
# *Ψηφιακή Επεξεργασία και Ανάλυση Εικόνας*

*Ακαδημαϊκό Έτος 2023-2024*

## *Εργαστηριακές Ασκήσεις - Μέρος Β*

*ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ*

*ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ*

**Ονοματεπώνυμο:** Φωτάκης Ανδρέας

**ΑΜ:** 1084674

# ΘΕΜΑ 1

Ταξινόμηση Εικόνων

## Μέρος Α

Κατηγοριοποίηση Εικόνων με χρήση Συνελικτικών Νευρωνικών Δικτύων (CNN)

Μετατροπή των εικόνων σε tensors PyTorch (ένας πολυδιάστατος πίνακας). Εφαρμόζεται κατά την εγκατάσταση των εικόνων στο επόμενο βήμα.

```
transform = transforms.Compose([transforms.ToTensor()])
```

Εγκατάσταση του MNIST trainset και testset μέσω του datasets.MNIST που είναι μια συνάρτηση από τη βιβλιοθήκη torchvision που φορτώνει το σύνολο δεδομένων MNIST.

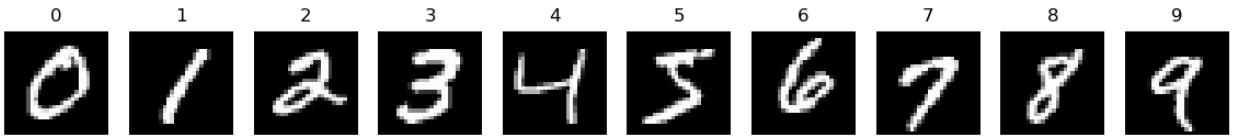
```
mnist_trainset = datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
mnist_testset = datasets.MNIST(root='./data', train=False, download=True,
transform=transform)
```

train=False υποδεικνύει ότι θέλουμε να φορτώσουμε το σύνολο δεδομένων test.
train=True υποδεικνύει ότι θέλουμε να φορτώσουμε το σύνολο δεδομένων training.

### Ερώτημα 1

Απεικόνιση ενός δείγματος από κάθε κλάση σε ένα plot. Τυπώνουμε την εικόνα και ως τίτλο το label της εικόνας.

```
fig, axes = plt.subplots(1, 10, figsize=(20, 2))
for i in range(10):
    img, label = mnist_trainset.data[mnist_trainset.targets == i][0], i
    axes[i].imshow(img, cmap='gray')
    axes[i].set_title(label)
    axes[i].axis('off')
plt.show()
```



## Ερώτημα 2

Χωρίζουμε το σύνολο δεδομένων εκπαίδευσης MNIST σε υποσύνολα εκπαίδευσης και επικύρωσης σε αναλογία 80%-20%. Συγκεκριμένα, χρησιμοποιούμε τη συνάρτηση random_split από το torch.utils.data για να χωρίσουμε τυχαία το σύνολο δεδομένων σε αυτά τα δύο υποσύνολα, αφού πρώτα υπολογίσαμε τα μεγέθη για τα σύνολα εκπαίδευσης και επικύρωσης με βάση τον συνολικό αριθμό δειγμάτων στο σύνολο δεδομένων εκπαίδευσης MNIST.

```
train_size = int(0.8 * len(mnist_trainset))
val_size = len(mnist_trainset) - train_size
mnist_trainset, mnist_valset = torch.utils.data.random_split(mnist_trainset,
[train_size, val_size])
```

## Ερώτημα 3

Υλοποίηση της αρχιτεκτονικής του Συνελικτικού Νευρωνικού Δικτύου:

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5, padding=0)
        self.conv2 = nn.Conv2d(6, 12, kernel_size=5, padding=0
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(12 * 4 * 4, 128
        self.fc2 = nn.Linear(128, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.pool(x)
        x = self.relu(self.conv2(x))
        x = self.pool(x)
        x = x.view(-1, 12 * 4 * 4)
        features = self.relu(self.fc1(x))
        logits = self.fc2(features)
        return logits, features
```

Η αρχιτεκτονική CNN στον κώδικα αντικατοπτρίζει την εικόνα. Το πρώτο συνελικτικό στρώμα (conv1) χρησιμοποιεί 6 φίλτρα μεγέθους 5x5, ακολουθούμενο από ένα στρώμα ενεργοποίησης ReLU και ένα στρώμα max pooling 2x2 για τη μείωση των χωρικών διαστάσεων. Αυτή η διαδικασία επαναλαμβάνεται με το δεύτερο στρώμα συνελικτικής ανάλυσης (conv2), το οποίο χρησιμοποιεί 12 φίλτρα μεγέθους 5x5, ακολουθούμενο από άλλο ένα στρώμα ενεργοποίησης ReLU και ένα αντίστοιχο max pooling. Στη συνέχεια, τα feature maps γίνονται flatten για τα πλήρως συνδεδεμένα στρώματα, όπου το πρώτο στρώμα (fc1) εξάγει 128 χαρακτηριστικά, ακολουθούμενο από ένα τελικό στρώμα (fc2) που εξάγει 10 class scores. Κάθε βήμα αντιστοιχεί στην εικόνα, εξασφαλίζοντας αποτελεσματική εξαγωγή χαρακτηριστικών και ταξινόμηση για το σύνολο δεδομένων MNIST.

```
model = CNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

**α]** Χρησιμοποιείται το DataLoader από το PyTorch για να δημιουργήσουμε τα mini-batch. Η παράμετρος batch_size καθορίζει το μέγεθος κάθε mini-batch. Θέτοντας shuffle=True, εξασφαλίζουμε ότι τα δεδομένα ανακατεύονται τυχαία πριν χωριστούν σε mini-batch, γεγονός που βοηθάει στη στοχαστική φύση της SGD.

```
train_dataloader = torch.utils.data.DataLoader(mnist_trainset, batch_size=32,
shuffle=True)
val_dataloader = torch.utils.data.DataLoader(mnist_valset, batch_size=32,
shuffle=False)
test_dataloader = torch.utils.data.DataLoader(mnist_testset, batch_size=32,
shuffle=False)
```

**b]** Στην συνάρτηση train_model, για κάθε mini-batch, εκτελούμε ένα forward pass για τον υπολογισμό της εξόδου του δικτύου και στη συνέχεια υπολογίζουμε την απώλεια. Η κλήση loss.backward() υπολογίζει τις κλίσεις για όλες τις παραμέτρους του μοντέλου. Τέλος, η optimizer.step() ενημερώνει τις παραμέτρους χρησιμοποιώντας αυτές τις κλίσεις.

```
for images, labels in train_loader:
            optimizer.zero_grad()           # Clear previous gradients
            outputs, _ = model(images)      # Forward pass
            loss = criterion(outputs, labels)  # Compute loss
            loss.backward()                 # Backward pass to compute gradients
            optimizer.step()                # Update parameters using gradients
            running_train_loss += loss.item()
```

**c]** Αυτό το βήμα περιλαμβάνεται στο προηγούμενο μπλοκ κώδικα με το optimizer.step().

**d]** Το running_train_loss που φαίνεται στον κώδικα του ερωτήματος (b) συσσωρεύει τη συνολική απώλεια training για όλα τα mini-batches στο κάθε epoch.

Μετά τη φάση εκπαίδευσης για ένα epoch, το μοντέλο τίθεται σε λειτουργία αξιολόγησης με τη μέθοδο model.eval().

Στη λειτουργία αξιολόγησης, οι παράμετροι του μοντέλου δεν ενημερώνονται και οι κλίσεις δεν υπολογίζονται.

Για κάθε mini-batch στο val_loader, υπολογίζονται οι προβλέψεις του μοντέλου και υπολογίζεται η απώλεια. Αυτό γίνεται στο βρόχο:

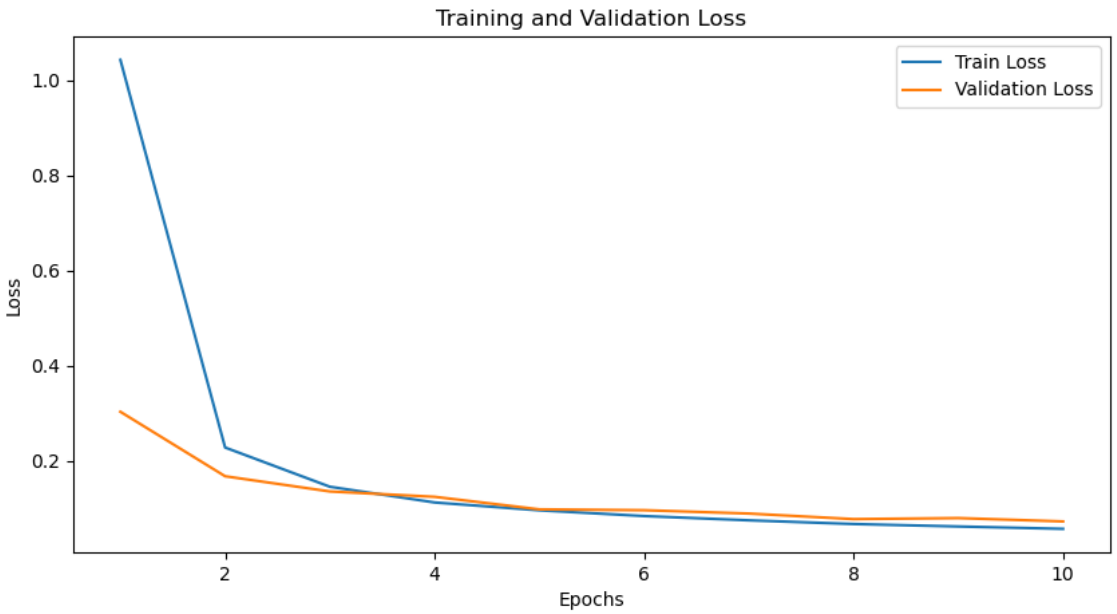
```
 with torch.no_grad():
            for images, labels in val_loader:
                outputs, _ = model(images)
                loss = criterion(outputs, labels)
                running_val_loss += loss.item()
```

Αντίστοιχα το running_val_loss συσσωρεύει τη συνολική απώλεια validation για για όλα τα mini-batches στο κάθε epoch.

Το average training loss και average validation loss για το epoch υπολογίζεται διαιρώντας το running_train_loss και running_val_loss αντίστοιχα με τον αριθμό των mini-batches:

```
epoch_train_loss = running_train_loss / len(train_loader)
epoch_val_loss = running_val_loss / len(val_loader)
train_loss.append(epoch_train_loss)
val_loss.append(epoch_val_loss)
```

Στην συνέχεια γίνονται plot οι τιμές των train_loss και val_loss που επιστρέφονται από την συνάρτηση train_model:




```
Epoch 1/10, Train Loss: 1.0435, Val Loss: 0.3026
Epoch 2/10, Train Loss: 0.2274, Val Loss: 0.1669
Epoch 3/10, Train Loss: 0.1448, Val Loss: 0.1348
Epoch 4/10, Train Loss: 0.1117, Val Loss: 0.1236
Epoch 5/10, Train Loss: 0.0951, Val Loss: 0.0972
Epoch 6/10, Train Loss: 0.0831, Val Loss: 0.0954
Epoch 7/10, Train Loss: 0.0741, Val Loss: 0.0884
Epoch 8/10, Train Loss: 0.0663, Val Loss: 0.0767
Epoch 9/10, Train Loss: 0.0611, Val Loss: 0.0788
Epoch 10/10, Train Loss: 0.0561, Val Loss: 0.0718
```

**e]** Κατά τη διάρκεια της εκπαίδευσης, μετά από κάθε epoch, εάν η απώλεια επικύρωσης είναι μικρότερη από την προηγούμενη καλύτερη απώλεια επικύρωσης, αποθηκεύουμε την κατάσταση του μοντέλου. Αυτό διασφαλίζει ότι διατηρούμε το μοντέλο με την καλύτερη απόδοση.

```
  if epoch_val_loss < best_val_loss:
            best_val_loss = epoch_val_loss
            torch.save(model.state_dict(), 'best_model.pth')
```

Συνήθως, το epoch_val_loss θεωρείται καλύτερο μέτρο απόδοσης για το μοντέλο, καθώς αντιπροσωπεύει την απόδοση του μοντέλου σε δεδομένα που δεν έχουν χρησιμοποιηθεί κατά την εκπαίδευση. Έτσι, το χαμηλό validation loss σημαίνει ότι το μοντέλο γενικεύει καλά και μπορεί να κάνει καλές προβλέψεις σε νέα δεδομένα.

**f]** Τα αποτελέσματα της διαδικασίας εκπαίδευσης δείχνουν ότι τόσο το σφάλμα εκπαίδευσης όσο και το σφάλμα επικύρωσης μειώνονται σταθερά κατά τη διάρκεια των epoch, υποδεικνύοντας ότι το μοντέλο μαθαίνει και γενικεύει καλά. Στην αρχή, παρατηρούμε μια γρήγορη μείωση και των δύο σφαλμάτων, ειδικά κατά τα πρώτα τρία epoch, κάτι που είναι ενδεικτικό της αποτελεσματικής αρχικής μάθησης. Από το τέταρτο έως το έβδομο epoch, τα σφάλματα συνεχίζουν να μειώνονται, αλλά με πιο αργό ρυθμό, υποδεικνύοντας ότι το μοντέλο πλησιάζει στη σύγκλιση. Από το όγδοη έως το δέκατο epoch, τα σφάλματα σταθεροποιούνται, με το σφάλμα εκπαίδευσης να φτάνει το 0.0561 και το σφάλμα επικύρωσης το 0.0718. Η μικρή και σταθερή διαφορά μεταξύ των δύο σφαλμάτων δείχνει ότι το μοντέλο δεν υπερεκπαιδεύεται και έχει καλή γενίκευση. Συνολικά, θεωρούμε ότι η διαδικασία εκπαίδευσης ολοκληρώθηκε επιτυχώς μεταξύ του όγδοου και δέκατου epoch, καθώς τα σφάλματα έχουν σταθεροποιηθεί και η περαιτέρω μείωση τους είναι ελάχιστη.

