Στατιστική επεξεργασία σήματος και μάθηση

Εργασία 2

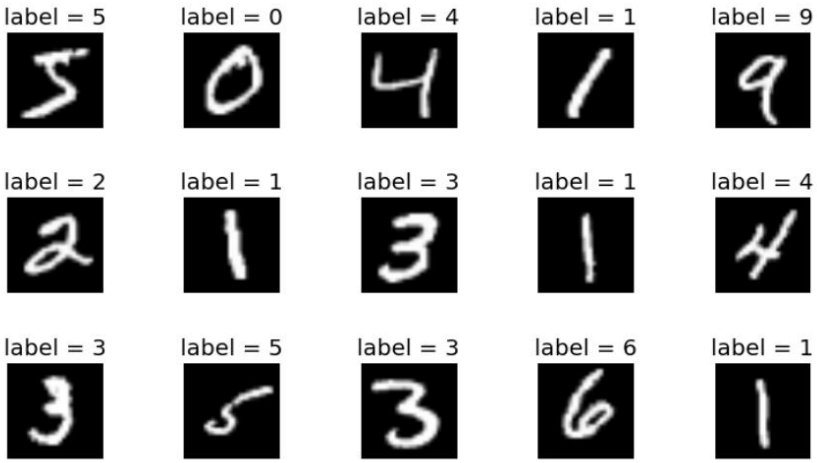
Περιεχόμενα

Ο κώδικας μπορεί να βρεθεί εδώ:

https://github.com/GrigorisTzortzakis/Statistical-signal-processing-and-machine-learning/tree/main/Exercise%202

Εισαγωγή

Αρχικά, ας δούμε το dataset που θέλουμε να χρησιμοποιήσουμε. Το σύνολο δεδομένων MNIST περιέχει εικόνες χειρόγραφων ψηφίων από 0 έως 9, οι οποίες έχουν διαστάσεις 28×28 pixels σε αποχρώσεις του γκρι. Κάθε εικόνα αντιστοιχεί σε ένα μοναδικό ψηφίο και περιλαμβάνει αντίστοιχα μια ετικέτα (label) που δηλώνει ποιο ψηφίο είναι. Έτσι, κάθε δείγμα είναι ένα ζεύγος (εικόνα, ετικέτα). Οι εικόνες είναι συνήθως αποθηκευμένες με τη μορφή πινάκων (arrays) μεγέθους 28×28, όπου κάθε στοιχείο αντιπροσωπεύει την τιμή έντασης του γκρι (από 0 ως 255 στην αρχική μορφή).



Από αυτά τα ψηφιά, έχουμε πολλαπλά δείγματα που διαφέρουν στον τρόπο γραφής, στο πάχος, στο χρώμα κλπ. Ο σκοπός μας είναι να εκπαιδεύσουμε το δίκτυο μας στο training set (60.000) και ύστερα να δούμε πόσο καλά αναγνωρίζει τις εικόνες στο test set (10.000), τις οποίες βλέπει πρώτη φορά.

Για το μοντέλο μάθησης, αξίζει να αναφέρουμε ότι χρησιμοποιούμε και τα 2 που αναφέρει η άσκηση (CNN και MLP). Ας αναλύσουμε αρχικά το MLP:

1. Μετατροπή της Εισόδου (Flatten)

Όπως είπαμε, κάθε δείγμα είναι μια grayscale εικόνα 28×28 pixel. Επειδή ένα πλήρως συνδεδεμένο στρώμα λαμβάνει συνήθως ένα μονοδιάστατο διάνυσμα ως είσοδο, η πρώτη ενέργεια που κάνουμε είναι να κανουμε flatten την εικόνα. Δηλαδή, διατάσσουμε τα 28×28 = 784 pixelσε ένα μόνο διάνυσμα 784 θέσεων. Η διάσταση αυτή καθορίζει πόσα βάρη θα έχει ο κάθε νευρώνας στο πρώτο κρυφό στρώμα. Συγκεκριμένα, ένας νευρώνας του πρώτου επιπέδου πολλαπλασιάζει καθεμία από τις 784 εισόδους με ένα ξεχωριστό βάρος, ενώ επιπλέον διαθέτει και bias.

2. Πρώτο Κρυφό Στρώμα (Fully Connected με 128 Νευρώνες)

Το πρώτο κρυφό επίπεδο περιλαμβάνει 128 νευρώνες, πλήρως συνδεδεμένους με τις 784 εισόδους. Έτσι, ο συνολικός αριθμός βαρών στο στρώμα αυτό ανέρχεται σε 784×128 (συν 128 bias, μία για κάθε νευρώνα). Ο καθένας από αυτούς τους νευρώνες υπολογίζει ένα γραμμικό συνδυασμό των εισόδων:

$$z_j = \Sigma_{i=1}^{784} w_{ij} x_i + b_j$$

όπου xi είναι το i-ο στοιχείο του διανύσματος εισόδου w_{ij} το βάρος που συνδέει την είσοδο i με τον νευρώνα j, και bj η μεροληψία. Έπειτα, εφαρμόζεται ενεργοποίηση $RELU(z_j) = \max(0, z_j)$.

3. Δεύτερο Κρυφό Στρώμα (Fully Connected με 128 Νευρώνες)

Τα αποτελέσματα (οι ενεργοποιήσεις) του πρώτου κρυφού επιπέδου αποτελούν πλέον ένα νέο διάνυσμα. Ουσιαστικά, 128 τιμές προκύπτουν από τη ReLU και προωθούνται ως είσοδος σε ένα δεύτερο κρυφό επίπεδο και πάλι με 128 νευρώνες πλήρους σύνδεσης. Οι πράξεις εδώ είναι ανάλογες, δηλαδή για κάθε νευρώνα, υπολογίζεται ένα γραμμικό άθροισμα βάσει των 128 εισόδων, συν μία μεροληψία. Εφαρμόζεται εκ νέου ReLU, για να διατηρηθεί η μη γραμμικότητα του μοντέλου.

Ο λόγος που επιλέξαμε δύο κρυφά επίπεδα των 128 νευρώνων αντί για ένα μόνο μεγαλύτερο στρώμα ή πολλά μικρότερα είναι κυρίως θέμα ισορροπίας, καθώς αρκούν για να μετασχηματίσουν επαρκώς τον χώρο εισόδων (784 διαστάσεων) σε μία παραστατική ενδιάμεση αναπαράσταση, χωρίς ταυτόχρονα να υπερφορτώνουν το μοντέλο με τεράστια πληθώρα παραμέτρων.

4. Τελικό Στρώμα (Fully Connected με 10 Νευρώνες)

Η έξοδος του δεύτερου κρυφού επιπέδου αποτελεί ένα διάνυσμα 128 τιμών ReLU. Αυτό το διάνυσμα τροφοδοτείται στο τελευταίο επίπεδο, το οποίο διαθέτει 10 νευρώνες, ένας για κάθε ψηφίο. Για κάθε νευρώνα,

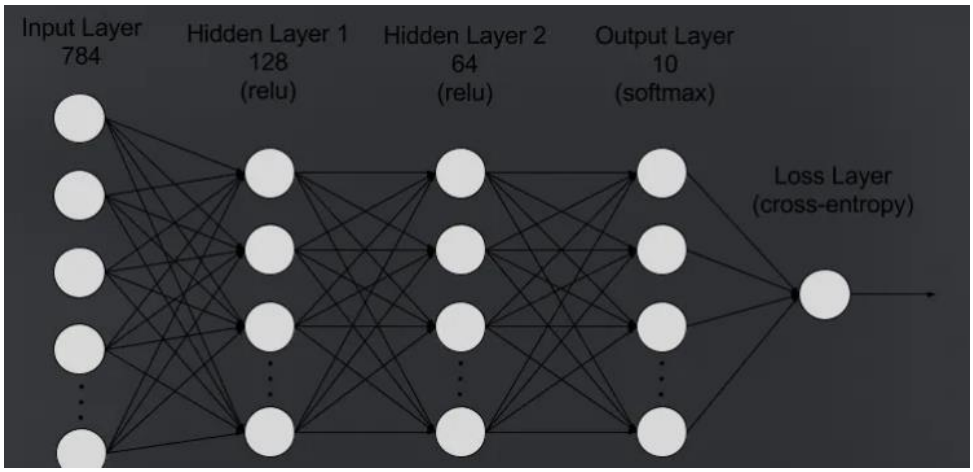
πραγματοποιείται και πάλι ένας γραμμικός υπολογισμός βάσει των 128 εισόδων, με τη μορφή:

$$o_k = \Sigma_{j=1}^{128} w_{jk}^{out} h_j + b_k^{out}$$

όπου hj είναι η j-η έξοδος του δεύτερου κρυφού επιπέδου.

