

**Ψηφιακή́ Επεξεργασία και Ανάλυση Εικόνας**

Ακαδημαϊκό́ Έτος 2023-2024

**Εργαστηριακές Ασκήσεις - Μέρος B**

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')

A black square with white text

Description automatically generatedplt.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]** Τοrunning\_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 τo running\_val\_loss συσσωρεύει τη συνολική απώλεια validation για για όλα τα mini-batches στο κάθε epoch.

To 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)

A graph of training and validation loss

Description automatically generatedΣτην συνέχεια γίνονται plot οι τιμές των train\_loss και val\_loss που επιστρέφονται από την συνάρτηση train\_model:

A screen shot of a computer

Description automatically generated

**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:



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

A number in squares with white text

Description automatically generated with medium confidence

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:

A screenshot of a computer screen

Description automatically generated

Για να γίνει plot o 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:

A graph of numbers and a line

Description automatically generated with medium confidence

**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()

A graph with numbers and a diagram

Description automatically generated with medium confidence

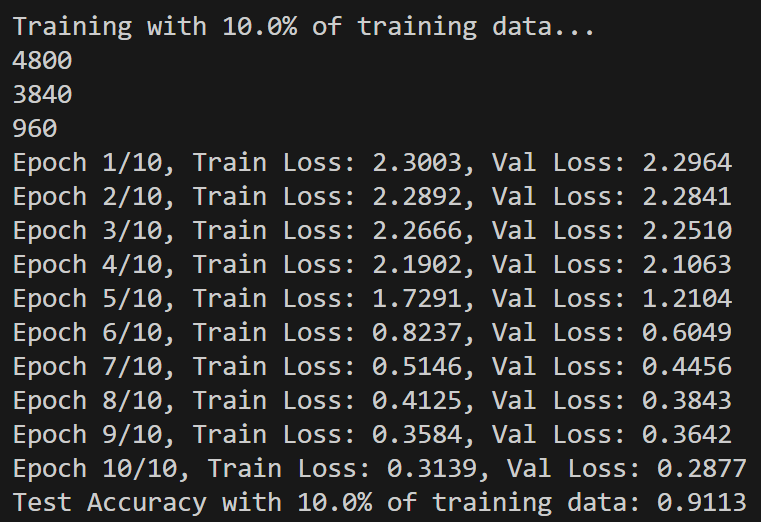
A screenshot of a computer

Description automatically generated

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

A graph with numbers and a grid

Description automatically generated with medium confidence



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

A screenshot of a computer

Description automatically generatedA graph with blue squares and numbers

Description automatically generated

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

A screenshot of a computer

Description automatically generatedA graph with numbers and squares

Description automatically generated

**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.

A graph of different colored lines

Description automatically generated

A graph with a line

Description automatically generatedΚαι 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.

A graph of training and validation loss

Description automatically generatedΣτην συνέχεια γίνονται plot οι τιμές των train\_loss και val\_loss που επιστρέφονται από την συνάρτηση train\_model:

A screen shot of a computer

Description automatically generated

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**

A graph with numbers and squares

Description automatically generatedΑρχικά 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.



**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.

A graph with numbers and a bar chart

Description automatically generated with medium confidence

A screen shot of a computer

Description automatically generated

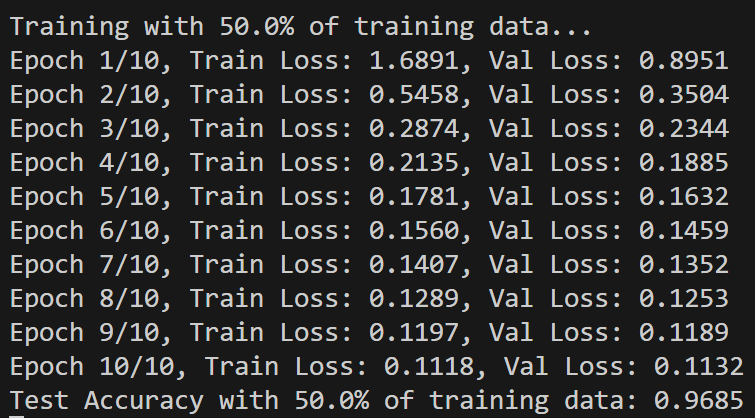
**Figure 2:** χρησιμοποιώντας το 5% των 60.000 δειγμάτων εκπαίδευσης

A screen shot of a computer

Description automatically generatedA graph with numbers and squares

Description automatically generated

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

A graph with numbers and a row of squares

Description automatically generated

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

A graph with numbers and squares

Description automatically generatedA screen shot of a computer

Description automatically generated

**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.

A graph of different colored lines

Description automatically generated

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

A graph with a line

Description automatically generated

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.

A graph of different colored lines

Description automatically generatedΑντίθετα, 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.

A graph of different colored lines

Description automatically generatedΣυνοψίζοντας, 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:**

**HοG:**

* Αποδοτικότητα πόρων: HοG είναι υπολογιστικά ελαφρύτερο και πιο κατάλληλο για περιβάλλοντα με περιορισμένους πόρους.
* Δομημένα δεδομένα: Χρήση του HοG 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](https://github.com/andreasftk/Image-Processing-Lab)**

**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)

# 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)](https://towardsdatascience.com/mnist-handwritten-digits-classification-using-a-convolutional-neural-network-cnn-af5fafbc35e9)
* [Testing of Convolutional Neural Network Model](https://www.javatpoint.com/pytorch-testing-of-convolutional-neural-network-model)
* [Recent advances in dealing with data size challenges in Deep Learning](https://towardsdatascience.com/review-of-recent-advances-in-dealing-with-data-size-challenges-in-deep-learning-ac5c1844af73)
* [HOG vs. CNN “Unveiling the Power of Image Classification”](https://medium.com/@entrepreneurbilal10/hog-vs-cnn-unveiling-the-power-of-image-classification-a3d1585d76e8)
* [Histogram of Oriented Gradients (and car logo recognition)](https://customers.pyimagesearch.com/lesson-sample-histogram-of-oriented-gradients-and-car-logo-recognition/)
* [Feature Engineering for Images: A Valuable Introduction to the HOG Feature Descriptor](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/)