## Ερώτημα 4

Αρχικά φορτώνουμε τις παραμέτρους του μοντέλου με τις καλύτερες επιδόσεις (αποθηκευμένες κατά την εκπαίδευση) από ένα αρχείο με όνομα 'best_model.pth'. Για το σκοπό αυτό χρησιμοποιείται η συνάρτηση load_state_dict. Μετά τη φόρτωση του μοντέλου, η model.eval() θέτει το μοντέλο σε κατάσταση αξιολόγησης.

```
model.load_state_dict(torch.load('best_model.pth'))
model.eval()
```

Στην συνέχεια, αρχικοποιείται μια κενή λίστα results και ένας μετρητής total για την αποθήκευση των σωστά προβλεπόμενων εικόνων και του αριθμού των σωστών προβλέψεων, αντίστοιχα. Ο κώδικας κάνει iterate το test dataset χρησιμοποιώντας τον test_dataloader και για κάθε παρτίδα εικόνων και ετικετών ελέγχει αν υπάρχει διαθέσιμη GPU με δυνατότητα CUDA. Εάν ναι, μεταφέρει τις εικόνες και τις ετικέτες στην GPU. Το μοντέλο κάνει προβλέψεις χρησιμοποιώντας την model(image) που χρησιμοποιεί το εκπαιδευμένο CNN μας  και οι προβλεπόμενες πιθανότητες μετατρέπονται σε class scores χρησιμοποιώντας τη συνάρτηση softmax. Για κάθε πρόβλεψη, ο κώδικας ελέγχει αν η προβλεπόμενη ετικέτα ταιριάζει με την πραγματική ετικέτα και ενημερώνει αναλόγως τον συνολικό αριθμό και τον κατάλογο

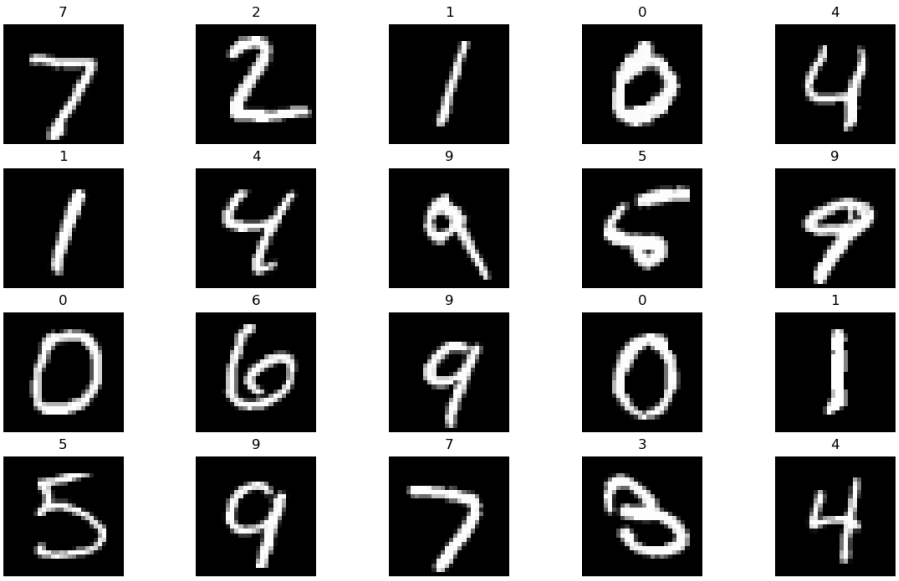
αποτελεσμάτων. Τέλος, υπολογίζει και εκτυπώνει την ακρίβεια δοκιμής διαιρώντας τον αριθμό των σωστών προβλέψεων με τον συνολικό αριθμό των δειγμάτων δοκιμής.

```
results = list()
total = 0

for itr, (image, label) in enumerate(test_dataloader):
    if torch.cuda.is_available():
        image = image.cuda()
        label = label.cuda()

    pred, _ = model(image)
    pred = torch.nn.functional.softmax(pred, dim=1)

    for i, p in enumerate(pred):
        if label[i] == torch.max(p.data, 0)[1]:
            total += 1
            results.append((image[i], torch.max(p.data, 0)[1]))

test_accuracy = total / len(mnist_testset)
print('Test accuracy {:.8f}'.format(test_accuracy))
```

Η ακρίβεια του μοντέλου που υπολογίστηκε:

```
Test accuracy 0.98310000
```

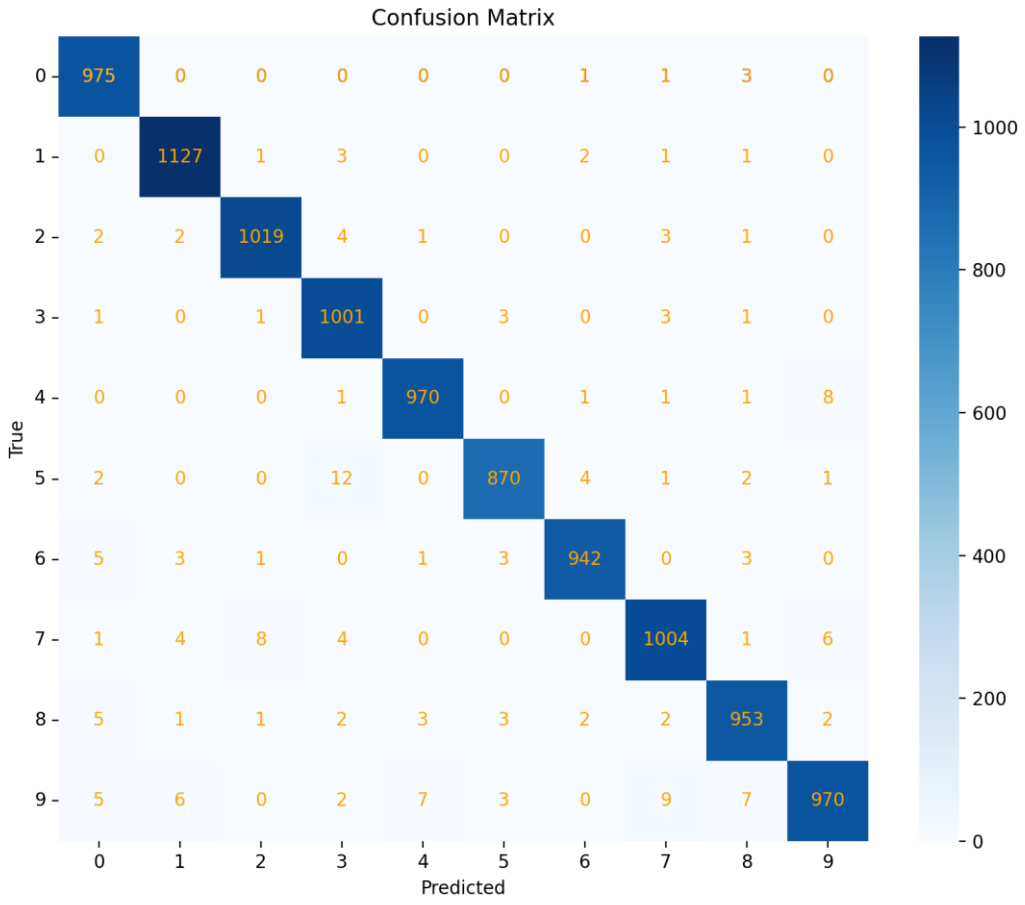
Γίνεται plot για τις πρώτες 20 σωστά προβλεπόμενες εικόνες από το σύνολο δεδομένων δοκιμής μαζί με τις προβλεπόμενες ετικέτες τους.



Με αντίστοιχο τρόπο, με αυτόν στον παραπάνω κώδικα, στην συνέχεια για τον υπολογισμό του confusion matrix γίνονται προβλέψεις χρησιμοποιώντας το μοντέλο και καθορίζεται η προβλεπόμενη κλάση για κάθε εικόνα χρησιμοποιώντας το torch.max. Οι αληθείς και οι προβλεπόμενες ετικέτες μετατρέπονται στη συνέχεια σε πίνακες numpy και προσαρτώνται στις αντίστοιχες λίστες τους. Οι λίστες αυτές περνάνε στη συνάρτηση confusion_matrix από την ενότητα sklearn.metrics που υπολογίζει τον πίνακα confusion matrix. Ο πίνακας έχει την παρακάτω μορφή με print:

```
Confusion Matrix:
[[ 968    1    0    0    1    1    2    1    3    3]
 [   0 1132    1    1    0    1    0    0    0    0]
 [   2    6 1010    5    0    0    0    8    1    0]
 [   1    0    0  998    0    5    0    3    3    0]
 [   0    0    2    0  969    0    2    1    1    7]
 [   1    0    0    8    0  876    4    1    2    0]
 [   3    4    0    1    1    4  944    0    1    0]
 [   0    7    7    3    1    0    0 1005    2    3]
 [   2    0    2    8    3    6    0    3  948    2]
 [   0    7    0    5    7    2    0    6    1  981]]
```

Για να γίνει plot ο πίνακας χρησιμοποιείται η συνάρτηση plot_confusion_matrix που απεικονίζει τον πίνακα σύγχυσης χρησιμοποιώντας ένα heatmap, μέσω του το seaborn.heatmap, και τις τιμές σε κάθε κελί:



## Ερώτημα 5

Για κάθε proportion, υπολογίζεται το μέγεθος του υποσυνόλου (subset_size) πολλαπλασιάζοντας το proportion με total size του training dataset. Στη συνέχεια, επιλέγει τυχαία ένα υποσύνολο των δεδομένων εκπαίδευσης. Αυτό το υποσύνολο δημιουργείται με τη χρήση του torch.utils.data.Subset, επιτρέποντας στο μοντέλο να εκπαιδευτεί σε διαφορετικά τμήματα του συνόλου δεδομένων για να αναλυθεί πώς το μέγεθος των δεδομένων εκπαίδευσης επηρεάζει την απόδοση.

```
proportions = [0.05, 0.1, 0.5, 1.0]
for prop in proportions:
    print(f'\nTraining with {prop * 100}% of training data...')
    subset_size = int(prop * total_train_size)
    # Randomly select subset from mnist_trainset
    subset_indices = np.random.choice(total_train_size, subset_size,
replace=False)
    subset_trainset = torch.utils.data.Subset(mnist_trainset, subset_indices)
```

Στη συνέχεια χωρίζεται το υποσύνολο των δεδομένων εκπαίδευσης σε σύνολα εκπαίδευσης και επικύρωσης. Συγκεκριμένα, το 80% του υποσυνόλου χρησιμοποιείται για εκπαίδευση (train_size) και το υπόλοιπο 20% για επικύρωση (val_size), όπως έγινε και προηγουμένως. Χρησιμοποιείται το DataLoader από το PyTorch για να δημιουργήσουμε τα mini-batch.

```
train_size = int(0.8 * subset_size)  # 80% for training
    val_size = subset_size - train_size  # 20% for validation

    subset_trainset, subset_valset =
torch.utils.data.random_split(subset_trainset, [train_size, val_size])
    subset_train_loader = torch.utils.data.DataLoader(subset_trainset,
batch_size=32, shuffle=True)
    subset_val_loader = torch.utils.data.DataLoader(subset_valset,
batch_size=32, shuffle=False)
```

Η συνάρτηση train_model (όπως εξηγήθηκε προηγουμένως) καλείται για να εκπαιδεύσει το μοντέλο στο υποσύνολο των δεδομένων εκπαίδευσης για 10 epoch. Οι απώλειες εκπαίδευσης και επικύρωσης για κάθε αναλογία προσαρτώνται στις αντίστοιχες λίστες (all_train_losses και all_val_losses). Στη συνέχεια φορτώνονται οι καλύτερες παράμετροι του μοντέλου που αποθηκεύτηκαν κατά την εκπαίδευση και το μοντέλο τίθεται σε κατάσταση αξιολόγησης.

```
    train_loss, val_loss = train_model(model, subset_train_loader,
subset_val_loader, criterion, optimizer, epochs=10)
    all_train_losses.append(train_loss)
    all_val_losses.append(val_loss)
    model.load_state_dict(torch.load('best_model.pth'))
    model.eval()
```

Αξιολογείται η απόδοση του μοντέλου στο σύνολο δεδομένων δοκιμής για τον υπολογισμό των τιμών που απαιτούνται για να υπολογίσουμε το test_accuracy και το confusion_matrix.

Στην συνέχεια τυπώνεται η τιμή του test_accuracy , η συνάρτηση confusion_matrix από το sklearn.metrics χρησιμοποιείται για τη δημιουργία του πίνακα και η plot_confusion_matrix καλείται για την εμφάνισή του ως χάρτη θερμότητας.