Συνάρτηση Κόστους (Cross-Entropy) και Αλγόριθμος Βελτιστοποίησης (SGD)
Για να επιβλέψουμε την εκπαίδευση του δικτύου, ορίζουμε ως συνάρτηση κόστους την cross-entropy. Με αυτόν τον τρόπο, συγκρίνουμε τις προβλέψεις του MLP (έπειτα από softmax στα logits) με την πραγματική ετικέτα (label) κάθε δείγματος. Όσο πιο κοντά βρίσκεται η πιθανότητα που αποδίδει το δίκτυο στην ορθή κλάση στο 1, τόσο χαμηλότερο βγαίνει το κόστος. Αντιστρόφως, λάθος ή αβέβαιες προβλέψεις οδηγούν σε υψηλότερη τιμή κόστους.

Κατόπιν, χρησιμοποιούμε Stochastic Gradient Descent προκειμένου να ελαχιστοποιήσουμε το κόστος. Σε κάθε βήμα, το δίκτυο εκτελεί εμπρός διάδοση (forward pass), υπολογίζει την τιμή της cross-entropy, και μέσω backpropagation προκύπτουν οι κλίσεις (gradients) ως προς τα βάρη/bias. Ο αλγόριθμος SGD προχωρά ένα βήμα στη σωστή κατεύθυνση, μειώνοντας τα βάρη που ευθύνονται για ψηλές τιμές κόστους και αυξάνοντας εκείνα που βελτιώνουν την ακρίβεια. Το μέγεθος της ενημέρωσης καθορίζεται από τον ρυθμό μάθησης (learning rate).



Ας δούμε τώρα την αρχιτεκτονική του CNN μας:

1. Δύο Διαδοχικά Στρώματα Συνελίξεων (Conv2D) με Max Pooling

Η αρχιτεκτονική του μοντέλου περιλαμβάνει καταρχάς έναν μηχανισμό εξαγωγής χαρακτηριστικών (feature extractor), ο οποίος αποτελείται από δύο διαδοχικά μπλοκ συνελίξεων και υποδειγματοληψίας. Στο πρώτο μπλοκ, τα δεδομένα (εικόνες 1×28×28) διέρχονται από ένα Conv2D στρώμα, που διαθέτει

32 φίλτρα 3×3. Η συνάρτηση ενεργοποίησης ReLU ακολουθεί αμέσως μετά, εισάγοντας μη γραμμικότητα στην αναπαράσταση. Στη συνέχεια, εφαρμόζεται ένα Max Pooling 2×2, το οποίο μειώνει τις διαστάσεις της εικόνας από 28×28 σε 14×14.

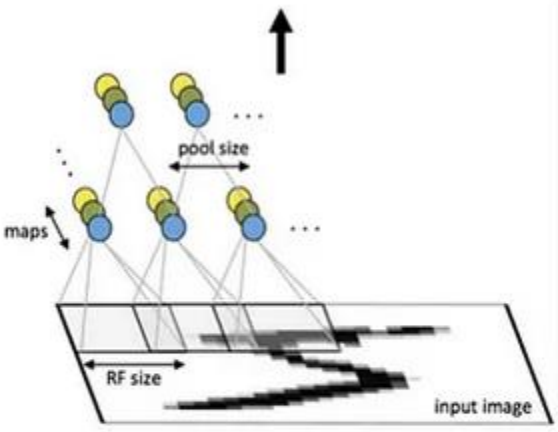
Το δεύτερο μπλοκ επαναλαμβάνει την ίδια προσέγγιση με μία ακόμα συνελίξη, αυτή τη φορά με 64 φίλτρα 3×3. Μετά τη ReLU, ακολουθεί και πάλι ένα Max Pooling 2×2, που μειώνει τις διαστάσεις από 14×14 σε 7×7.

2. Flatten και Πλήρως Συνδεδεμένα Στρώματα

Μετά το πέρας της δεύτερης υποδειγματοληψίας, οι εξαγόμενοι χάρτες χαρακτηριστικών έχουν μέγεθος 64×7×7 (δηλαδή 3136 τιμές ανά δείγμα). Για να προετοιμάσουμε αυτά τα δεδομένα για ταξινόμηση, τα κάνουμε flatten σε μονοδιάστατο διάνυσμα 3136 στοιχείων. Οι πληροφορίες που πήρε το CNN από τα στάδια συνελίξεων και pooling συνοψίζονται πλέον σε ένα εύχρηστο διάνυσμα, το οποίο περνάει σε πλήρως συνδεδεμένα στρώματα.

Επίσης, χρησιμοποιούμε ένα Dense στρώμα με 128 νευρώνες, ακολουθούμενο από συνάρτηση ενεργοποίησης ReLU για να διατηρήσουμε μη γραμμικότητα. Τέλος, προσθέτουμε ένα δεύτερο πλήρως συνδεδεμένο στρώμα 10 νευρώνων, ένα για καθεμία από τις δέκα κλάσεις. Η έξοδός του συνιστά ένα διάνυσμα, επί των οποίων εφαρμόζεται (σε επίπεδο κόστους) συνάρτηση softmax, ώστε να προκύψουν οι πιθανότητες ταξινόμησης.

Για την συνάρτηση κόστους και αλγόριθμο βελτιστοποίησης είναι ακριβώς τα ίδια με του MLP.



Bottom up view
Image of a 5

Ερώτημα 1

MLP

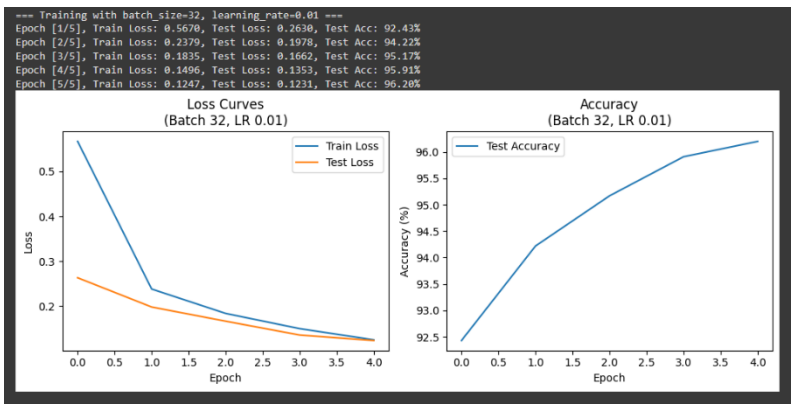
Με σκοπό να μελετήσουμε πώς διαφορετικές ρυθμίσεις επηρεάζουν τη διαδικασία εκπαίδευσης, ορίσαμε δύο σετ παραμέτρων που συνδυάζουν διαφορετικές τιμές στο batch size (32 και 128) και στο learning rate (0.01 και 0.001). Το batch size είναι πόσα δείγματα θα χρησιμοποιηθούν προτού γίνει μία ενημέρωση των βαρών, δηλαδή, πόσες εικόνες περνάνε στο δίκτυο προτού υπολογιστούν τα gradients και ενημερωθούν τα βάρη. Ένα μικρό batch size τείνει να παράγει πιο θορυβώδεις κλίσεις, ωστόσο επιτρέπει συχνότερες ενημερώσεις κατά τη διάρκεια μιας εποχής. Από την άλλη, ένα μεγάλο batch size συνεισφέρει σε πιο σταθερές κλίσεις, αλλά θα γίνονται λιγότερες ενημερώσεις ανά εποχή.

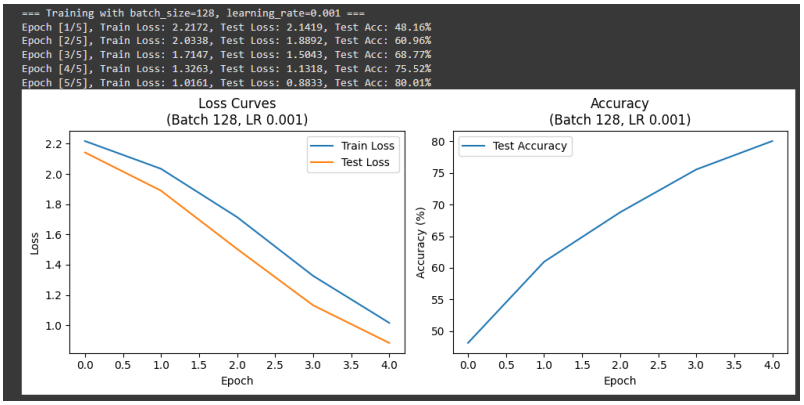
Επιπροσθέτως, ο ρυθμός μάθησης ελέγχει πόσο μεγάλο βήμα κάνει το μοντέλο στην κατεύθυνση της κλίσης, κάθε φορά που υπολογίζει το σφάλμα και θέλει να ενημερώσει τα βάρη του δικτύου. Όταν ο ρυθμός μάθησης είναι υψηλός (π.χ. 0.01), το μοντέλο προβαίνει σε μεγαλύτερες αλλαγές στα βάρη σε κάθε επανάληψη. Αυτό μπορεί να επιταχύνει αισθητά την αρχική μείωση του σφάλματος, καθώς τα βάρη μετακινούνται γρήγορα προς τη σωστή κατεύθυνση. Ωστόσο, αν ο ρυθμός μάθησης είναι υπερβολικά μεγάλος, ενδέχεται οι ενημερώσεις να γίνονται πολύ επιθετικές και τελικά να μην επιτυγχάνουμε σύγκλιση.

