Train Accuracy')
plt.plot(test_accuracies, label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

```

Output Graphs for Cross Entropy Loss:



MSE LOSS(DIDN'T WORK):

```
class FeedForwardNeuralNetworker(nn.Module):
    def __init__(self, input_size, num_classes):
        super(FeedForwardNeuralNetworker, self).__init__()
        self.fc1 = nn.Linear(input_size, 100) #First layer with 100
        self.fc2 = nn.Linear(100, 100) #Second layer
        self.fc3 = nn.Linear(100, 50) #Third layer
        self.fc4 = nn.Linear(50, num_classes) #Output layer
        self.relu = nn.LeakyReLU() #LeakyReLU function
        self.softmax = nn.Softmax(dim=1) #Softmax function

    def forward(self, x):
        out = self.relu(self.fc1(x))
        out = self.relu(self.fc2(out))
        out = self.relu(self.fc3(out))
        out = self.fc4(out)
        return self.softmax(out)

#Define the input_size and num_classes and then create the
FeedForwardNeuralNetworker
input_size = x_train_tensor.shape[1]
num_classes = len(t.unique(y_train_tensor))
model = FeedForwardNeuralNetworker(input_size, num_classes)

num_param = sum(p.numel() for p in model.parameters() if
p.requires_grad)
```

```

print(f"Number of parameters: {num_param}\n")

#Loss and optimizer function
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

#Batch and epochs initializer with the loss and accuracy arrays
epochs = 100
batch_size = 50
train_losses = []
train_accuracies = []
test_losses = []
test_accuracies = []

#Define the scalers for features and target variable
scaler_x = MinMaxScaler()
x_train_scaled = scaler_x.fit_transform(x_train_array)
x_test_scaled = scaler_x.transform(x_test_array)

# Scale output values if needed
scaler_y = MinMaxScaler()
y_train_scaled = scaler_y.fit_transform(y_train.reshape(-1,
1)).flatten()
y_test_scaled = scaler_y.transform(y_test.reshape(-1, 1)).flatten()

# Transform to Tensors
x_train_tensor = t.tensor(x_train_scaled, dtype=t.float32)
x_test_tensor = t.tensor(x_test_scaled, dtype=t.float32)
y_train_tensor = t.tensor(y_train_scaled, dtype=t.float32)
y_test_tensor = t.tensor(y_test_scaled, dtype=t.float32)

#Perform grid search
best_score = float('inf')
best_params = {}

#Start of training loop
#num_batches = len(x_train_tensor) // batch_size
for epoch in range(epochs):
    epoch_train_loss = 0.0
    epoch_train_correct = 0.0
    epoch_test_loss = 0.0
    epoch_test_correct = 0.0

    model.train()
    for i in range(0, len(x_train_tensor), batch_size):
        batch_x = t.tensor(x_train_tensor[i:i+batch_size],
                           dtype=t.float32)
        batch_y = t.tensor(y_train_tensor[i:i+batch_size],
                           dtype=t.float32).view(-1, 1)
        optimizer.zero_grad()
        outputs = model(batch_x)

```

```

loss = criterion(outputs, batch_y)
loss.backward()
optimizer.step()
epoch_train_loss += loss.item()
_, predicted = t.max(outputs, 1)
epoch_train_correct += (predicted == batch_y).sum().item()
#Model evaluation
model.eval()
with t.no_grad():
    for j in range(0, len(x_test_tensor), batch_size):
        batch_test_x = t.tensor(x_test_tensor[j:j+batch_size],
dtype=t.float32)
        batch_test_y = t.tensor(y_test_tensor[j:j+batch_size],
dtype=t.float32).view(-1, 1)
        y_pred = model(batch_test_x)
        loss_test = criterion(y_pred, batch_test_y)
        epoch_test_loss += loss_test.item()
        _, predicted_test = t.max(y_pred, 1)
        epoch_test_correct += (predicted_test ==
batch_test_y).sum().item()

epoch_train_loss /= len(x_train_tensor)
epoch_train_accuracy = epoch_train_correct / len(x_train_tensor)
epoch_test_loss /= len(x_test_tensor)
epoch_test_accuracy = epoch_test_correct / len(x_test_tensor)

train_losses.append(epoch_train_loss)
train_accuracies.append(epoch_train_accuracy)
test_losses.append(epoch_test_loss)
test_accuracies.append(epoch_test_accuracy)

print(f"Epoch [{epoch+1}/{epochs}], Train Loss: {epoch_train_loss:.4f}, Train Acc: {epoch_train_accuracy:.4f}, Test Loss: {epoch_test_loss:.4f}, Test Acc: {epoch_test_accuracy:.4f}")