Στην συνέχεια αδειάζουν οι λίστες για να μπούν οι τιμές του επόμενου proportion.

```
    test_correct = 0
    with torch.no_grad():
        for images, labels in test_dataloader:
            outputs, _ = model(images)
            _, preds = torch.max(outputs, 1)
            test_correct += (preds == labels).sum().item()

            true_labels.extend(labels.cpu().numpy())
            pred_labels.extend(preds.cpu().numpy())

    test_accuracy = test_correct / len(mnist_testset)
    results[prop] = test_accuracy
    print(f'Test Accuracy with {prop * 100}% of training data:
{test_accuracy:.4f}')

    conf_matrix = confusion_matrix(true_labels, pred_labels)
    plot_confusion_matrix(conf_matrix)
    print(conf_matrix)

    true_labels.clear()
    pred_labels.clear()
```

```
Training with 5.0% of training data...
2400
1920
480
Epoch 1/10, Train Loss: 2.2949, Val Loss: 2.2911
Epoch 2/10, Train Loss: 2.2868, Val Loss: 2.2835
Epoch 3/10, Train Loss: 2.2772, Val Loss: 2.2726
Epoch 4/10, Train Loss: 2.2621, Val Loss: 2.2547
Epoch 5/10, Train Loss: 2.2346, Val Loss: 2.2188
Epoch 6/10, Train Loss: 2.1740, Val Loss: 2.1294
Epoch 7/10, Train Loss: 2.0155, Val Loss: 1.8924
Epoch 8/10, Train Loss: 1.6423, Val Loss: 1.4767
Epoch 9/10, Train Loss: 1.1853, Val Loss: 1.0150
Epoch 10/10, Train Loss: 0.8860, Val Loss: 0.8014
Test Accuracy with 5.0% of training data: 0.7824
```

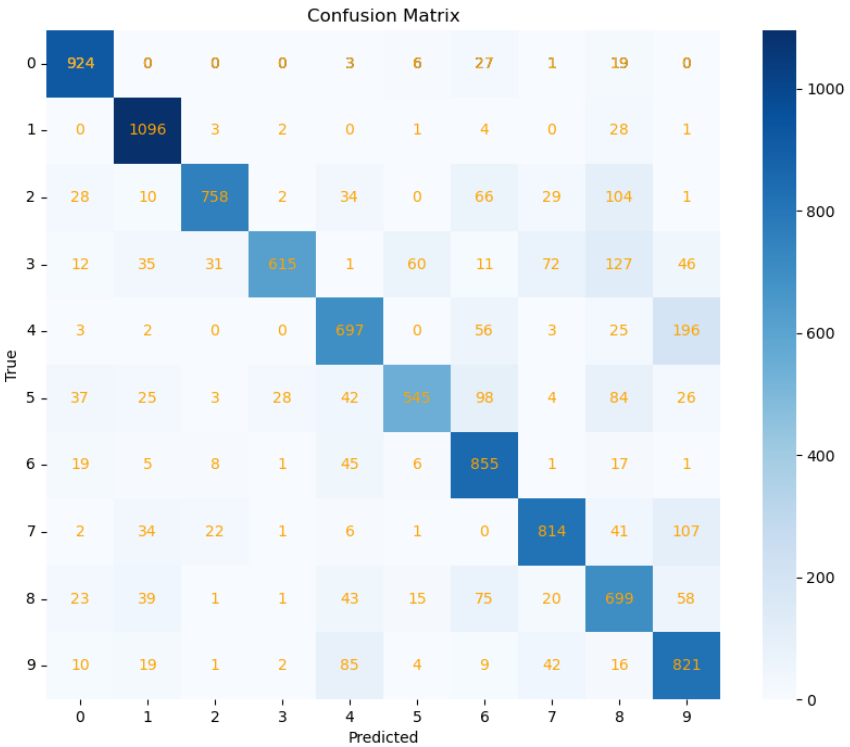
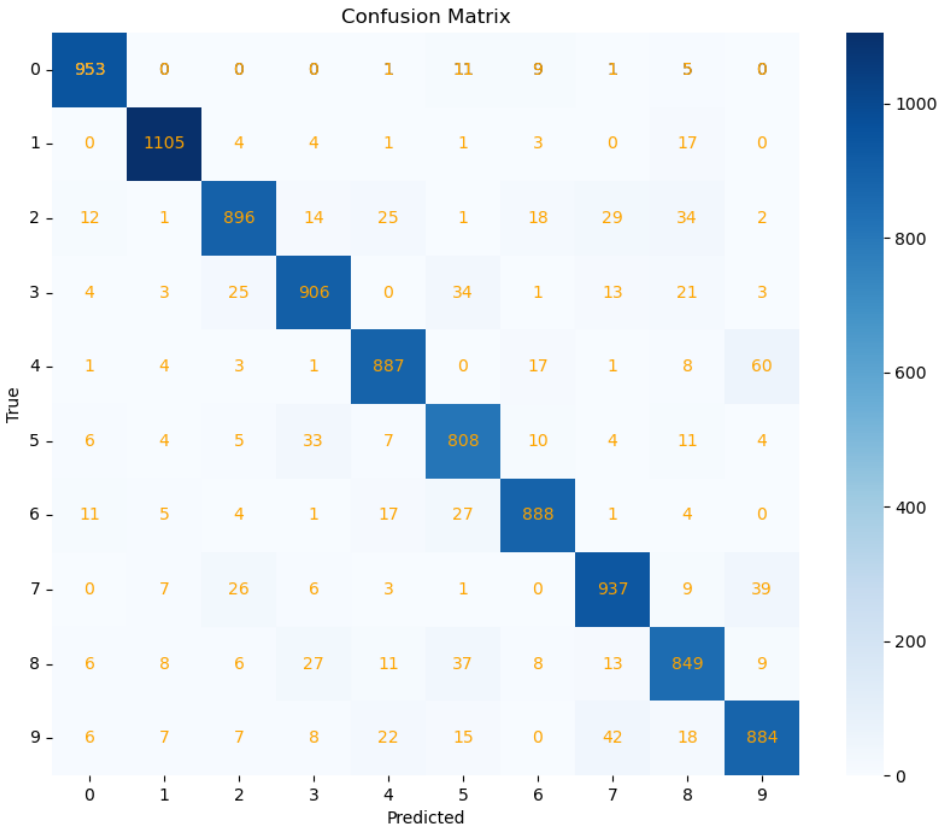



**Figure 1:** χρησιμοποιώντας το 5% των 60.000 δειγμάτων εκπαίδευσης

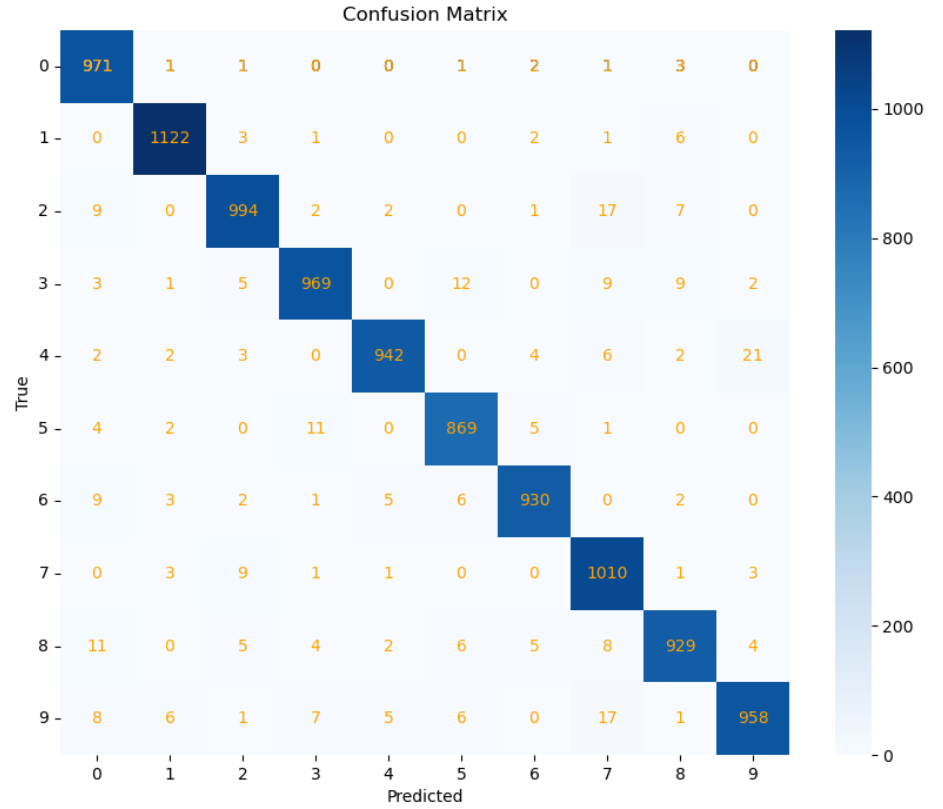
```
Training with 10.0% of training data...
4800
3840
960
Epoch 1/10, Train Loss: 2.3003, Val Loss: 2.2964
Epoch 2/10, Train Loss: 2.2892, Val Loss: 2.2841
Epoch 3/10, Train Loss: 2.2666, Val Loss: 2.2510
Epoch 4/10, Train Loss: 2.1902, Val Loss: 2.1063
Epoch 5/10, Train Loss: 1.7291, Val Loss: 1.2104
Epoch 6/10, Train Loss: 0.8237, Val Loss: 0.6049
Epoch 7/10, Train Loss: 0.5146, Val Loss: 0.4456
Epoch 8/10, Train Loss: 0.4125, Val Loss: 0.3843
Epoch 9/10, Train Loss: 0.3584, Val Loss: 0.3642
Epoch 10/10, Train Loss: 0.3139, Val Loss: 0.2877
Test Accuracy with 10.0% of training data: 0.9113
```

**Figure 2:** χρησιμοποιώντας το 10% των 60.000 δειγμάτων εκπαίδευσης



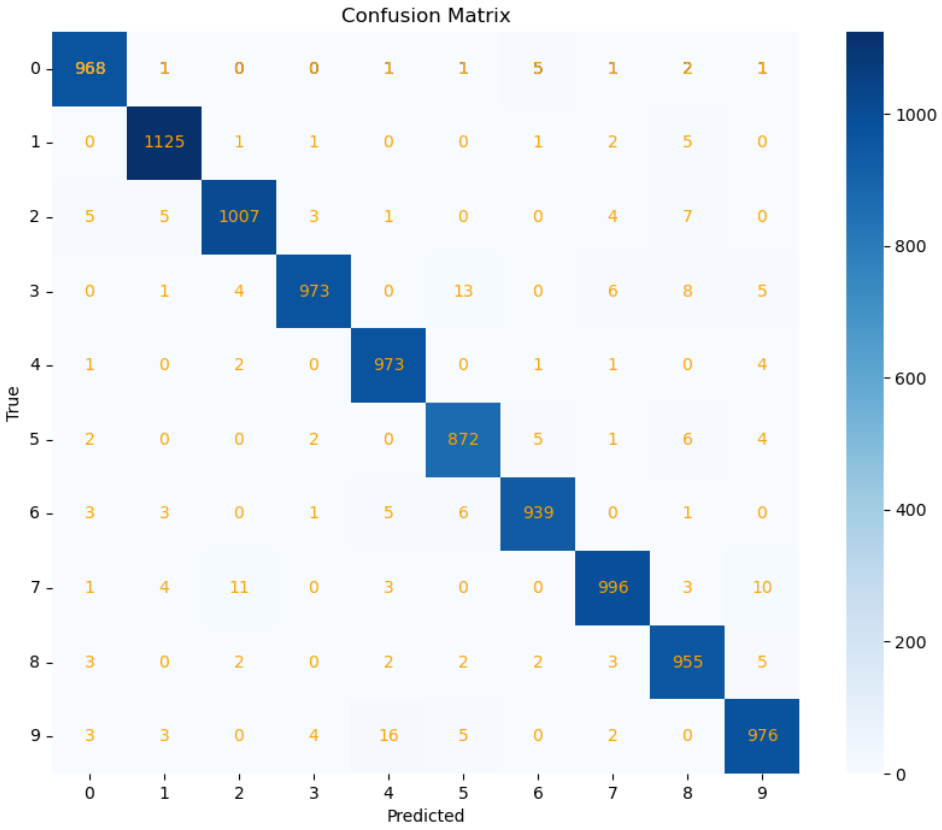

```
Training with 50.0% of training data...
24000
19200
4800
Epoch 1/10, Train Loss: 1.9540, Val Loss: 0.7062
Epoch 2/10, Train Loss: 0.4623, Val Loss: 0.3299
Epoch 3/10, Train Loss: 0.2880, Val Loss: 0.2401
Epoch 4/10, Train Loss: 0.2156, Val Loss: 0.1880
Epoch 5/10, Train Loss: 0.1746, Val Loss: 0.1632
Epoch 6/10, Train Loss: 0.1472, Val Loss: 0.1488
Epoch 7/10, Train Loss: 0.1285, Val Loss: 0.1400
Epoch 8/10, Train Loss: 0.1136, Val Loss: 0.1222
Epoch 9/10, Train Loss: 0.1022, Val Loss: 0.1136
Epoch 10/10, Train Loss: 0.0935, Val Loss: 0.1137
Test Accuracy with 50.0% of training data: 0.9694
```