Αντιθέτως, ένας μικρότερος ρυθμός μάθησης (π.χ. 0.001) επιτρέπει πιο ήπια και σταδιακά βήματα βελτιστοποίησης. Το μειονέκτημα είναι ότι ενδέχεται να χρειάζονται περισσότερες εποχές ή iterations, ώστε το μοντέλο να προσεγγίσει το ίδιο επίπεδο απόδοσης που θα επιτύγχανε με έναν υψηλότερο ρυθμό μάθησης.

Κατά την εκπαίδευση, διαιρούμε τα δεδομένα σε batches βάσει του εκάστοτε batch size, και για πέντε εποχές επαναλαμβάνουμε τη διαδικασία εμπρός και πίσω διάδοσης (forward/backward pass).

Τέλος, σε κάθε εποχή, υπολογίζουμε μια μέση τιμή της συνάρτησης κόστους στα δείγματα εκπαίδευσης. Αυτό δείχνει το λάθος πάνω στα γνωστά δεδομένα. Το ίδιο ακριβώς κάνουμε και για το test loss, με την διαφορά ότι κοιτάμε το σφάλμα των αγνώστων δεδομένων. Το accuracy μας δείχνει κατά πόσο είναι σωστές οι προβλέψεις που κάνει το μοντέλο μας (η ταξινόμηση των ετικετών των εικόνων).





Από τα αποτελέσματα, επιβεβαιώνουμε την θεωρητική μας μελέτη. Το μικρό batch size επιφέρει και περισσότερες ενημερώσεις ανά εποχή (συγκεκριμένα έχουμε 60.000/32=1875), άρα ξεκινάμε με μεγάλη ακρίβεια και επιπλέον το μεγάλο learning rate μας επιτρέπει να αλλάζουμε τα βάρη περισσότερο. Επίσης, βλέπουμε πως το σφάλμα ελαχιστοποιείτε γρήγορα και πρακτικά έχει συγκλείσει μετρά από την 3 εποχή. Αντιθέτως, το μεγάλο batch size και μικρό learning rate επιφέρει χαμηλότερη ακρίβεια, αφού έχουμε λιγότερες ενημερώσεις (468, δηλαδή το ¼ από size 32) με μικρότερες αλλαγές κάθε φορά. Χρειάζεται περισσότερο χρόνο για την σύγκλιση, αλλά πράγματι οι καμπύλες είναι αρκετά πιο ομαλές και πιθανόν με αρκετό χρόνο να πέτυχουμε μεγαλύτερη ακρίβεια.

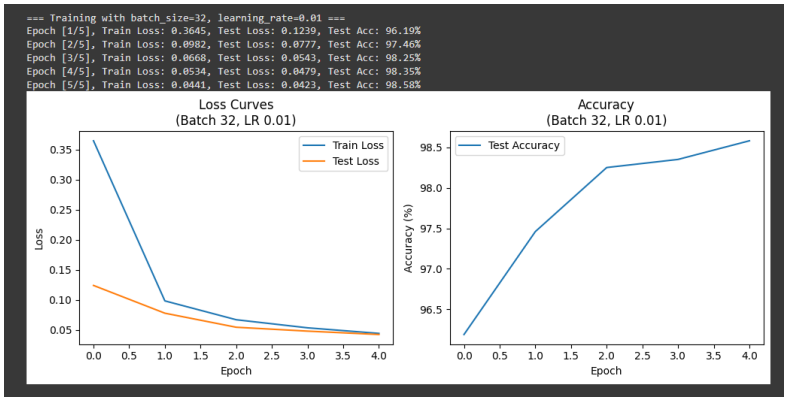
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

%matplotlib inline

# batch_size, learning_rate
PARAM_SETS = [
    {"batch_size": 32, "learning_rate": 0.01},
    {"batch_size": 128, "learning_rate": 0.001}
]

# Number of total epochs
EPOCHS = 5

# MLP model
class MLP(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=128, num_classes=10):
        super(MLP, self).__init__()
```

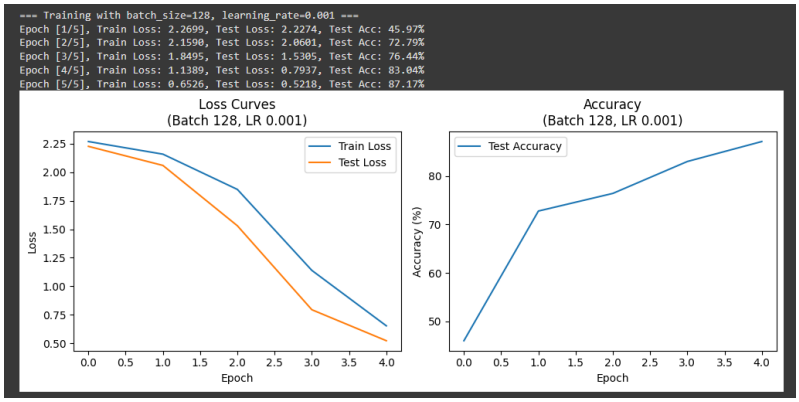
```
        self.net = nn.Sequential(
            nn.Flatten(),
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, num_classes)
        )
    def forward(self, x):
        return self.net(x)

def train_one_epoch(model, dataloader, criterion, optimizer, device):
    model.train()
    running_loss = 0.0
    for images, labels in dataloader:
        images, labels = images.to(device), labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * images.size(0)

    epoch_loss = running_loss / len(dataloader.dataset)
    return epoch_loss

def evaluate(model, dataloader, criterion, device):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)

            running_loss += loss.item() * images.size(0)

            # Calculate accuracy
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
```

```
            total += labels.size(0)

    epoch_loss = running_loss / len(dataloader.dataset)
    accuracy = 100.0 * correct / total
    return epoch_loss, accuracy

# Load mnist
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

train_dataset = torchvision.datasets.MNIST(
    root="./data", train=True, download=True, transform=transform
)
test_dataset = torchvision.datasets.MNIST(
    root="./data", train=False, download=True, transform=transform
)

# Gpu
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

for i, params in enumerate(PARAM_SETS):
    print(f"\n=== Training with batch_size={params['batch_size']}, "
          f"learning_rate={params['learning_rate']} ===")

    # Create data loaders
    train_loader = torch.utils.data.DataLoader(
        train_dataset, batch_size=params["batch_size"], shuffle=True
    )
    test_loader = torch.utils.data.DataLoader(
        test_dataset, batch_size=1000, shuffle=False
    )

    # Initialize model, criterion, optimizer
    model = MLP().to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(),
lr=params["learning_rate"])

    train_losses = []
    test_losses = []
    test_accuracies = []

    # Training loop
    for epoch in range(EPOCHS):
        train_loss = train_one_epoch(model, train_loader, criterion,
```

```
optimizer, device)
        test_loss, test_acc = evaluate(model, test_loader, criterion,
device)

        train_losses.append(train_loss)
        test_losses.append(test_loss)
        test_accuracies.append(test_acc)

        print(f"Epoch [{epoch+1}/{EPOCHS}], "
              f"Train Loss: {train_loss:.4f}, "
              f"Test Loss: {test_loss:.4f}, "
              f"Test Acc: {test_acc:.2f}%")

    # Plot

    plt.figure(figsize=(10, 4))

    #  training & test loss
    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label='Train Loss')
    plt.plot(test_losses, label='Test Loss')
    plt.title(f"Loss Curves\n(Batch {params['batch_size']}, LR
{params['learning_rate']})")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()

    # test accuracy
    plt.subplot(1, 2, 2)
    plt.plot(test_accuracies, label='Test Accuracy')
    plt.title(f"Accuracy\n(Batch {params['batch_size']}, LR
{params['learning_rate']})")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy (%)")
    plt.legend()