# Plot loss and accuracy
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(train_accuracies, label='Train Accuracy')
plt.plot(test_accuracies, label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

```

Convolutional Neural Network

A SMALL DESCRIPTION:

Ακολουθώντας τα βήματα που κάναμε στην Feed Forward Neural Network, πράξαμε παρομοίως και εδώ για τον ορισμό του μοντέλου. Αυτή τη φορά όμως δεν έχουμε arrays αλλά εικόνες 28x28. Γι' αυτόν τον λόγω έγινε το reshape και στην συνέχεια η μετατροπή σε tensors μαζί με permute για να έρθουν τα στοιχεία στη σειρά που επιθυμούμε(αλλιώς υπάρχουν προβλήματα με το input). Ακόμα για το πρώτο μοντέλο χρησιμοποιήθηκε φίλτρο 16 ενώ για το δεύτερο .

Στη συνέχεια ορίζουμε το τελικό μοντέλο, το loss function και το optimizer function. Ακολούθως ξεκινάει το training loop ακολουθώντας παρόμοια διαδικασία με το FeedForwardNeuralNetwork και το test evaluation loop τα οποία παράγουν τα loss και accuracy για το καθένα. Τελικώς εκτυπώνουμε τα αποτελέσματα για το loss και accuracy των Train και Test. Στο run εμφανίζονται κάποια warnings που για ακόμη μία φορά τα αγνοούμε καθώς δεν μας επηρεάζουν.

Αλλαγή Φίλτρου: Για την αλλαγή των φίλτρων αρκεί στα filters να επιλεχθεί το φίλτρο στην είσοδο π.χ. 16 το οποίο στο δεύτερο επίπεδο από 16 γίνεται 32 και στο τρίτο από 32 64. Ακόμα, στο fc1 χρειάζεται ο πρώτος αριθμός, αυτός που πολλαπλασιάζεται με το $14^2 \cdot 14$, να είναι η έξοδος του τρίτου επιπέδου, στην προκειμένη περίπτωση 64.

```
class ConvolutionalNeuralNetworkTest(nn.Module):
    def __init__(self):
        super(ConvolutionalNeuralNetworkTest, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16,
kernel_size=3, padding=1) #Input to first layer
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32,
kernel_size=3, padding=1) #First to second layer
        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64,
kernel_size=3, padding=1) #Second to third
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2) #Max Pooling
layer
        self.flatten = nn.Flatten() #Important for fc1
        self.fc1 = nn.Linear(64 * 14 * 14, 100) # Adjusted input size
to match the output of conv2
        self.dropout = nn.Dropout(0.3) #Dropout layer
        self.fc2 = nn.Linear(100, 10) # Output size matches the number
of classes
        self.softmax = nn.Softmax() #Softmax layer

    def forward(self, x):
        out = t.relu(self.conv1(x))
        out = t.relu(self.conv2(out))
        out = self.pool(out)
```

```

        out = t.relu(self.conv3(out))
        out = self.flatten(out)
        out = t.relu(self.fc1(out))
        out = self.dropout(out)
        out = self.fc2(out)
        return self.softmax(out)

#Regulate the shape of the images before the transformation to tensor
x_train_imager = x_train.reshape((-1, 28, 28, 1))
x_test_imager = x_test.reshape((-1, 28, 28, 1))

#Turn to tensors the train and test data with the correct order of data
x_train_torch = t.tensor(x_train_imager, dtype=t.float32).permute(0, 3,
1, 2)
x_test_torch = t.tensor(x_test_imager, dtype=t.float32).permute(0, 3,
1, 2)
y_train_torch = t.tensor(y_train, dtype=t.long)
y_test_torch = t.tensor(y_test, dtype=t.long)