**Figure 3:** χρησιμοποιώντας το 50% των 60.000 δειγμάτων εκπαίδευσης

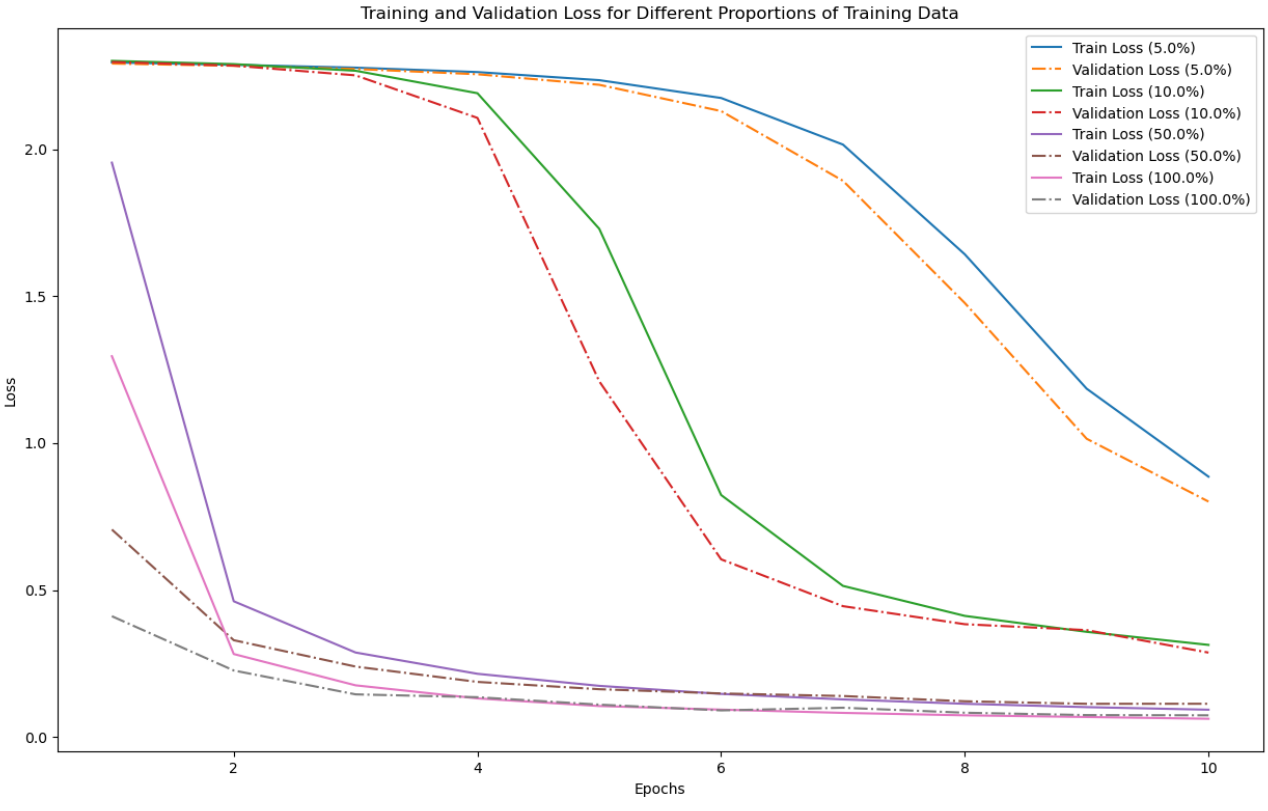


```
Training with 100.0% of training data...
48000
38400
9600
Epoch 1/10, Train Loss: 1.2957, Val Loss: 0.4114
Epoch 2/10, Train Loss: 0.2829, Val Loss: 0.2269
Epoch 3/10, Train Loss: 0.1764, Val Loss: 0.1457
Epoch 4/10, Train Loss: 0.1319, Val Loss: 0.1360
Epoch 5/10, Train Loss: 0.1060, Val Loss: 0.1107
Epoch 6/10, Train Loss: 0.0937, Val Loss: 0.0911
Epoch 7/10, Train Loss: 0.0826, Val Loss: 0.1002
Epoch 8/10, Train Loss: 0.0743, Val Loss: 0.0832
Epoch 9/10, Train Loss: 0.0686, Val Loss: 0.0752
Epoch 10/10, Train Loss: 0.0627, Val Loss: 0.0743
Test Accuracy with 100.0% of training data: 0.9784
```

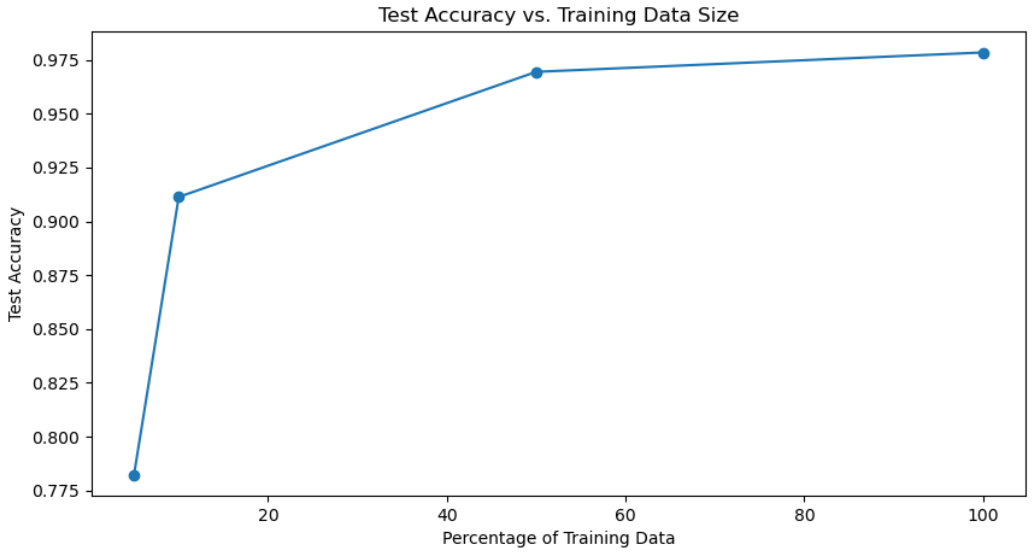
**Figure 4:** χρησιμοποιώντας το 100% των 60.000 δειγμάτων εκπαίδευσης




Αφού εκτελεστεί ο κώδικας για όλα τα proportion τότε κάνουμε plot τις απώλειες εκπαίδευσης και επικύρωσης για διαφορετικά ποσοστά των δεδομένων εκπαίδευσης.



Και το test accuracy σε συνάρτηση με το ποσοστό των δεδομένων εκπαίδευσης που χρησιμοποιήθηκαν.




Από την ανάλυση είναι προφανές ότι η αύξηση του όγκου των δεδομένων εκπαίδευσης βελτιώνει σημαντικά την απόδοση του μοντέλου. Με μικρά ποσοστά δεδομένων εκπαίδευσης (5% και 10%), το μοντέλο δυσκολεύεται τόσο με τη μάθηση όσο και με τη γενίκευση, γεγονός που αντικατοπτρίζεται από τις υψηλότερες απώλειες επικύρωσης. Καθώς το ποσοστό αυξάνεται στο 50% και στο 100%, η ικανότητα του μοντέλου να μαθαίνει και να γενικεύει βελτιώνεται αισθητά, με τις απώλειες επικύρωσης να αντιστοιχούν σε μεγάλο βαθμό στις απώλειες εκπαίδευσης. Αυτό επισημαίνει τη σημασία επαρκών δεδομένων εκπαίδευσης για την ανάπτυξη εύρωστων μοντέλων που γενικεύουν καλά σε νέα δεδομένα.

# Μέρος Β

Κατηγοριοποίηση Εικόνων μέσω εξαγωγής χαρακτηριστικών και νευρωνικού Ταξινομητή

## Ερώτημα 1

Το ανιχνευτής/περιγραφέας χαρακτηριστικών γενικού σκοπού που θα χρησιμοποιηθεί είναι ο HoG. Το Histogram of Oriented Gradients (HoG) είναι ικανοποιητικό για το σύνολο δεδομένων MNIST λόγω της απλότητας και της σαφήνειας των εικόνων των ψηφίων που περιέχει. Τα ψηφία MNIST είναι ασπρόμαυρες εικόνες μικρού μεγέθους (28x28 pixels) με καθαρά και σαφή περιγράμματα, γεγονός που επιτρέπει στο HoG να εξάγει αποτελεσματικά τα τοπικά χαρακτηριστικά και τις δομές κάθε ψηφίου. Η μέθοδος HoG είναι γρήγορη στον υπολογισμό και αποδοτική, καθώς παράγει διανύσματα χαρακτηριστικών που είναι ανθεκτικά σε μικρές μετατοπίσεις και περιστροφές, κάτι που αρκεί για τις σταθερές και κανονικοποιημένες εικόνες του MNIST. Επιπλέον, το HoG παρέχει μια καλή ισορροπία μεταξύ υπολογιστικής πολυπλοκότητας και ακρίβειας, κάνοντάς το ιδανική επιλογή για ένα dataset όπως το MNIST, όπου η ταχύτητα και η απόδοση είναι κρίσιμα ζητήματα για την επεξεργασία μεγάλου αριθμού εικόνων.

```
def extract_hog_features(images):
    hog_features = []
    for image in images:

        hog_feature = hog(image, pixels_per_cell=(2, 2), cells_per_block=(1, 1),
visualize=False)
        # plt.axis("off")
        # plt.imshow(hog_image, cmap="gray")
        # plt.show()
        hog_features.append(hog_feature)
    return np.array(hog_features)

print("Extracting Train HoG features.")
train_hog_features = extract_hog_features(train_images)
print("Extracting Train HoG features.")
test_hog_features = extract_hog_features(test_images)
```

Η εξαγωγή των χαρακτηριστικών της εικόνας γίνεται μέσω της συνάρτησης extract_hog_features η οποία εφαρμόζει την hog της skimage.feature σε κάθε εικόνα. Η συνάτηση αυτή εφαρμόζεται τόσο στις εικόνες εκπαίδευσης όσο και στις εικόνες ελέγχου.

## Ερώτημα 2

Χρησιμοποιείται το train_test_split από το sklearn.model_selection για να χωρίσει το σύνολο δεδομένων εκπαίδευσης σε υποσύνολα εκπαίδευσης και επικύρωσης με διαχωρισμό 80-20.

```
X_train, X_val, y_train, y_val = train_test_split(train_hog_features,
train_labels, test_size=0.2, random_state=42)
```

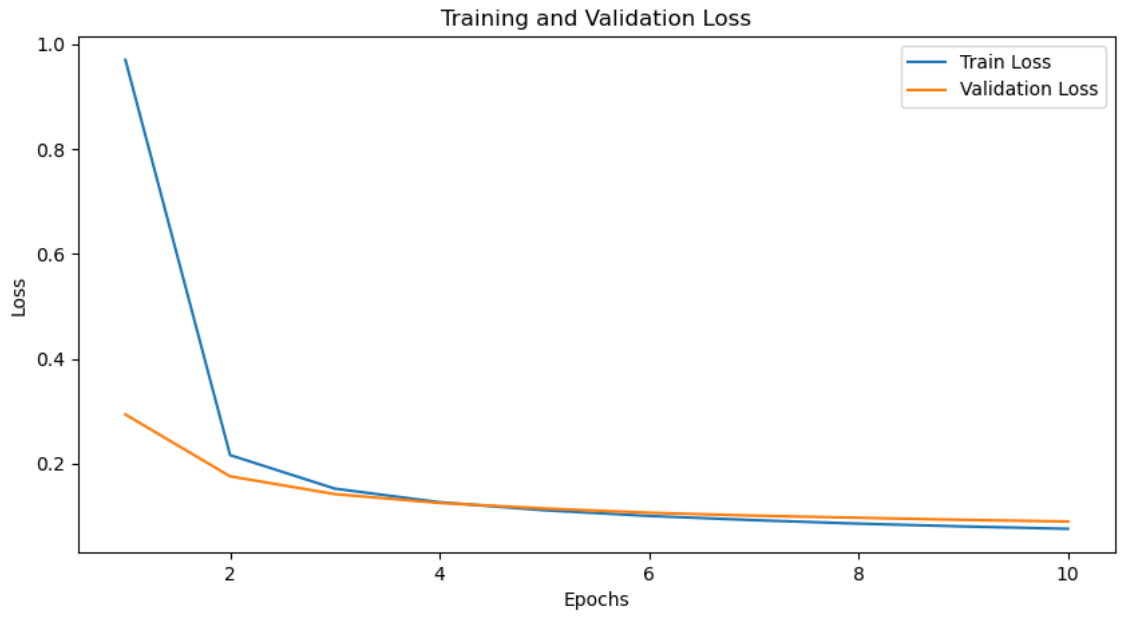
## Ερώτημα 3

**α]** Χρησιμοποιείται το DataLoader από το PyTorch για να δημιουργήσουμε τα mini-batch.

```
train_dataset = torch.utils.data.TensorDataset(X_train, y_train)
val_dataset = torch.utils.data.TensorDataset(X_val, y_val)
test_dataset = torch.utils.data.TensorDataset(X_test, y_test)

train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=32,
shuffle=True)
val_dataloader = torch.utils.data.DataLoader(val_dataset, batch_size=32,
shuffle=False)
test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=32,
shuffle=False)
```

**b/c/d]** Η συνάρτηση train_model υλοποιεί τη διαδικασία εκπαίδευσης SGD, με αντίστοιχο τρόπο με αυτό που περιγράψαμε στο μέρος Α.