    plt.tight_layout()
    plt.show()
```

CNN





Έχουμε αποτελέσματα ίδιας λογικής, με την διαφορά ότι σε σχέση με το mlp στην πρώτη περίπτωση πετυχαίνουμε από την αρχή καλυτέρα αποτελέσματα (μικρότερο σφάλμα και καλύτερη ακρίβεια) ενώ στην δεύτερη αρχικά (μέχρι την εποχή 4) έχουμε χειροτέρα αποτελέσματα και μετα βελτιώνονται.

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

%matplotlib inline

# batch_size, learning_rate
PARAM_SETS = [
    {"batch_size": 32, "learning_rate": 0.01},
    {"batch_size": 128, "learning_rate": 0.001}
]

# Number of total epochs
EPOCHS = 5

# CNN model
class CNN(nn.Module):
    """
    A small CNN for MNIST:
      1) Conv2D(1->32) + ReLU + MaxPool(2)
      2) Conv2D(32->64) + ReLU + MaxPool(2)
      3) Flatten
      4) Linear(64*7*7 -> 128) + ReLU
      5) Linear(128 -> 10)
```

```
    """
    def __init__(self, num_classes=10):
        super(CNN, self).__init__()

        # Feature extractor
        self.features = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3,
padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),

            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3,
padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2)
        )

        # Classifier
        self.classifier = nn.Sequential(
            nn.Linear(64 * 7 * 7, 128),
            nn.ReLU(),
            nn.Linear(128, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x

def train_one_epoch(model, dataloader, criterion, optimizer, device):
    model.train()
    running_loss = 0.0
    for images, labels in dataloader:
        images, labels = images.to(device), labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * images.size(0)

    epoch_loss = running_loss / len(dataloader.dataset)
```

```
    return epoch_loss

def evaluate(model, dataloader, criterion, device):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)

            running_loss += loss.item() * images.size(0)

            # Calculate accuracy
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

    epoch_loss = running_loss / len(dataloader.dataset)
    accuracy = 100.0 * correct / total
    return epoch_loss, accuracy

# Load mnist
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

train_dataset = torchvision.datasets.MNIST(
    root="./data", train=True, download=True, transform=transform
)
test_dataset = torchvision.datasets.MNIST(
    root="./data", train=False, download=True, transform=transform
)

# Gpu
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

for i, params in enumerate(PARAM_SETS):
    print(f"\n=== Training with batch_size={params['batch_size']}, "
          f"learning_rate={params['learning_rate']} ===")

    # Create data loaders
    train_loader = torch.utils.data.DataLoader(
```

```
        train_dataset, batch_size=params["batch_size"], shuffle=True
    )
    test_loader = torch.utils.data.DataLoader(
        test_dataset, batch_size=1000, shuffle=False
    )

    # Initialize model, criterion, optimizer
    model = CNN().to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(),
lr=params["learning_rate"])

    train_losses = []
    test_losses = []
    test_accuracies = []

    # Training loop
    for epoch in range(EPOCHS):
        train_loss = train_one_epoch(model, train_loader, criterion,
optimizer, device)
        test_loss, test_acc = evaluate(model, test_loader, criterion,
device)

        train_losses.append(train_loss)
        test_losses.append(test_loss)
        test_accuracies.append(test_acc)

        print(f"Epoch [{epoch+1}/{EPOCHS}], "
              f"Train Loss: {train_loss:.4f}, "
              f"Test Loss: {test_loss:.4f}, "
              f"Test Acc: {test_acc:.2f}%")

      # Plot
    plt.figure(figsize=(10, 4))

    #  training & test loss
    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label='Train Loss')
    plt.plot(test_losses, label='Test Loss')
    plt.title(f"Loss Curves\n(Batch {params['batch_size']}, LR
{params['learning_rate']})")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()

  # test accuracy
    plt.subplot(1, 2, 2)
    plt.plot(test_accuracies, label='Test Accuracy')
```

```
    plt.title(f"Accuracy\n(Batch {params['batch_size']}, LR
{params['learning_rate']})")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy (%)")
    plt.legend()

    plt.tight_layout()
    plt.show()
```

Ερώτημα 2

MLP

Με federated learning αντί να μεταφορτώνουμε τα δεδομένα από κάθε χρήστη σε έναν κεντρικό server, ο καθένας διατηρεί τα δικά του παραδείγματα τοπικά και εκπαιδεύει σε αυτά έναν τοπικό αντιπρόσωπο του μοντέλου. Στη συνέχεια, οι ενημερωμένες παράμετροι από κάθε χρήστη συγχωνεύονται σε έναν κεντρικό server χρησιμοποιώντας έναν αλγόριθμο όπως το Federated Averaging (FedAvg). Με αυτόν τον τρόπο, το global μοντέλο που διατηρεί ο server ενσωματώνει τη γνώση από όλους τους χρήστες, χωρίς να απαιτείται ανταλλαγή των ίδιων των δεδομένων.

Στο πλαίσιο της υλοποίησής μας, ο server οργανώνει τη μάθηση ως εξής:

1. Broadcast του Global Μοντέλου:
Ο κεντρικός server στέλνει σε κάθε χρήστη το τρέχον global model.

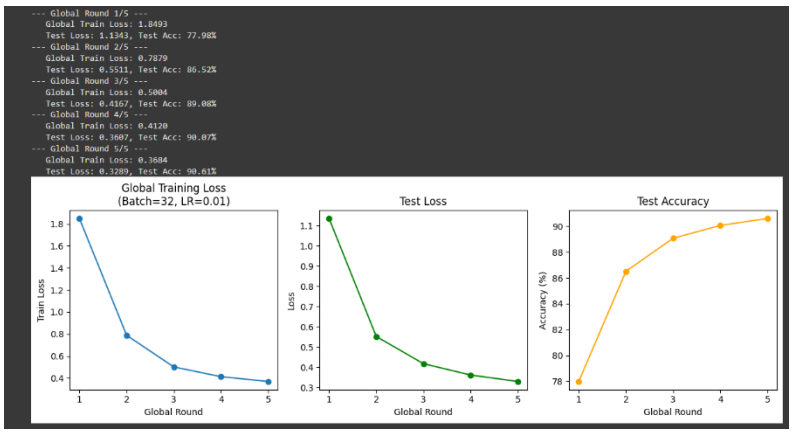
2. Τοπική Εκπαίδευση (Local Training):
Κάθε χρήστης λαμβάνει το αντίγραφο του global μοντέλου και το εκπαιδεύει στα δικά του δεδομένα, που προέρχονται από τοπικό υποσύνολο (IID) του MNIST. Χρησιμοποιεί για αυτό gradient descent, για ορισμένες τοπικές εποχές (local_epochs). Έτσι προκύπτει ένα ενημερωμένο μοντέλο για τον καθένα.

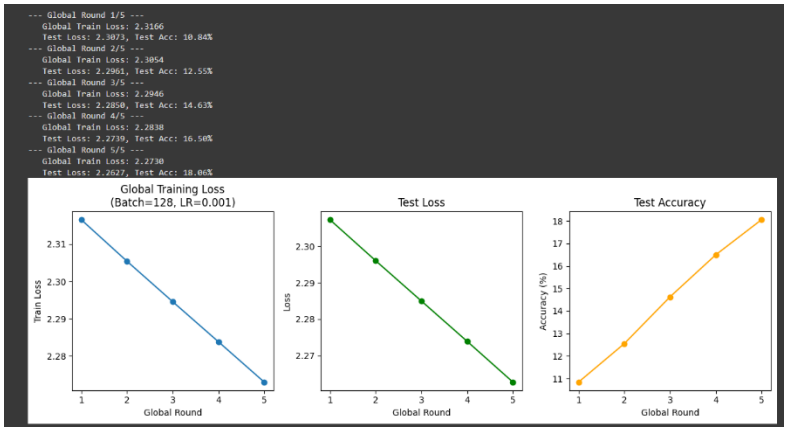
3. Συγχώνευση (Aggregation) στον Server:
Μετά την τοπική εκπαίδευση, κάθε χρήστης στέλνει μόνο τις ενημερωμένες παραμέτρους πίσω στον server. Ο server υπολογίζει έναν μέσο όρο αυτών των παραμέτρων και έτσι παράγει ένα νέο global model. Αυτή η διεργασία είναι ο FedAvg.

4. Επαναλήψεις (Global Rounds):
Η διαδικασία (broadcast → local training → aggregation) επαναλαμβάνεται για έναν αριθμό γύρων (global rounds). Σε κάθε γύρο, το global model που

προκύπτει αποκτά καλύτερη γνώση από τα τοπικά δεδομένα, ενοποιώντας τις βελτιωμένες παραμέτρους όλων των χρηστών.

Όσον αφορά τον IID τρόπο, κάθε χρήστης έχει ένα υποσύνολο του MNIST που περιλαμβάνει όλα τα ψηφία (0–9) στην ίδια αναλογία. Αυτό σημαίνει πως το δείγμα που διαθέτει κάθε χρήστης είναι αντιπροσωπευτικό του συνολικού dataset. Έχουμε 5 συνολικούς γύρους και 1 τοπική εποχή για τον κάθε χρήστη.