#Model, criterion and optimizer definition
modeler = ConvolutionalNeuralNetworkTest()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(modeler.parameters(), lr=0.001)

#Important definitions
epochs = 100
batch_size = 50
train_losses = []
train_accuracies = []
test_losses = []
test_accuracies = []

#Start of training loop
for epoch in range(epochs):
    epoch_train_loss = 0.0
    epoch_train_correct = 0.0
    epoch_test_loss = 0.0
    epoch_test_correct = 0.0
    modeler.train()
    for i in range(0, len(x_train_torch), batch_size):
        optimizer.zero_grad()
        batch_x = x_train_torch[i:i+batch_size]
        batch_y = y_train_torch[i:i+batch_size]

        outputs = modeler(batch_x) #Train definition
        loss = criterion(outputs, batch_y) #Loss definition
        loss.backward()
        optimizer.step()
        epoch_train_loss += loss.item() #Loss counter
        _, predicted = t.max(outputs, 1) #Accuracy counter
        epoch_train_correct += (predicted == batch_y).sum().item()

    #Start of test loop
    modeler.eval()
    with t.no_grad():

```

```

for j in range(0, len(x_test_torch), batch_size):
    batch_x = x_test_torch[j:j+batch_size]
    batch_y = y_test_torch[j:j+batch_size]
    outputs = modeler(batch_x) #Trainer for test
    loss = criterion(outputs, batch_y) #Loss for test
    epoch_test_loss += loss.item() #Loss counter
    _, predicted = t.max(outputs, 1) #Accuracy counter
    epoch_test_correct += (predicted == batch_y).sum().item()

#Data classification to arrays for plot visualization
epoch_train_loss /= len(x_train_torch)
epoch_train_accuracy = epoch_train_correct / len(x_train_torch)
epoch_test_loss /= len(x_test_torch)
epoch_test_accuracy = epoch_test_correct / len(x_test_torch)

train_losses.append(epoch_train_loss)
train_accuracies.append(epoch_train_accuracy)
test_losses.append(epoch_test_loss)
test_accuracies.append(epoch_test_accuracy)

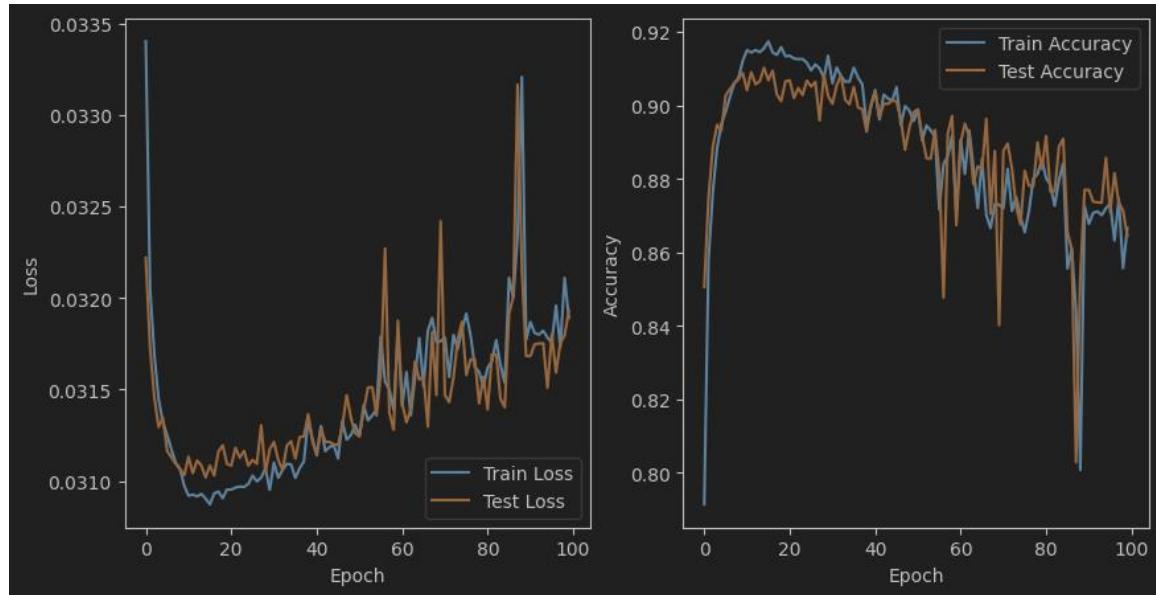
print(f"Epoch [{epoch+1}/{epochs}], Train Loss: {epoch_train_loss:.4f}, Train Acc: {epoch_train_accuracy:.4f}, Test Loss: {epoch_test_loss:.4f}, Test Acc: {epoch_test_accuracy:.4f}")