Στην συνέχεια γίνονται plot οι τιμές των train_loss και val_loss που επιστρέφονται από την συνάρτηση train_model:



```
Extracting Train HoG features.
Extracting Test HoG features.
Epoch 1/10, Train Loss: 0.9549, Val Loss: 0.2917
Epoch 2/10, Train Loss: 0.2159, Val Loss: 0.1758
Epoch 3/10, Train Loss: 0.1524, Val Loss: 0.1421
Epoch 4/10, Train Loss: 0.1264, Val Loss: 0.1252
Epoch 5/10, Train Loss: 0.1109, Val Loss: 0.1143
Epoch 6/10, Train Loss: 0.1001, Val Loss: 0.1066
Epoch 7/10, Train Loss: 0.0920, Val Loss: 0.1012
Epoch 8/10, Train Loss: 0.0856, Val Loss: 0.0964
Epoch 9/10, Train Loss: 0.0802, Val Loss: 0.0926
Epoch 10/10, Train Loss: 0.0756, Val Loss: 0.0896
Test Accuracy: 97.56%
```

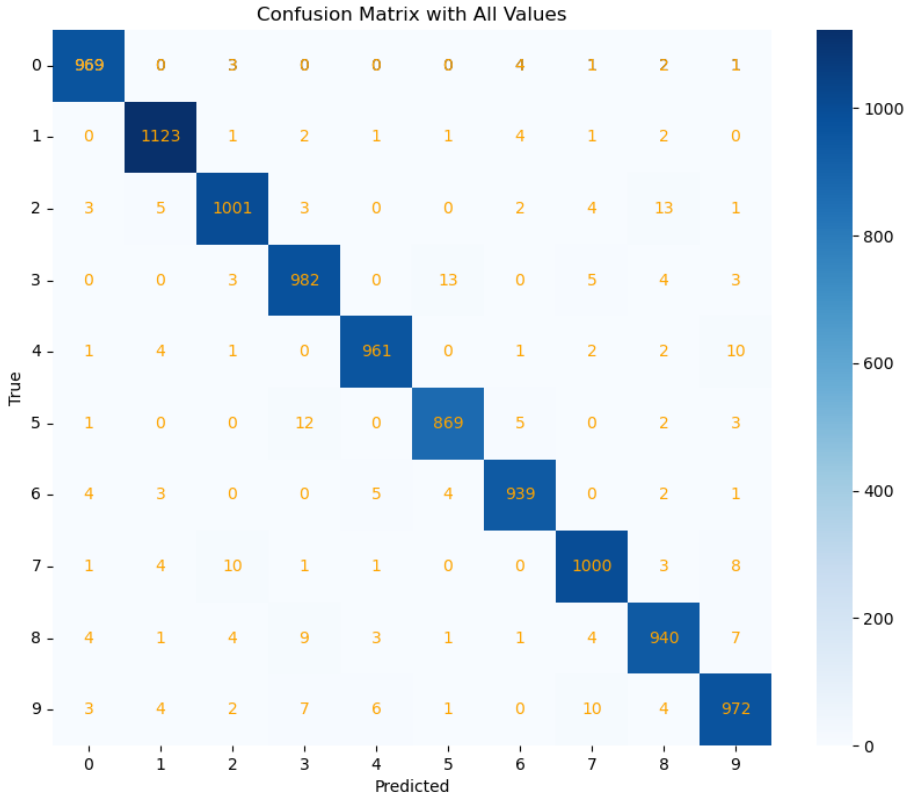
Κατά τη διάρκεια της εκπαίδευσης, μετά από κάθε epoch, εάν η απώλεια επικύρωσης είναι μικρότερη από την προηγούμενη καλύτερη απώλεια επικύρωσης, αποθηκεύουμε την κατάσταση του μοντέλου σε ένα ξεχωριστό .pth αρχείο από αυτό του μέρους Α.

```
if epoch_val_loss < best_val_loss:
            best_val_loss = epoch_val_loss
            torch.save(model.state_dict(), 'best_fcn_model.pth')
```

## Ερώτημα 4

Αρχικά φορτώνουμε τις παραμέτρους του μοντέλου με τις καλύτερες επιδόσεις (αποθηκευμένες κατά την εκπαίδευση) από ένα αρχείο με όνομα 'best_model.pth'. Για το σκοπό αυτό χρησιμοποιείται η συνάρτηση load_state_dict. Μετά τη φόρτωση του μοντέλου, η model.eval() θέτει το μοντέλο σε κατάσταση αξιολόγησης και με αντίστοιχο τρόπο με αυτόν στο μέρος Α ελέγχουμε την ακρίβεια του μοντέλου και τυπώνουμε το confusion matrix.

```
Test Accuracy: 97.56%
```



## Ερώτημα 5

Για κάθε αναλογία (proportion), υπολογίζεται το μέγεθος του υποσυνόλου (subset_size) πολλαπλασιάζοντας το proportion με το συνολικό μέγεθος του συνόλου δεδομένων εκπαίδευσης (total_train_size). Στη συνέχεια, επιλέγεται τυχαία ένα υποσύνολο των δεδομένων εκπαίδευσης χρησιμοποιώντας τη συνάρτηση np.random.choice.

```
total_train_size = len(X_train)
proportions = [0.05, 0.1, 0.5, 1.0]
results = {}
all_train_losses = []
all_val_losses = []

for prop in proportions:
    print(f'\nTraining with {prop * 100}% of training data...')
    subset_size = int(prop * total_train_size)
    # Randomly select subset from X_train
    subset_indices = np.random.choice(total_train_size, subset_size,
replace=False)
    subset_X_train = X_train[subset_indices]
    subset_y_train = y_train[subset_indices]
```

Στη συνέχεια, το υποσύνολο των δεδομένων εκπαίδευσης χωρίζεται σε σύνολα εκπαίδευσης και επικύρωσης. Συγκεκριμένα, το 80% του υποσυνόλου χρησιμοποιείται για εκπαίδευση (train_size) και το υπόλοιπο 20% για επικύρωση (val_size), όπως έγινε και προηγουμένως. Χρησιμοποιείται το DataLoader από το PyTorch για να δημιουργηθούν τα mini-batch.

```
train_size = int(0.8 * subset_size)  # 80% for training
val_size = subset_size - train_size  # 20% for validation

subset_X_train, subset_X_val, subset_y_train, subset_y_val =
train_test_split(subset_X_train, subset_y_train, test_size=0.2,
random_state=42)

subset_train_loader = torch.utils.data.DataLoader(subset_train_dataset,
batch_size=32, shuffle=True)
subset_val_loader = torch.utils.data.DataLoader(subset_val_dataset,
batch_size=32, shuffle=False)
```

Η συνάρτηση train_model καλείται για να εκπαιδεύσει το μοντέλο στο υποσύνολο των δεδομένων εκπαίδευσης για 10 epochs. Οι απώλειες εκπαίδευσης και επικύρωσης για κάθε αναλογία αποθηκεύονται στις αντίστοιχες λίστες (all_train_losses και all_val_losses). Στη συνέχεια, φορτώνονται οι καλύτερες παράμετροι του μοντέλου που αποθηκεύτηκαν κατά την εκπαίδευση και το μοντέλο τίθεται σε κατάσταση αξιολόγησης.

```
model = FCN()
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01)

    train_loss, val_loss = train_model(model, subset_train_loader,
subset_val_loader, criterion, optimizer, epochs=10)
    all_train_losses.append(train_loss)
    all_val_losses.append(val_loss)
    model.load_state_dict(torch.load('best_fcn_model.pth'))
    model.eval()
```

Έπειτα, ηαπόδοση του μοντέλου αξιολογείται στο test set και η ακρίβεια καταγράφεται για κάθε αναλογία εκπαίδευσης. Επίσης, δημιουργείται και απεικονίζεται το confusion matrix για να αναλυθεί η απόδοση του μοντέλου.

```
Training with 5.0% of training data...
Epoch 1/10, Train Loss: 2.2699, Val Loss: 2.2310
Epoch 2/10, Train Loss: 2.1846, Val Loss: 2.1344
Epoch 3/10, Train Loss: 2.0711, Val Loss: 2.0076
Epoch 4/10, Train Loss: 1.9271, Val Loss: 1.8527
Epoch 5/10, Train Loss: 1.7575, Val Loss: 1.6779
Epoch 6/10, Train Loss: 1.5726, Val Loss: 1.4972
Epoch 7/10, Train Loss: 1.3853, Val Loss: 1.3215
Epoch 8/10, Train Loss: 1.2072, Val Loss: 1.1595
Epoch 9/10, Train Loss: 1.0462, Val Loss: 1.0164
Epoch 10/10, Train Loss: 0.9068, Val Loss: 0.8937
Test Accuracy with 5.0% of training data: 0.8990
```

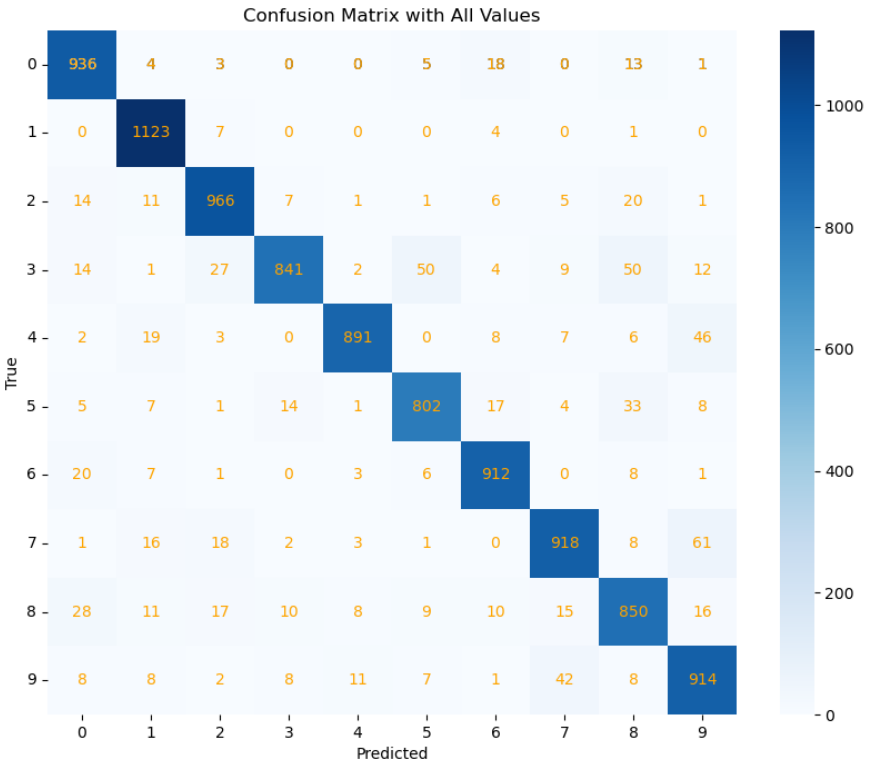
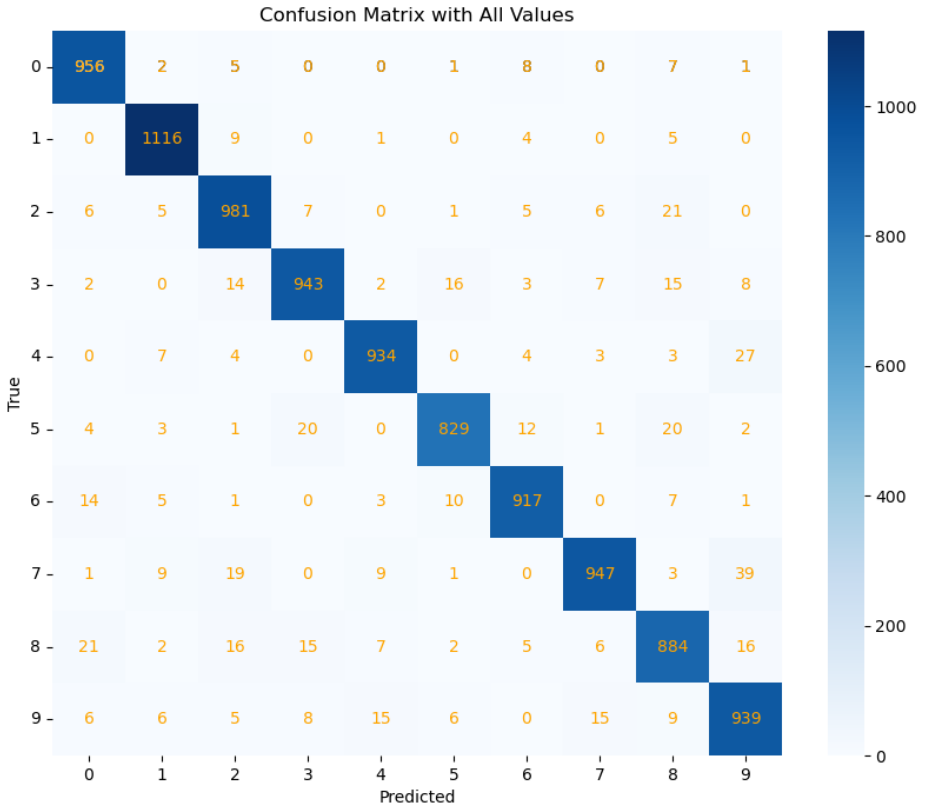