Από τα αποτελέσματα βλέπουμε παρόμοιο πόρισμα με αυτό του προηγουμένου ερωτήματος. Και για τις 2 περιπτώσεις, το αρχικό σφάλμα προφανώς είναι μεγάλο, αφού ακόμα δεν έχουμε ενημερώσει το μοντέλο, μόλις το στείλαμε στους χρήστες για εκπαίδευση. Στην πρώτη περίπτωση, παρατηρούμε τεράστια μείωση στο σφάλμα στον 2ο γύρο καθώς και μεγάλη άνοδο στην ακρίβεια. Αυτό συμβαίνει καθώς ο κάθε χρήστης έτρεξε το μοντέλο με τα δεδομένα του και ενημέρωσε τον server, άρα έλαβε στην συνέχεια το νέο ανανεωμένο μοντέλο. Όπως και στην κεντρική μάθηση, ουσιαστικά συγκλίνει μετά από τον 3ο γύρο. Για την 2η περίπτωση, βλέπουμε ότι το μεγάλο batch size με μικρό learning rate έχει σταθερή βελτίωση ανάμεσα στους γύρους. Η διαφορά είναι ότι τα αποτελέσματα είναι χειροτέρα κατά πολύ, ειδικά η ακρίβεια. Το μικρό local dataset σε συνδυασμό με λίγες και ήπιες ενημερώσεις επιφέρει κακά αποτελέσματα με λίγους γύρους, χρειάζεται αρκετά παραπάνω ώστε να έρθει σε ικανοποιητικά επίπεδα. Αυτό βέβαια σε μη κεντρική μάθηση είναι απαγορευτικό, χαλάμε χρόνο για την επιπλέον εκπαίδευση, bandwidth για την αποστολή δεδομένων καθώς και ενέργεια των συσκευών. Τέλος, και στις 2 περιπτώσεις τα αποτελέσματα είναι χειροτέρα από αυτά της κεντρικής μάθησης, αφού ο κάθε χρήστης δεν έχει ολόκληρο το dataset και πρέπει να συνδυάζουμε αποτελέσματα από όλους.

```
import torch
import torch.nn as nn
import torch.optim as optim
```

```
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

import copy
import math

%matplotlib inline

# batch_size, learning_rate
PARAM_SETS = [
    {"batch_size": 32,  "learning_rate": 0.01},
    {"batch_size": 128, "learning_rate": 0.001}
]

NUM_USERS = 10          # 10 federated clients
LOCAL_EPOCHS = 1        # local epochs per global round
GLOBAL_ROUNDS = 5
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", DEVICE)

# MLP model
class MLP(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=128, num_classes=10):
        super(MLP, self).__init__()
        self.net = nn.Sequential(
            nn.Flatten(),
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, num_classes)
        )
    def forward(self, x):
        return self.net(x)

# Load mnist
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

train_dataset = torchvision.datasets.MNIST(
    root="./data", train=True, download=True, transform=transform
)
test_dataset = torchvision.datasets.MNIST(
    root="./data", train=False, download=True, transform=transform
)
```

```
#  Partition Training Data (IID) for 10 Users

def iid_partition(dataset, num_users=10):
    """
    Return a dict: user_id -> list of image indices (IID partition).
    Each user gets ~1/num_users of the dataset, randomly chosen.
    """
    num_items = len(dataset) // num_users
    all_indices = torch.randperm(len(dataset))

    dict_users = {}
    start = 0
    for user_id in range(num_users):
        end = start + num_items
        dict_users[user_id] = all_indices[start:end]
        start = end
    return dict_users

user_indices = iid_partition(train_dataset, NUM_USERS)

# Build Local DataLoaders

def get_local_dataloader(dataset, indices, batch_size):
    subset = torch.utils.data.Subset(dataset, indices)
    loader = torch.utils.data.DataLoader(subset,
batch_size=batch_size, shuffle=True)
    return loader

def local_train(model, dataloader, epochs, lr, device):
    """
    Train 'model' on 'dataloader' for 'epochs' using SGD(lr=lr).
    Return: (updated_state_dict, average_training_loss)
    """
    model.train()
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=lr)

    total_samples = 0
    running_loss = 0.0

    for _ in range(epochs):
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)

            outputs = model(images)
            loss = criterion(outputs, labels)
```

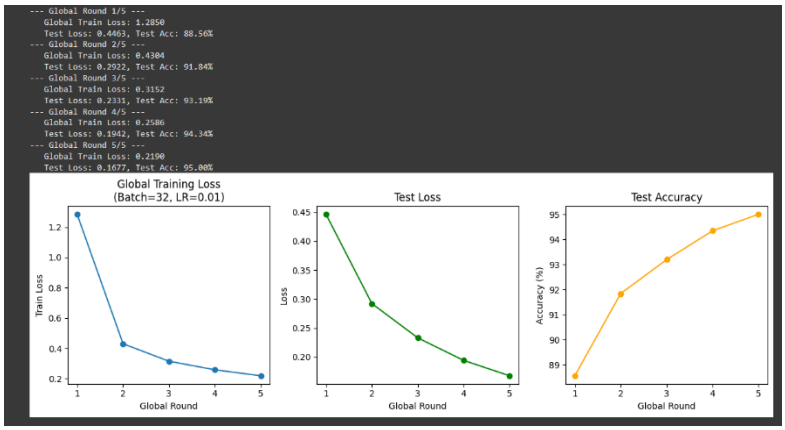
```
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            batch_size_local = images.size(0)
            running_loss += loss.item() * batch_size_local
            total_samples += batch_size_local

    avg_train_loss = running_loss / total_samples if total_samples > 0
else 0.0
    return copy.deepcopy(model.state_dict()), avg_train_loss

def fedavg(global_model, user_updates, user_data_sizes):
    """
    Weighted average of user_updates by number of samples.
    Returns a new state_dict for the global model.
    """
    new_state_dict = copy.deepcopy(global_model.state_dict())
    for key in new_state_dict.keys():
        new_state_dict[key] = 0.0

    total_data_points = sum(user_data_sizes)
    for i, state_dict_i in enumerate(user_updates):
        user_weight = user_data_sizes[i] / total_data_points
        for key in state_dict_i.keys():
            new_state_dict[key] += state_dict_i[key] * user_weight

    return new_state_dict

def evaluate(model, dataloader, device):
    """
    Evaluate model on dataloader -> return (avg_loss, accuracy).
    """
    model.eval()
    criterion = nn.CrossEntropyLoss()
    running_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)

            running_loss += loss.item() * images.size(0)
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
```

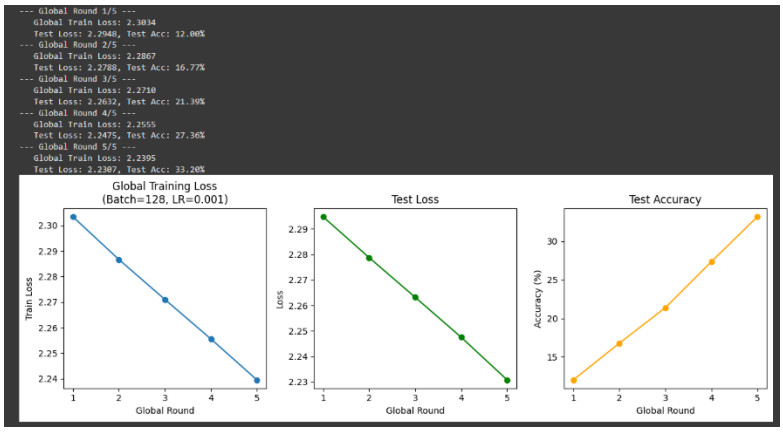
```
            total += labels.size(0)

    avg_loss = running_loss / total if total > 0 else 0.0
    accuracy = 100.0 * correct / total if total > 0 else 0.0
    return avg_loss, accuracy

# One DataLoader for the entire test set (global evaluation)
test_loader = torch.utils.data.DataLoader(test_dataset,
batch_size=1000, shuffle=False)

# Run FedAvg for Each Param Set

for ps_idx, ps in enumerate(PARAM_SETS):
    print(f"\n\n============================")
    print(f" FEDERATED RUN {ps_idx+1} of {len(PARAM_SETS)} ")
    print(f" batch_size={ps['batch_size']},
learning_rate={ps['learning_rate']}")
    print(f"============================\n")


    local_loaders = [
        get_local_dataloader(train_dataset, user_indices[u],
ps["batch_size"])
        for u in range(NUM_USERS)
    ]
    user_data_sizes = [len(user_indices[u]) for u in range(NUM_USERS)]

    # Initialize a fresh global model
    global_model = MLP().to(DEVICE)


    global_train_losses = []
    global_test_losses  = []
    global_test_accs    = []

    # Federated training loop
    for round_num in range(GLOBAL_ROUNDS):
        print(f"--- Global Round {round_num+1}/{GLOBAL_ROUNDS} ---")

        # List of local updates
        user_updates = []

        total_local_train_loss = 0.0
        total_local_samples    = 0

        # Broadcast & local train
        for user_id in range(NUM_USERS):
            # Copy global -> local
```

```
            local_model = MLP().to(DEVICE)
            local_model.load_state_dict(copy.deepcopy(global_model.sta
te_dict()))

            # Train locally
            updated_params, local_train_loss = local_train(
                local_model,
                local_loaders[user_id],
                LOCAL_EPOCHS,
                ps["learning_rate"],
                DEVICE
            )
            user_updates.append(updated_params)

            # Weighted sum for global train loss
            num_samples_user = user_data_sizes[user_id]
            total_local_train_loss += local_train_loss *
num_samples_user
            total_local_samples    += num_samples_user

        # Federated Averaging
        new_global_state = fedavg(global_model, user_updates,
user_data_sizes)
        global_model.load_state_dict(new_global_state)

        # Compute global training loss
        global_train_loss = total_local_train_loss /
total_local_samples

        # Evaluate on test set
        test_loss, test_acc = evaluate(global_model, test_loader,
DEVICE)

        # Log
        global_train_losses.append(global_train_loss)
        global_test_losses.append(test_loss)
        global_test_accs.append(test_acc)

        print(f"   Global Train Loss: {global_train_loss:.4f}")
        print(f"   Test Loss: {test_loss:.4f}, Test Acc:
{test_acc:.2f}%")


    rounds = range(1, GLOBAL_ROUNDS+1)

    plt.figure(figsize=(12,4))

    # Training Loss
```

```
    plt.subplot(1,3,1)
    plt.plot(rounds, global_train_losses, marker='o')
    plt.title(f"Global Training Loss\n(Batch={ps['batch_size']},
LR={ps['learning_rate']})")
    plt.xlabel("Global Round")
    plt.ylabel("Train Loss")

    # Test Loss
    plt.subplot(1,3,2)
    plt.plot(rounds, global_test_losses, marker='o', color='green')
    plt.title("Test Loss")
    plt.xlabel("Global Round")
    plt.ylabel("Loss")

    # Test Accuracy
    plt.subplot(1,3,3)
    plt.plot(rounds, global_test_accs, marker='o', color='orange')
    plt.title("Test Accuracy")
    plt.xlabel("Global Round")
    plt.ylabel("Accuracy (%)")

    plt.tight_layout()
    plt.show()
```