# Plot loss and accuracy
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

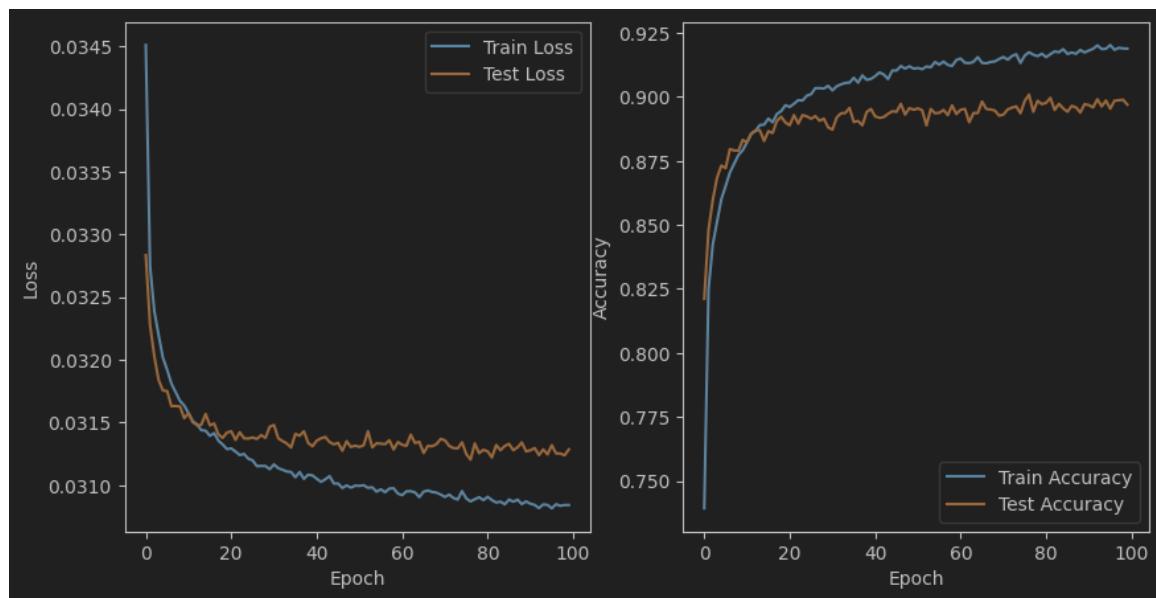
plt.subplot(1, 2, 2)
plt.plot(train_accuracies, label='Train Accuracy')
plt.plot(test_accuracies, label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

```

Output for filter 16:



Output for filter 2:



COMMENT ON RESULTS:

Τα αποτελέσματα από τα δύο φίλτρα είναι ενδιαφέροντα. Αρχικά, με φίλτρο 2 φαίνεται, όχι μόνο πιο γρήγορο αλλά και πιο ικανό να αξιοποιήσει το πρόγραμμα τους πόρους του επεξεργαστή καλύτερα(4.5GHz vs 4.63 GHz). Ακόμη πιο ενδιαφέρον είναι το γεγονός ότι για φίλτρο 16 το loss ανέβαινε με τα epochs μειώνοντας έτσι το accuracy. Κάτι τέτοιο όμως δεν παρατηρείται για φίλτρο 2, το οποίο φαίνεται και πιο σταθερό αλλά και αυξάνει το accuracy αντί να το μειώνει. Συνοπτικά, όσο αυξάνουμε το φίλτρο, τόσο περισσότερο χρόνο παίρνει και τόσο πιο «απρόβλεπτα» αποτελέσματα.