**Figure 2:** χρησιμοποιώντας το 5% των 60.000 δειγμάτων εκπαίδευσης

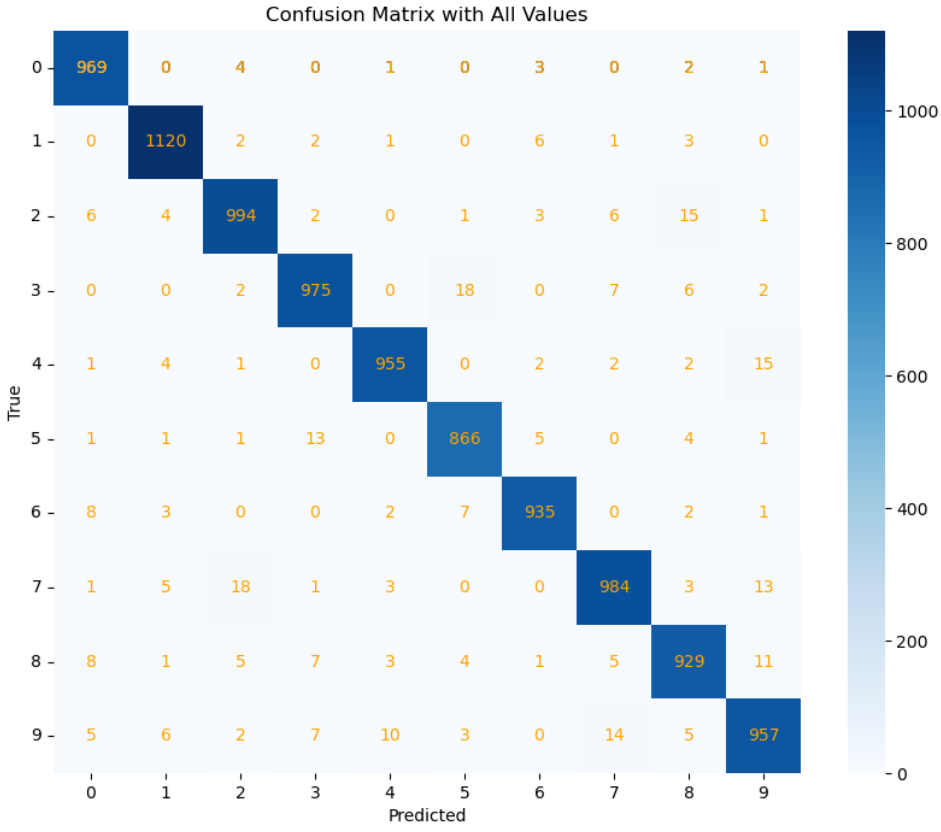
```
Training with 10.0% of training data...
Epoch 1/10, Train Loss: 2.2351, Val Loss: 2.1552
Epoch 2/10, Train Loss: 2.0276, Val Loss: 1.8933
Epoch 3/10, Train Loss: 1.7013, Val Loss: 1.5381
Epoch 4/10, Train Loss: 1.3289, Val Loss: 1.1865
Epoch 5/10, Train Loss: 1.0060, Val Loss: 0.9113
Epoch 6/10, Train Loss: 0.7671, Val Loss: 0.7185
Epoch 7/10, Train Loss: 0.6033, Val Loss: 0.5872
Epoch 8/10, Train Loss: 0.4927, Val Loss: 0.4966
Epoch 9/10, Train Loss: 0.4162, Val Loss: 0.4345
Epoch 10/10, Train Loss: 0.3611, Val Loss: 0.3883
Test Accuracy with 10.0% of training data: 0.9437
```

**Figure 2:** χρησιμοποιώντας το 10% των 60.000 δειγμάτων εκπαίδευσης



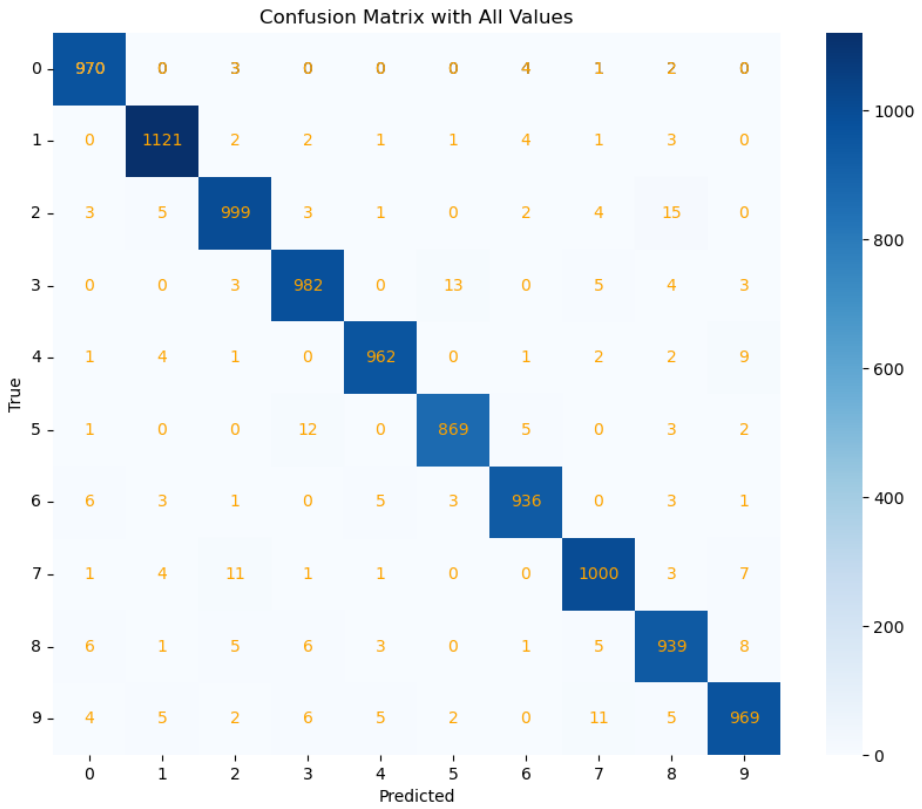

```
Training with 50.0% of training data...
Epoch 1/10, Train Loss: 1.6891, Val Loss: 0.8951
Epoch 2/10, Train Loss: 0.5458, Val Loss: 0.3504
Epoch 3/10, Train Loss: 0.2874, Val Loss: 0.2344
Epoch 4/10, Train Loss: 0.2135, Val Loss: 0.1885
Epoch 5/10, Train Loss: 0.1781, Val Loss: 0.1632
Epoch 6/10, Train Loss: 0.1560, Val Loss: 0.1459
Epoch 7/10, Train Loss: 0.1407, Val Loss: 0.1352
Epoch 8/10, Train Loss: 0.1289, Val Loss: 0.1253
Epoch 9/10, Train Loss: 0.1197, Val Loss: 0.1189
Epoch 10/10, Train Loss: 0.1118, Val Loss: 0.1132
Test Accuracy with 50.0% of training data: 0.9685
```

**Figure 3:** χρησιμοποιώντας το 50% των 60.000 δειγμάτων εκπαίδευσης

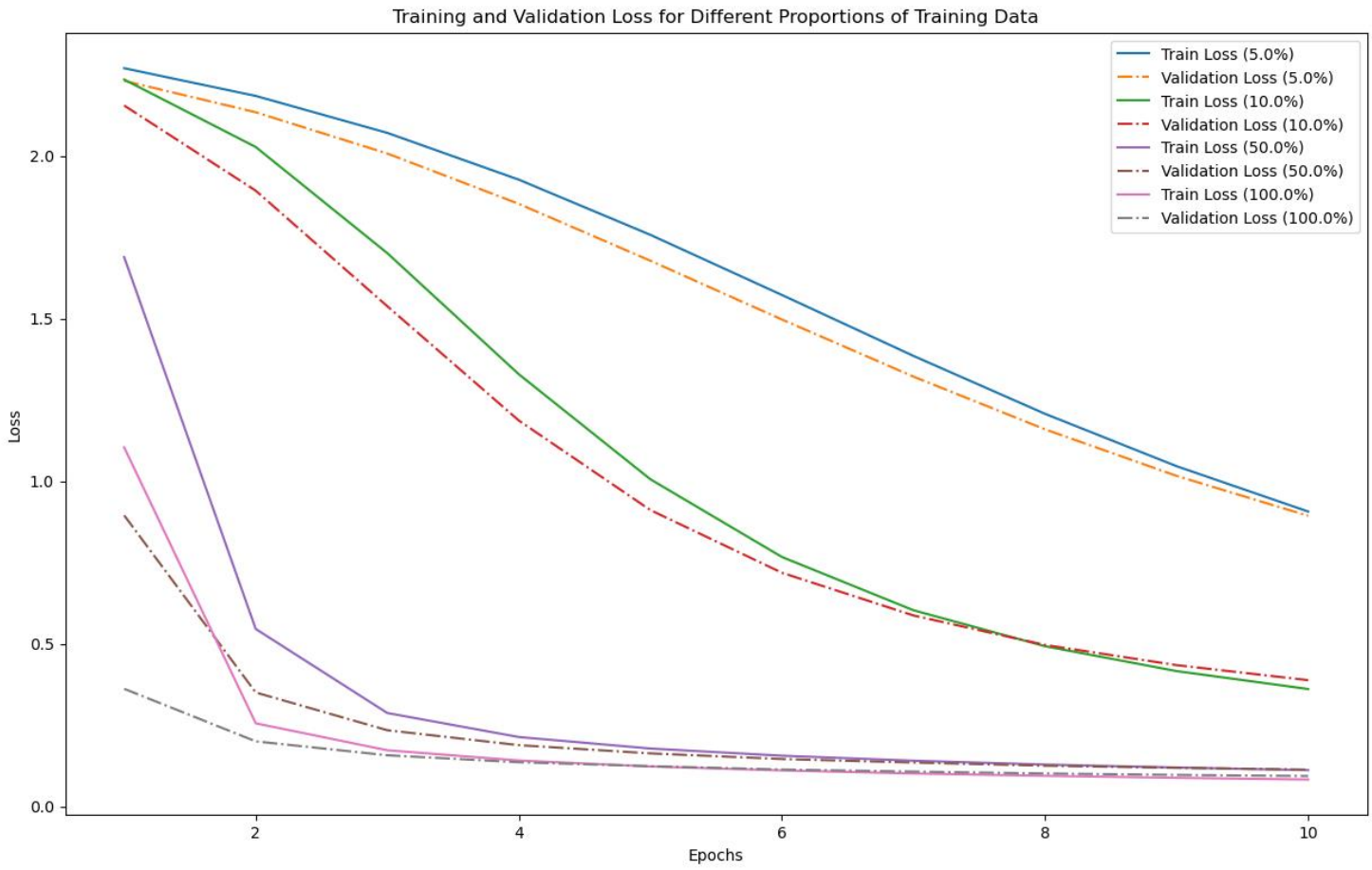


```
Training with 100.0% of training data...
Epoch 1/10, Train Loss: 1.1040, Val Loss: 0.3618
Epoch 2/10, Train Loss: 0.2557, Val Loss: 0.2003
Epoch 3/10, Train Loss: 0.1728, Val Loss: 0.1574
Epoch 4/10, Train Loss: 0.1413, Val Loss: 0.1361
Epoch 5/10, Train Loss: 0.1231, Val Loss: 0.1238
Epoch 6/10, Train Loss: 0.1108, Val Loss: 0.1136
Epoch 7/10, Train Loss: 0.1014, Val Loss: 0.1072
Epoch 8/10, Train Loss: 0.0942, Val Loss: 0.1013
Epoch 9/10, Train Loss: 0.0881, Val Loss: 0.0970
Epoch 10/10, Train Loss: 0.0829, Val Loss: 0.0936
Test Accuracy with 100.0% of training data: 0.9748
```

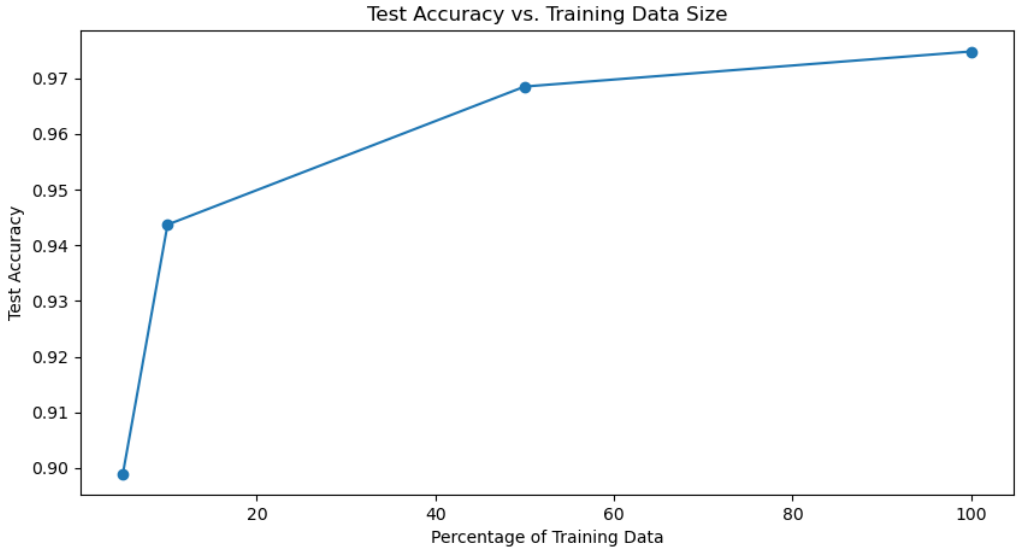
**Figure 4:** χρησιμοποιώντας το 100% των 60.000 δειγμάτων εκπαίδευσης




Αφού εκτελεστεί ο κώδικας για όλα τα proportion τότε κάνουμε plot τις απώλειες εκπαίδευσης και επικύρωσης για διαφορετικά ποσοστά των δεδομένων εκπαίδευσης.



Και το test accuracy σε συνάρτηση με το ποσοστό των δεδομένων εκπαίδευσης που χρησιμοποιήθηκαν.