CNN





Ιδιά συμπεράσματα λαμβάνουμε και με διαφορετικό μοντέλο.

```
import torch
import torch.nn as nn
import torch.optim as optim
```

```
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

import copy
import math

%matplotlib inline

# batch_size, learning_rate
PARAM_SETS = [
    {"batch_size": 32,  "learning_rate": 0.01},
    {"batch_size": 128, "learning_rate": 0.001}
]

NUM_USERS = 10
LOCAL_EPOCHS = 1
GLOBAL_ROUNDS = 5
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", DEVICE)

# CNN model
class CNN(nn.Module):
    """
    Ένα απλό CNN για το MNIST σε ομοσπονδιακό (federated) σενάριο:
      - Δύο μπλοκ Conv+ReLU+MaxPool
      - Flatten
      - Δύο γραμμικά στρώματα (Linear) για την ταξινόμηση
    """
    def __init__(self, num_classes=10):
        super(CNN, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3,
padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),

            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3,
padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2)
        )


        self.classifier = nn.Sequential(
            nn.Linear(64*7*7, 128),
            nn.ReLU(),
            nn.Linear(128, num_classes)
```

```
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x

# Load mnist
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

train_dataset = torchvision.datasets.MNIST(
    root="./data", train=True, download=True, transform=transform
)
test_dataset = torchvision.datasets.MNIST(
    root="./data", train=False, download=True, transform=transform
)

#  Partition Training Data (IID) for 10 Users

def iid_partition(dataset, num_users=10):
    """
    Κάθε χρήστης λαμβάνει ~1/num_users του dataset, τυχαία επιλεγμένο.
    """
    num_items = len(dataset) // num_users
    all_indices = torch.randperm(len(dataset))

    dict_users = {}
    start = 0
    for user_id in range(num_users):
        end = start + num_items
        dict_users[user_id] = all_indices[start:end]
        start = end
    return dict_users

user_indices = iid_partition(train_dataset, NUM_USERS)

# Build Local DataLoaders
def get_local_dataloader(dataset, indices, batch_size):
    subset = torch.utils.data.Subset(dataset, indices)
    loader = torch.utils.data.DataLoader(subset,
batch_size=batch_size, shuffle=True)
    return loader

def local_train(model, dataloader, epochs, lr, device):
```

```
    """
    Εκπαίδευση 'model' τοπικά για 'epochs' επαναλήψεις σε
'dataloader',
    με SGD(lr=lr). Επιστρέφει (ενημερωμένο_state_dict, μέσο
training_loss).
    """
    model.train()
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=lr)

    total_samples = 0
    running_loss = 0.0

    for _ in range(epochs):
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)

            outputs = model(images)
            loss = criterion(outputs, labels)

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

            b_size = images.size(0)
            running_loss += loss.item() * b_size
            total_samples += b_size

    avg_train_loss = running_loss / total_samples if total_samples > 0
else 0.0
    return copy.deepcopy(model.state_dict()), avg_train_loss

def fedavg(global_model, user_updates, user_data_sizes):
    """
    Weighted average των τοπικών ενημερώσεων.
    Επιστρέφει ένα state_dict για το global μοντέλο.
    """
    new_state_dict = copy.deepcopy(global_model.state_dict())
    for key in new_state_dict.keys():
        new_state_dict[key] = 0.0

    total_data_points = sum(user_data_sizes)
    for i, state_dict_i in enumerate(user_updates):
        user_weight = user_data_sizes[i] / total_data_points
        for key in state_dict_i.keys():
            new_state_dict[key] += state_dict_i[key] * user_weight

    return new_state_dict
```

```
def evaluate(model, dataloader, device):
    """
    Αξιολόγηση του 'model' στο δοσμένο dataloader. Επιστρέφει
(avg_loss, accuracy).
    """
    model.eval()
    criterion = nn.CrossEntropyLoss()
    running_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)

            running_loss += loss.item() * images.size(0)
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

    avg_loss = running_loss / total if total > 0 else 0.0
    accuracy = 100.0 * correct / total if total > 0 else 0.0
    return avg_loss, accuracy

# One DataLoader for the entire test set (global evaluation)
test_loader = torch.utils.data.DataLoader(test_dataset,
batch_size=1000, shuffle=False)

for ps_idx, ps in enumerate(PARAM_SETS):
    print(f"\n\n============================")
    print(f" FEDERATED RUN {ps_idx+1} of {len(PARAM_SETS)} ")
    print(f" batch_size={ps['batch_size']},
learning_rate={ps['learning_rate']}")
    print(f"============================\n")


    local_loaders = []
    user_data_sizes = []
    for u in range(NUM_USERS):
        loader_u = get_local_dataloader(train_dataset,
user_indices[u], ps["batch_size"])
        local_loaders.append(loader_u)
        user_data_sizes.append(len(user_indices[u]))
```

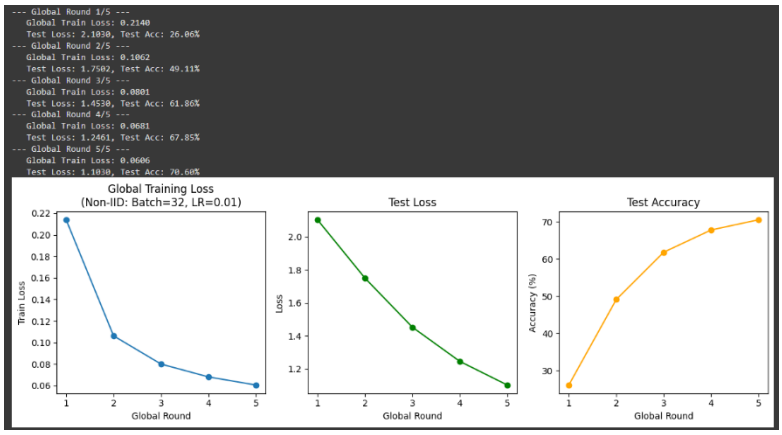
```
    global_model = CNN().to(DEVICE)


    global_train_losses = []
    global_test_losses  = []
    global_test_accs    = []

    # Federated training loop
    for round_num in range(GLOBAL_ROUNDS):
        print(f"--- Global Round {round_num+1}/{GLOBAL_ROUNDS} ---")

        user_updates = []
        total_local_loss = 0.0
        total_local_samples = 0

        # Broadcast & Local Train
        for u in range(NUM_USERS):
            # Copy global -> local
            local_model = CNN().to(DEVICE)
            local_model.load_state_dict(copy.deepcopy(global_model.sta
te_dict()))

            # Local training
            updated_params, local_train_loss = local_train(
                local_model,
                local_loaders[u],
                LOCAL_EPOCHS,
                ps["learning_rate"],
                DEVICE
            )
            user_updates.append(updated_params)

            # Weighted sum για train loss
            num_samples_user = user_data_sizes[u]
            total_local_loss += local_train_loss * num_samples_user
            total_local_samples += num_samples_user

        # FedAvg
        new_global_state = fedavg(global_model, user_updates,
user_data_sizes)
        global_model.load_state_dict(new_global_state)

        # Global training loss
        global_train_loss = total_local_loss / total_local_samples

        # Evaluate στο test set
        test_loss, test_acc = evaluate(global_model, test_loader,
DEVICE)
```

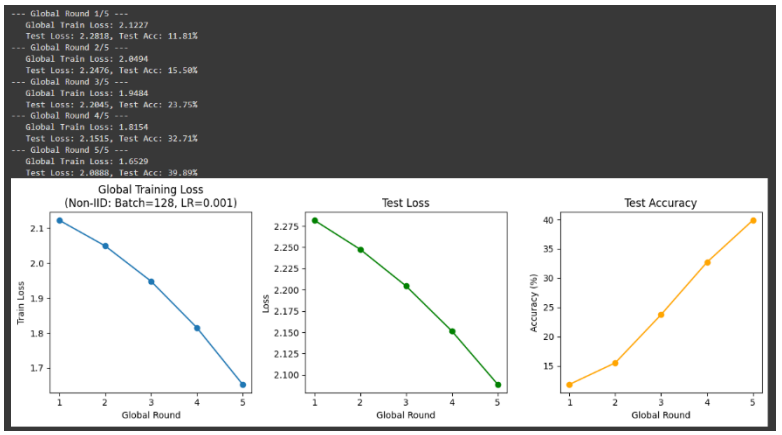
```
        global_train_losses.append(global_train_loss)
        global_test_losses.append(test_loss)
        global_test_accs.append(test_acc)

        print(f"   Global Train Loss: {global_train_loss:.4f}")
        print(f"   Test Loss: {test_loss:.4f}, Test Acc:
{test_acc:.2f}%")

    # Plot
    rounds = range(1, GLOBAL_ROUNDS+1)

    plt.figure(figsize=(12,4))

    # Training Loss
    plt.subplot(1,3,1)
    plt.plot(rounds, global_train_losses, marker='o')
    plt.title(f"Global Training Loss\n(Batch={ps['batch_size']},
LR={ps['learning_rate']})")
    plt.xlabel("Global Round")
    plt.ylabel("Train Loss")

    # Test Loss
    plt.subplot(1,3,2)
    plt.plot(rounds, global_test_losses, marker='o', color='green')
    plt.title("Test Loss")
    plt.xlabel("Global Round")
    plt.ylabel("Loss")

    # Test Accuracy
    plt.subplot(1,3,3)
    plt.plot(rounds, global_test_accs, marker='o', color='orange')
    plt.title("Test Accuracy")
    plt.xlabel("Global Round")
    plt.ylabel("Accuracy (%)")

    plt.tight_layout()
    plt.show()
```

Ερώτημα 3

MLP

Κάθε χρήστης τώρα έχει μόνο 2 από τα 10 ψηφιά. Άρα, ο καθένας εκπαιδεύει το μοντέλο για αυτά και όχι για όλα, συνεπώς θα έχουμε σίγουρα πιο αργή σύγκλιση, αφού πρέπει να συγκεντρώνουμε τις πληροφορίες από όλους τους χρήστες για διαφορετικά ψηφιά.





Από τα αποτελέσματα βλέπουμε τώρα ότι το train loss είναι παραπλανητικά μικρό. Αυτό συμβαίνει επειδή ο κάθε χρήστης έχει πολύ λιγότερα ψηφιά να κοιτάει, άρα λιγότερα δεδομένα, για αυτό έχει τώρα μικρό σφάλμα. Στην πραγματικότητα, βλέπουμε από το test ότι το σφάλμα παραμένει υψηλό (υψηλότερο από το iid) και βελτιώνεται με αργότερο ρυθμό σε σχέση με iid. Η ακρίβεια μεγαλώνει σταθερά και μάλιστα φαίνεται να συγκλίνει σε αισθητά χαμηλότερο επίπεδο από το προηγούμενο ερώτημα, αφού ο κάθε χρήστης έχει μόνο 2 ψηφιά αντί για 10. Στην δεύτερη περίπτωση, τα αποτελέσματα είναι καλυτέρα από το iid, επειδή ο μεγαλύτερος αριθμός δειγμάτων επωφελή τα περιορισμένα patterns του χρήστη.

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

import copy
import math

%matplotlib inline

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", DEVICE)
```

```
# batch_size, learning_rate
PARAM_SETS = [
    {"batch_size": 32,  "learning_rate": 0.01},
    {"batch_size": 128, "learning_rate": 0.001}
]

NUM_USERS = 10          # 10 clients
LOCAL_EPOCHS = 1        # how many epochs each user trains locally per
round
GLOBAL_ROUNDS = 5       # total number of federated rounds

# MLP model
class MLP(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=128, num_classes=10):
        super(MLP, self).__init__()
        self.net = nn.Sequential(
            nn.Flatten(),
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, num_classes)
        )
    def forward(self, x):
        return self.net(x)

# Load mnist
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

train_dataset = torchvision.datasets.MNIST(
    root="./data", train=True, download=True, transform=transform
)
test_dataset = torchvision.datasets.MNIST(
    root="./data", train=False, download=True, transform=transform
)

# Non-IID Partition
from collections import defaultdict

def non_iid_partition_2_classes(dataset, num_users=10):
    """
    Return a dict: user_id -> list of indices
    so that each user i gets only the samples of
    classes (i, (i+1) mod 10).
    """
```

```
    # Separate all samples by class
    class_indices = defaultdict(list)
    for idx, (image, label) in enumerate(dataset):
        class_indices[label].append(idx)

    # For each user i, gather the indices for classes i and (i+1) mod
10
    dict_users = {}
    for i in range(num_users):
        c1 = i % 10
        c2 = (i+1) % 10
        user_idx_list = class_indices[c1] + class_indices[c2]

        user_idx_list =
torch.tensor(user_idx_list)[torch.randperm(len(user_idx_list))]
        dict_users[i] = user_idx_list
    return dict_users

# Build the non-IID partition
user_indices = non_iid_partition_2_classes(train_dataset, NUM_USERS)

def get_local_dataloader(dataset, indices, batch_size):
    subset = torch.utils.data.Subset(dataset, indices)
    loader = torch.utils.data.DataLoader(subset,
batch_size=batch_size, shuffle=True)
    return loader

def local_train(model, dataloader, epochs, lr, device):
    """
    Train 'model' on 'dataloader' for 'epochs' using SGD(lr=lr).
    Return (updated_state_dict, average_training_loss).
    """
    model.train()
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=lr)

    total_samples = 0
    running_loss = 0.0

    for _ in range(epochs):
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)

            outputs = model(images)
            loss = criterion(outputs, labels)

            optimizer.zero_grad()
            loss.backward()
```

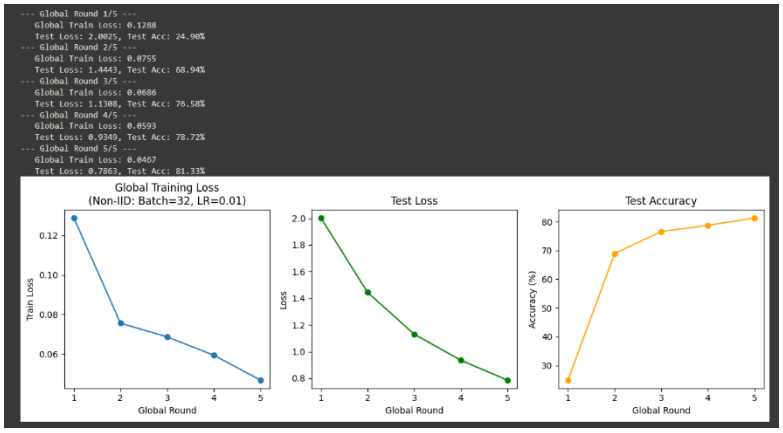
```
            optimizer.step()

            bs_local = images.size(0)
            running_loss += loss.item() * bs_local
            total_samples += bs_local

    avg_train_loss = running_loss / total_samples if total_samples > 0
else 0.0
    return copy.deepcopy(model.state_dict()), avg_train_loss

def fedavg(global_model, user_updates, user_data_sizes):
    """
    Weighted average of user updates by sample count.
    Returns a new state_dict for the global model.
    """
    new_state_dict = copy.deepcopy(global_model.state_dict())
    for key in new_state_dict.keys():
        new_state_dict[key] = 0.0

    total_data_points = sum(user_data_sizes)
    for i, state_dict_i in enumerate(user_updates):
        frac = user_data_sizes[i] / total_data_points
        for key in state_dict_i.keys():
            new_state_dict[key] += state_dict_i[key] * frac

    return new_state_dict

def evaluate(model, dataloader, device):
    """
    Return (avg_loss, accuracy%) evaluating 'model' on 'dataloader'.
    """
    model.eval()
    criterion = nn.CrossEntropyLoss()
    correct = 0
    total = 0
    total_loss = 0.0

    with torch.no_grad():
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)

            total_loss += loss.item() * images.size(0)
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)
```

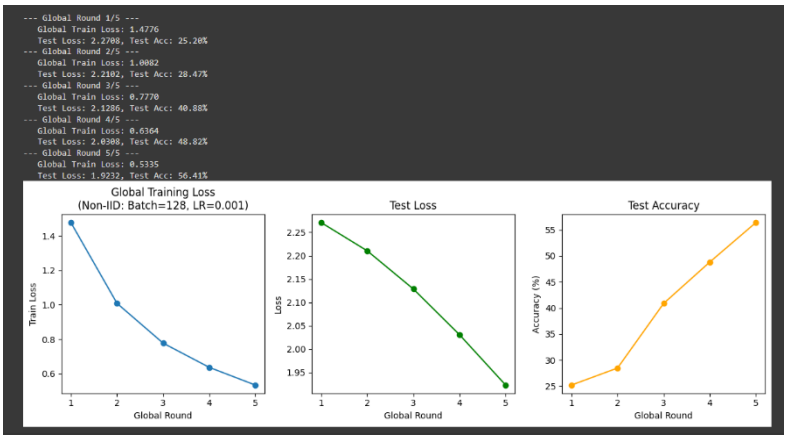
```
    avg_loss = total_loss / total if total > 0 else 0.0
    accuracy = 100.0 * correct / total if total > 0 else 0.0
    return avg_loss, accuracy