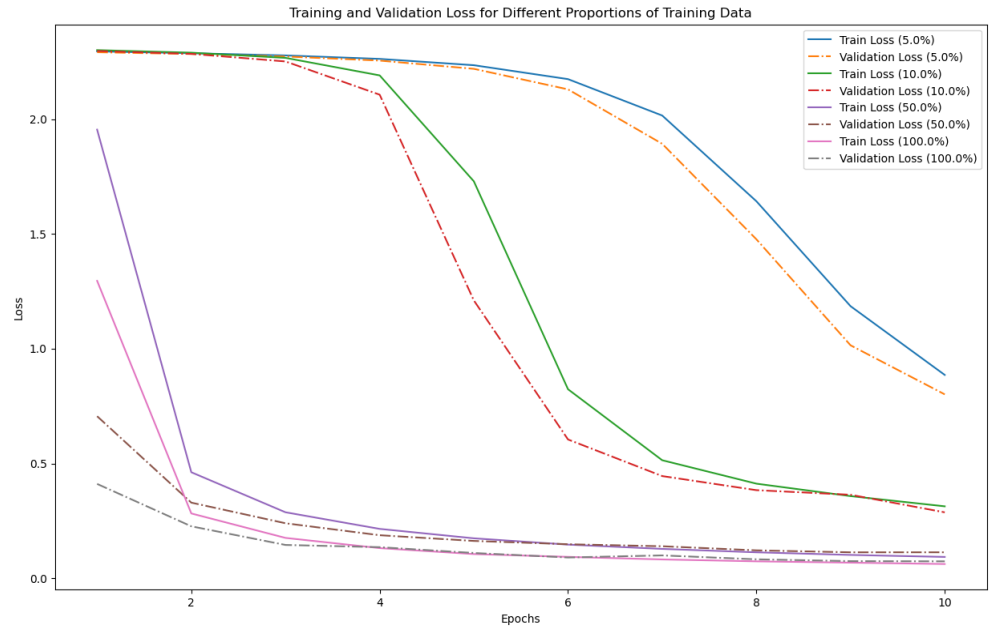

Η ανάλυση δείχνει ότι η αύξηση του μεγέθους του συνόλου εκπαίδευσης βελτιώνει σημαντικά την απόδοση του μοντέλου. Με μικρότερα ποσοστά εκπαίδευσης (5% και 10%), το μοντέλο δυσκολεύεται τόσο στην εκμάθηση όσο και στη γενίκευση, όπως φαίνεται από τις υψηλές τιμές απώλειας επικύρωσης. Με την αύξηση του ποσοστού στο 50% και 100%, η ικανότητα του μοντέλου να μαθαίνει και να γενικεύει βελτιώνεται αισθητά, με τις απώλειες επικύρωσης να είναι πολύ κοντά στις απώλειες εκπαίδευσης. Αυτό τονίζει τη σημασία της επαρκούς ποσότητας δεδομένων εκπαίδευσης για την ανάπτυξη αποτελεσματικών και γενικεύσιμων μοντέλων, υπογραμμίζοντας τον κρίσιμο ρόλο της διαθεσιμότητας δεδομένων στην απόδοση των μοντέλων μηχανικής μάθησης.

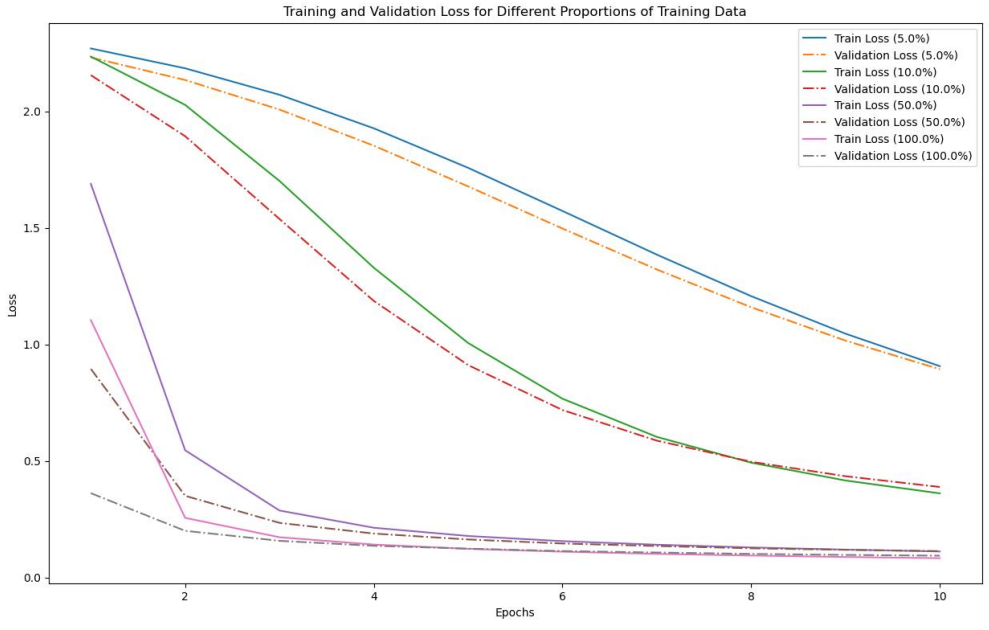
## Ερώτημα 6

Παρατηρούμε ότι και με τις δύο μεθόδους μπορούμε αποδοτικά να ταξινομήσουμε τις εικόνες. Στις εικόνες όμως που χρησιμοποιήσαμε είναι, τα ψηφία MNIST είναι ασπρόμαυρες εικόνες μικρού μεγέθους (28x28 pixels) με καθαρά και σαφή περιγράμματα, γεγονός που επιτρέπει στο HoG να εξάγει αποτελεσματικά τα τοπικά χαρακτηριστικά και τις δομές κάθε ψηφίου και έτσι εύκολα να ταξινομήσουμε στη συνέχεια τα ψηφία.

Η μέθοδος αυτή μπορεί να είναι αποτελεσματική για απλούστερες εργασίες ταξινόμησης εικόνων όπου τα χαρακτηριστικά είναι λιγότερο πολύπλοκα. Ωστόσο, η αποτελεσματικότητά της μπορεί να εξαντληθεί νωρίτερα και μπορεί να μην είναι τόσο καλή για πιο περίπλοκες εικόνες ή μεγαλύτερα σύνολα δεδομένων.

Αντίθετα, η ικανότητα του CNN να μαθαίνει χωρικές ιεραρχίες χαρακτηριστικών το καθιστά ιδιαίτερα αποτελεσματικό για εργασίες ταξινόμησης εικόνων. Τα CNN υπερέχουν σε σενάρια με πολύπλοκες εικόνες όπου η λεπτομερής εξαγωγή χαρακτηριστικών είναι ζωτικής σημασίας.




Συνοψίζοντας, ενώ τα CNN προσφέρουν ανώτερη αποτελεσματικότητα στην ταξινόμηση εικόνων λόγω της ικανότητάς τους να μαθαίνουν σύνθετα μοτίβα, απαιτούν περισσότερο χρόνο και υπολογιστικούς πόρους για την εκπαίδευσή τους. Αντίθετα, το HoG με ένα πλήρως συνδεδεμένο στρώμα είναι πιο αποδοτικό όσον αφορά την ταχύτητα εκπαίδευσης και τις υπολογιστικές απαιτήσεις, αλλά μπορεί να υπολείπεται σε αποτελεσματικότητα ταξινόμησης για πιο σύνθετα δεδομένα εικόνας.

**Figures:** Σφάλμα εκπαίδευσης και επικύρωσης (α)CNN (β) HoG + Fully Connected Layer για classification

**Επιλογή μεταξύ HoG και CNN:**

**HoG:**

- Αποδοτικότητα πόρων: HoG είναι υπολογιστικά ελαφρύτερο και πιο κατάλληλο για περιβάλλοντα με περιορισμένους πόρους.
- Δομημένα δεδομένα: Χρήση του HoG όταν έχουμε να κάνουμε με δομημένα δεδομένα και όχι με εικόνες.

**CNN:**

- Απόδοση τελευταίας τεχνολογίας: Τα CNN θέτουν τα standard για την ακρίβεια ταξινόμησης εικόνων.
- Μεγάλα σύνολα δεδομένων: Τα CNN ευδοκιμούν όταν υπάρχουν άφθονα δεδομένα με ετικέτες.

# ΠΑΡΑΡΤΗΜΑ

Κώδικας

**Κώδικας στο Github**

## Μέρος Α

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
from sklearn.decomposition import PCA
from sklearn.metrics import confusion_matrix
import seaborn as sns

# Step 1: Load MNIST dataset
transform = transforms.Compose([transforms.ToTensor()])

mnist_trainset = datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
mnist_testset = datasets.MNIST(root='./data', train=False, download=True,
transform=transform)

# Step 2: Display a sample image from each class
fig, axes = plt.subplots(1, 10, figsize=(20, 2))
```

```
for i in range(10):
    img, label = mnist_trainset.data[mnist_trainset.targets == i][0], i
    axes[i].imshow(img, cmap='gray')
    axes[i].set_title(label)
    axes[i].axis('off')
plt.show()

# Step 3: Split the dataset into training and validation sets (80%-20%)
train_size = int(0.8 * len(mnist_trainset))
val_size = len(mnist_trainset) - train_size
mnist_trainset, mnist_valset = torch.utils.data.random_split(mnist_trainset,
[train_size, val_size])

train_dataloader = torch.utils.data.DataLoader(mnist_trainset, batch_size=32,
shuffle=True)
val_dataloader = torch.utils.data.DataLoader(mnist_valset, batch_size=32,
shuffle=False)
test_dataloader = torch.utils.data.DataLoader(mnist_testset, batch_size=32,
shuffle=False)

# Step 4: Define the CNN model
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5, padding=0)
        self.conv2 = nn.Conv2d(6, 12, kernel_size=5, padding=0)  # Changed
input channels to 6
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(12 * 4 * 4, 128)  # Adjusted to match the reduced
spatial dimensions after pooling
        self.fc2 = nn.Linear(128, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.pool(x)
        x = self.relu(self.conv2(x))
        x = self.pool(x)
        x = x.view(-1, 12 * 4 * 4)  # Adjusted to match the reduced spatial
dimensions after pooling
        features = self.relu(self.fc1(x))
        logits = self.fc2(features)
        return logits, features


model = CNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

```python
# Step 5: Train the model
def train_model(model, train_loader, val_loader, criterion, optimizer,
epochs=10):
    train_loss, val_loss = [], []
    best_val_loss = float('inf')

    for epoch in range(epochs):
        model.train()
        running_train_loss = 0.0
        for images, labels in train_loader:
            optimizer.zero_grad()           # Clear previous gradients
            outputs, _ = model(images)      # Forward pass
            loss = criterion(outputs, labels)  # Compute loss
            loss.backward()                 # Backward pass to compute
gradients
            optimizer.step()                # Update parameters using
gradients
            running_train_loss += loss.item()

        model.eval()
        running_val_loss = 0.0
        with torch.no_grad():
            for images, labels in val_loader:
                outputs, _ = model(images)
                loss = criterion(outputs, labels)
                running_val_loss += loss.item()

        epoch_train_loss = running_train_loss / len(train_loader)
        epoch_val_loss = running_val_loss / len(val_loader)
        train_loss.append(epoch_train_loss)
        val_loss.append(epoch_val_loss)

        if epoch_val_loss < best_val_loss:
            best_val_loss = epoch_val_loss
            torch.save(model.state_dict(), 'best_model.pth')

        print(f'Epoch {epoch + 1}/{epochs}, Train Loss:
{epoch_train_loss:.4f}, Val Loss: {epoch_val_loss:.4f}')

    return train_loss, val_loss

train_loss, val_loss = train_model(model, train_dataloader, val_dataloader,
criterion, optimizer, epochs=10)

# Step 6: Plot training and validation loss
plt.figure(figsize=(10, 5))
plt.plot(range(1, 11), train_loss, label='Train Loss')
plt.plot(range(1, 11), val_loss, label='Validation Loss')
```

```
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.show()

# Step 7: Evaluate the model on the test set
model.load_state_dict(torch.load('best_model.pth'))
model.eval()

results = list()
total = 0

for itr, (image, label) in enumerate(test_dataloader):
    if torch.cuda.is_available():
        image = image.cuda()
        label = label.cuda()

    pred, _ = model(image)
    pred = torch.nn.functional.softmax(pred, dim=1)

    for i, p in enumerate(pred):
        if label[i] == torch.max(p.data, 0)[1]:
            total += 1
            results.append((image[i], torch.max(p.data, 0)[1]))

test_accuracy = total / len(mnist_testset)
print('Test accuracy {:.8f}'.format(test_accuracy))

# Visualize results
fig = plt.figure(figsize=(20, 10))

for i in range(1, 21):
    if i-1 < len(results):

        img = results[i-1][0].squeeze(0).detach().cpu()

        img = transforms.ToPILImage(mode='L')(img)
        fig.add_subplot(4, 5, i)
        plt.title(results[i-1][1].item())
        plt.imshow(img, cmap='gray')
        plt.axis('off')
    else:
        break

plt.show()

# Initialize lists to collect true and predicted labels
```

```
true_labels = []
pred_labels = []

# Set the model to evaluation mode
model.eval()

# Evaluate the model on the test dataset
with torch.no_grad():
    for images, labels in test_dataloader:
        if torch.cuda.is_available():
            images = images.cuda()
            labels = labels.cuda()

        # Get model predictions
        outputs, _ = model(images)
        _, preds = torch.max(outputs, 1)

        # Extend the lists with true and predicted labels
        true_labels.extend(labels.cpu().numpy())
        pred_labels.extend(preds.cpu().numpy())

# Compute the confusion matrix
conf_matrix = confusion_matrix(true_labels, pred_labels)

# Function to plot confusion matrix with all values
def plot_confusion_matrix(conf_matrix):
    plt.figure(figsize=(10, 8))
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
xticklabels=range(10), yticklabels=range(10))
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.title('Confusion Matrix')
    plt.xticks(ticks=[0.5 + i for i in range(10)], labels=[str(i) for i in
range(10)], rotation=0)
    plt.yticks(ticks=[0.5 + i for i in range(10)], labels=[str(i) for i in
range(10)], rotation=0)

    # Add all values of confusion matrix
    for i in range(conf_matrix.shape[0]):
        for j in range(conf_matrix.shape[1]):
            plt.text(j + 0.5, i + 0.5, conf_matrix[i, j], ha='center',
va='center', color='orange', fontsize=10)

    plt.show()

# Plot the confusion matrix with all values
plot_confusion_matrix(conf_matrix)
```

```
# Print the confusion matrix for debugging
print('Confusion Matrix:')
print(conf_matrix)



# Initialize lists to collect true and predicted labels
true_labels = []
pred_labels = []

# Step 8: Repeat training with different proportions of training data and
analyze performance
total_train_size = len(mnist_trainset)
proportions = [0.05, 0.1, 0.5, 1.0]
results = {}
all_train_losses = []
all_val_losses = []

for prop in proportions:
    print(f'\nTraining with {prop * 100}% of training data...')
    subset_size = int(prop * total_train_size)
    # Randomly select subset from mnist_trainset
    subset_indices = np.random.choice(total_train_size, subset_size,
replace=False)
    subset_trainset = torch.utils.data.Subset(mnist_trainset, subset_indices)
    print(len(subset_trainset))

    train_size = int(0.8 * subset_size)  # 80% for training
    val_size = subset_size - train_size  # 20% for validation

    subset_trainset, subset_valset =
torch.utils.data.random_split(subset_trainset, [train_size, val_size])
    print(len(subset_trainset))
    print(len(subset_valset))
    subset_train_loader = torch.utils.data.DataLoader(subset_trainset,
batch_size=32, shuffle=True)
    subset_val_loader = torch.utils.data.DataLoader(subset_valset,
batch_size=32, shuffle=False)

    model = CNN()
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01)

    train_loss, val_loss = train_model(model, subset_train_loader,
subset_val_loader, criterion, optimizer, epochs=10)
    all_train_losses.append(train_loss)
    all_val_losses.append(val_loss)
    model.load_state_dict(torch.load('best_model.pth'))
    model.eval()
```

```
    test_correct = 0
    with torch.no_grad():
        for images, labels in test_dataloader:
            outputs, _ = model(images)
            _, preds = torch.max(outputs, 1)
            test_correct += (preds == labels).sum().item()

            true_labels.extend(labels.cpu().numpy())
            pred_labels.extend(preds.cpu().numpy())

    test_accuracy = test_correct / len(mnist_testset)
    results[prop] = test_accuracy
    print(f'Test Accuracy with {prop * 100}% of training data:
{test_accuracy:.4f}')

    conf_matrix = confusion_matrix(true_labels, pred_labels)
    plot_confusion_matrix(conf_matrix)
    print(conf_matrix)

    true_labels.clear()
    pred_labels.clear()

# Plot training and validation losses for different proportions
plt.figure(figsize=(15, 10))
epochs = range(1, 11)
for i, prop in enumerate(proportions):
    plt.plot(epochs, all_train_losses[i], label=f'Train Loss ({prop * 100}%)',
linestyle='-')
    plt.plot(epochs, all_val_losses[i], label=f'Validation Loss ({prop *
100}%)', linestyle='-.')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss for Different Proportions of Training
Data')
plt.legend()
plt.show()


# Plot performance vs. training data size
plt.figure(figsize=(10, 5))
plt.plot([prop * 100 for prop in proportions], [results[prop] for prop in
proportions], marker='o')
plt.xlabel('Percentage of Training Data')
plt.ylabel('Test Accuracy')
plt.title('Test Accuracy vs. Training Data Size')
plt.show()
```

## Μέρος Β

```
import torch
import torch.nn as nn
import torch.optim as optim
from skimage.feature import hog
from skimage.transform import resize
from torchvision import datasets, transforms
from sklearn.metrics import confusion_matrix, accuracy_score
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.model_selection import train_test_split

# Step 1: Load MNIST dataset
transform = transforms.Compose([transforms.ToTensor()])

mnist_trainset = datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
mnist_testset = datasets.MNIST(root='./data', train=False, download=True,
transform=transform)

# Convert MNIST dataset to numpy arrays
def dataset_to_numpy(dataset):
    data = dataset.data.numpy()
    labels = dataset.targets.numpy()
    return data, labels

train_images, train_labels = dataset_to_numpy(mnist_trainset)
test_images, test_labels = dataset_to_numpy(mnist_testset)

# Step 2: Extract HoG features
def extract_hog_features(images):
    hog_features = []
    for image in images:
        hog_feature = hog(image, pixels_per_cell=(2, 2), cells_per_block=(1,
1), visualize=False)
        hog_features.append(hog_feature)
    return np.array(hog_features)

print("Extracting Train HoG features.")
train_hog_features = extract_hog_features(train_images)
print("Extracting Test HoG features.")
test_hog_features = extract_hog_features(test_images)

# Step 3: Define the FCN model (Fully Connected Network)
class FCN(nn.Module):
    def __init__(self):
```

```
        super(FCN, self).__init__()
        self.fc1 = nn.Linear(train_hog_features.shape[1], 128)
        self.fc2 = nn.Linear(128, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Convert features and labels to PyTorch tensors
X_train, X_val, y_train, y_val = train_test_split(train_hog_features,
train_labels, test_size=0.2, random_state=42)
X_train, X_val = torch.tensor(X_train, dtype=torch.float32),
torch.tensor(X_val, dtype=torch.float32)
y_train, y_val = torch.tensor(y_train, dtype=torch.long), torch.tensor(y_val,
dtype=torch.long)
X_test, y_test = torch.tensor(test_hog_features, dtype=torch.float32),
torch.tensor(test_labels, dtype=torch.long)

train_dataset = torch.utils.data.TensorDataset(X_train, y_train)
val_dataset = torch.utils.data.TensorDataset(X_val, y_val)
test_dataset = torch.utils.data.TensorDataset(X_test, y_test)

train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=32,
shuffle=True)
val_dataloader = torch.utils.data.DataLoader(val_dataset, batch_size=32,
shuffle=False)
test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=32,
shuffle=False)

model = FCN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Step 4: Train the model
def train_model(model, train_loader, val_loader, criterion, optimizer,
epochs=10):
    train_loss, val_loss = [], []
    best_val_loss = float('inf')

    for epoch in range(epochs):
        model.train()
        running_train_loss = 0.0
        for images, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
```

```
            loss.backward()
            optimizer.step()
            running_train_loss += loss.item()

        model.eval()
        running_val_loss = 0.0
        with torch.no_grad():
            for images, labels in val_loader:
                outputs = model(images)
                loss = criterion(outputs, labels)
                running_val_loss += loss.item()

        epoch_train_loss = running_train_loss / len(train_loader)
        epoch_val_loss = running_val_loss / len(val_loader)
        train_loss.append(epoch_train_loss)
        val_loss.append(epoch_val_loss)

        if epoch_val_loss < best_val_loss:
            best_val_loss = epoch_val_loss
            torch.save(model.state_dict(), 'best_fcn_model.pth')

        print(f'Epoch {epoch + 1}/{epochs}, Train Loss:
{epoch_train_loss:.4f}, Val Loss: {epoch_val_loss:.4f}')

    return train_loss, val_loss

train_loss, val_loss = train_model(model, train_dataloader, val_dataloader,
criterion, optimizer, epochs=10)

# Step 5: Plot training and validation loss
plt.figure(figsize=(10, 5))
plt.plot(range(1, 11), train_loss, label='Train Loss')
plt.plot(range(1, 11), val_loss, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.show()

# Step 6: Evaluate the model on the test set
model.load_state_dict(torch.load('best_fcn_model.pth'))
model.eval()

test_correct = 0
with torch.no_grad():
    for images, labels in test_dataloader:
        outputs = model(images)
        _, preds = torch.max(outputs, 1)
```

```
        test_correct += (preds == labels).sum().item()

test_accuracy = test_correct / len(test_dataset)
print(f'Test Accuracy: {test_accuracy * 100:.2f}%')

# Compute the confusion matrix
true_labels, pred_labels = [], []
with torch.no_grad():
    for images, labels in test_dataloader:
        outputs = model(images)
        _, preds = torch.max(outputs, 1)
        true_labels.extend(labels.cpu().numpy())
        pred_labels.extend(preds.cpu().numpy())

conf_matrix = confusion_matrix(true_labels, pred_labels)

# Plot the confusion matrix
# Function to plot confusion matrix with all values
def plot_confusion_matrix(conf_matrix):
    plt.figure(figsize=(10, 8))
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
xticklabels=range(10), yticklabels=range(10))
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.title('Confusion Matrix with All Values')
    plt.xticks(ticks=[0.5 + i for i in range(10)], labels=[str(i) for i in
range(10)], rotation=0)
    plt.yticks(ticks=[0.5 + i for i in range(10)], labels=[str(i) for i in
range(10)], rotation=0)

    # Add all values of confusion matrix
    for i in range(conf_matrix.shape[0]):
        for j in range(conf_matrix.shape[1]):
            plt.text(j + 0.5, i + 0.5, conf_matrix[i, j], ha='center',
va='center', color='orange', fontsize=10)

    plt.show()

# Plot the confusion matrix with all values
plot_confusion_matrix(conf_matrix)

true_labels.clear()
pred_labels.clear()

# Step 8: Repeat training with different proportions of training data and
analyze performance
total_train_size = len(X_train)
proportions = [0.05, 0.1, 0.5, 1.0]
```

```
results = {}
all_train_losses = []
all_val_losses = []

for prop in proportions:
    print(f'\nTraining with {prop * 100}% of training data...')
    subset_size = int(prop * total_train_size)
    # Randomly select subset from X_train
    subset_indices = np.random.choice(total_train_size, subset_size,
replace=False)
    subset_X_train = X_train[subset_indices]
    subset_y_train = y_train[subset_indices]

    train_size = int(0.8 * subset_size)  # 80% for training
    val_size = subset_size - train_size  # 20% for validation

    subset_X_train, subset_X_val, subset_y_train, subset_y_val =
train_test_split(subset_X_train, subset_y_train, test_size=0.2,
random_state=42)

    subset_train_dataset =
torch.utils.data.TensorDataset(subset_X_train.clone().detach(),
subset_y_train.clone().detach())
    subset_val_dataset =
torch.utils.data.TensorDataset(subset_X_val.clone().detach(),
subset_y_val.clone().detach())
    subset_train_loader = torch.utils.data.DataLoader(subset_train_dataset,
batch_size=32, shuffle=True)
    subset_val_loader = torch.utils.data.DataLoader(subset_val_dataset,
batch_size=32, shuffle=False)

    model = FCN()
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01)

    train_loss, val_loss = train_model(model, subset_train_loader,
subset_val_loader, criterion, optimizer, epochs=10)
    all_train_losses.append(train_loss)
    all_val_losses.append(val_loss)
    model.load_state_dict(torch.load('best_fcn_model.pth'))
    model.eval()

    test_correct = 0
    with torch.no_grad():
        for images, labels in test_dataloader:
            outputs = model(images)
            _, preds = torch.max(outputs, 1)
            test_correct += (preds == labels).sum().item()
```

```
            true_labels.extend(labels.cpu().numpy())
            pred_labels.extend(preds.cpu().numpy())

    test_accuracy = test_correct / len(test_dataset)
    results[prop] = test_accuracy
    print(f'Test Accuracy with {prop * 100}% of training data:
{test_accuracy:.4f}')

    conf_matrix = confusion_matrix(true_labels, pred_labels)
    plot_confusion_matrix(conf_matrix)
    print(conf_matrix)

    true_labels.clear()
    pred_labels.clear()

# Plot training and validation losses for different proportions
plt.figure(figsize=(15, 10))
epochs = range(1, 11)
for i, prop in enumerate(proportions):
    plt.plot(epochs, all_train_losses[i], label=f'Train Loss ({prop * 100}%)',
linestyle='-')
    plt.plot(epochs, all_val_losses[i], label=f'Validation Loss ({prop *
100}%)', linestyle='-.')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss for Different Proportions of Training
Data')
plt.legend()
plt.show()

# Plot performance vs. training data size
plt.figure(figsize=(10, 5))
plt.plot([prop * 100 for prop in proportions], [results[prop] for prop in
proportions], marker='o')
plt.xlabel('Percentage of Training Data')
plt.ylabel('Test Accuracy')
plt.title('Test Accuracy vs. Training Data Size')
plt.show()
```

# ΒΙΒΛΙΟΓΡΑΦΙΑ


- R. C. Gonzalez, “Deep Convolutional Neural Networks [Lecture Notes],” IEEE Signal Process. Mag., vol. 35, no. 6, pp. 79–87, Nov. 2018
- N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), 2005, pp. 886- 893 vol. 1, doi: 10.1109/CVPR.2005.177.
- MNIST Handwritten Digits Classification using a Convolutional Neural Network (CNN)
- Testing of Convolutional Neural Network Model
- Recent advances in dealing with data size challenges in Deep Learning
- HOG vs. CNN “Unveiling the Power of Image Classification”
- Histogram of Oriented Gradients (and car logo recognition)
- Feature Engineering for Images: A Valuable Introduction to the HOG Feature Descriptor