# Single test loader for the entire test set
test_loader = torch.utils.data.DataLoader(test_dataset,
batch_size=1000, shuffle=False)

for ps_i, ps in enumerate(PARAM_SETS):
    print(f"\n\n==================================")
    print(f" FEDERATED NON-IID RUN {ps_i+1} of {len(PARAM_SETS)} ")
    print(f" batch_size={ps['batch_size']},
learning_rate={ps['learning_rate']}")
    print(f"==================================\n")

    # Create local DataLoaders for each user with the given batch size
    local_loaders = []
    user_data_sizes = []
    for u in range(NUM_USERS):
        loader_u = get_local_dataloader(train_dataset,
user_indices[u], ps["batch_size"])
        local_loaders.append(loader_u)
        user_data_sizes.append(len(user_indices[u]))

    # Fresh global model
    global_model = MLP().to(DEVICE)


    global_train_losses = []
    global_test_losses  = []
    global_test_accs    = []

    # Federated training
    for round_num in range(GLOBAL_ROUNDS):
        print(f"--- Global Round {round_num+1}/{GLOBAL_ROUNDS} ---")

        user_updates = []
        total_local_loss = 0.0
        total_local_samples = 0

        # Broadcast + Local Train
        for u in range(NUM_USERS):
            # Copy global -> local
            local_model = MLP().to(DEVICE)
            local_model.load_state_dict(copy.deepcopy(global_model.sta
te_dict()))

            # Local training
```

```
            updated_params, local_loss = local_train(local_model,
local_loaders[u],
                                                     LOCAL_EPOCHS,
ps["learning_rate"], DEVICE)
            user_updates.append(updated_params)

            # Weighted sum for global training loss
            num_samples_user = user_data_sizes[u]
            total_local_loss += local_loss * num_samples_user
            total_local_samples += num_samples_user

        # FedAvg
        new_global_state = fedavg(global_model, user_updates,
user_data_sizes)
        global_model.load_state_dict(new_global_state)

        # Weighted average local training loss
        global_train_loss = total_local_loss / total_local_samples

        # Evaluate on the global test set
        test_loss, test_acc = evaluate(global_model, test_loader,
DEVICE)

        global_train_losses.append(global_train_loss)
        global_test_losses.append(test_loss)
        global_test_accs.append(test_acc)

        print(f"   Global Train Loss: {global_train_loss:.4f}")
        print(f"   Test Loss: {test_loss:.4f}, Test Acc:
{test_acc:.2f}%")


    rounds = range(1, GLOBAL_ROUNDS+1)
    plt.figure(figsize=(12,4))

    # Global Training Loss
    plt.subplot(1,3,1)
    plt.plot(rounds, global_train_losses, marker='o')
    plt.title(f"Global Training Loss\n(Non-IID:
Batch={ps['batch_size']}, LR={ps['learning_rate']})")
    plt.xlabel("Global Round")
    plt.ylabel("Train Loss")

    # Test Loss
    plt.subplot(1,3,2)
    plt.plot(rounds, global_test_losses, marker='o', color='green')
    plt.title("Test Loss")
    plt.xlabel("Global Round")
```

```
    plt.ylabel("Loss")

    # Test Accuracy
    plt.subplot(1,3,3)
    plt.plot(rounds, global_test_accs, marker='o', color='orange')
    plt.title("Test Accuracy")
    plt.xlabel("Global Round")
    plt.ylabel("Accuracy (%)")

    plt.tight_layout()
    plt.show()
```

CNN






```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

import copy
import math
from collections import defaultdict

%matplotlib inline

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", DEVICE)
```

```
# batch_size, learning_rate
PARAM_SETS = [
    {"batch_size": 32,  "learning_rate": 0.01},
    {"batch_size": 128, "learning_rate": 0.001}
]

NUM_USERS = 10          # 10 clients
LOCAL_EPOCHS = 1        # how many epochs each user trains locally per
round
GLOBAL_ROUNDS = 5       # total number of federated rounds

# MLP model
class CNN(nn.Module):
    """
    Απλό CNN:
      1) Δύο διαδοχικά Conv(3x3) + ReLU + MaxPool(2x2)
      2) Flatten
      3) Linear(64*7*7 -> 128) + ReLU
      4) Linear(128 -> 10)
    """
    def __init__(self, num_classes=10):
        super(CNN, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3,
padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),

            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3,
padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2)
        )
        self.classifier = nn.Sequential(
            nn.Linear(64*7*7, 128),
            nn.ReLU(),
            nn.Linear(128, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x

# Load mnist
transform = transforms.Compose([
    transforms.ToTensor(),
```

```
    transforms.Normalize((0.1307,), (0.3081,))
])

train_dataset = torchvision.datasets.MNIST(
    root="./data", train=True, download=True, transform=transform
)
test_dataset = torchvision.datasets.MNIST(
    root="./data", train=False, download=True, transform=transform
)

# Non-IID Partition
def non_iid_partition_2_classes(dataset, num_users=10):
    """
    Για κάθε i, ο χρήστης i βλέπει κλάσεις (i) και ((i+1)%10).
    """
    class_indices = defaultdict(list)
    for idx, (image, label) in enumerate(dataset):
        class_indices[label].append(idx)

    dict_users = {}
    for i in range(num_users):
        c1 = i % 10
        c2 = (i+1) % 10
        user_idx_list = class_indices[c1] + class_indices[c2]
        user_idx_list =
torch.tensor(user_idx_list)[torch.randperm(len(user_idx_list))]
        dict_users[i] = user_idx_list
    return dict_users

user_indices = non_iid_partition_2_classes(train_dataset, NUM_USERS)

def get_local_dataloader(dataset, indices, batch_size):
    subset = torch.utils.data.Subset(dataset, indices)
    loader = torch.utils.data.DataLoader(subset,
batch_size=batch_size, shuffle=True)
    return loader

def local_train(model, dataloader, epochs, lr, device):
    """
    Εκπαίδευση 'model' τοπικά για 'epochs' σε 'dataloader' με
SGD(lr=lr).
    Επιστρέφει: (updated_state_dict, average_training_loss)
    """
    model.train()
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=lr)

    total_samples = 0
```

```
    running_loss = 0.0

    for _ in range(epochs):
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)

            outputs = model(images)
            loss = criterion(outputs, labels)

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

            bs_local = images.size(0)
            running_loss += loss.item() * bs_local
            total_samples += bs_local

    avg_train_loss = running_loss / total_samples if total_samples > 0
else 0.0
    return copy.deepcopy(model.state_dict()), avg_train_loss

def fedavg(global_model, user_updates, user_data_sizes):
    """
    Weighted average του local state_dicts κάθε χρήστη, με βάση το
μέγεθος δεδομένων.
    """
    new_state_dict = copy.deepcopy(global_model.state_dict())
    for key in new_state_dict.keys():
        new_state_dict[key] = 0.0

    total_data_points = sum(user_data_sizes)
    for i, state_dict_i in enumerate(user_updates):
        frac = user_data_sizes[i] / total_data_points
        for key in state_dict_i.keys():
            new_state_dict[key] += state_dict_i[key] * frac

    return new_state_dict

def evaluate(model, dataloader, device):
    """
    Επιστρέφει (avg_loss, accuracy) στο 'dataloader'.
    """
    model.eval()
    criterion = nn.CrossEntropyLoss()
    correct = 0
    total = 0
    total_loss = 0.0
```

```
    with torch.no_grad():
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)

            total_loss += loss.item() * images.size(0)
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

    avg_loss = total_loss / total if total > 0 else 0.0
    accuracy = 100.0 * correct / total if total > 0 else 0.0
    return avg_loss, accuracy

# Test loader
test_loader = torch.utils.data.DataLoader(test_dataset,
batch_size=1000, shuffle=False)

for ps_i, ps in enumerate(PARAM_SETS):
    print(f"\n\n==================================")
    print(f" FEDERATED NON-IID RUN {ps_i+1} of {len(PARAM_SETS)} ")
    print(f" batch_size={ps['batch_size']},
learning_rate={ps['learning_rate']}")
    print(f"==================================\n")

    # Local loaders
    local_loaders = []
    user_data_sizes = []
    for u in range(NUM_USERS):
        loader_u = get_local_dataloader(train_dataset,
user_indices[u], ps["batch_size"])
        local_loaders.append(loader_u)
        user_data_sizes.append(len(user_indices[u]))

    # Global model
    global_model = CNN().to(DEVICE)

    global_train_losses = []
    global_test_losses  = []
    global_test_accs    = []

    for round_num in range(GLOBAL_ROUNDS):
        print(f"--- Global Round {round_num+1}/{GLOBAL_ROUNDS} ---")

        user_updates = []
        total_local_loss = 0.0
        total_local_samples = 0
```

```
        # Broadcast & Local Train
        for u in range(NUM_USERS):
            local_model = CNN().to(DEVICE)
            local_model.load_state_dict(copy.deepcopy(global_model.sta
te_dict()))

            updated_params, local_loss = local_train(
                local_model,
                local_loaders[u],
                LOCAL_EPOCHS,
                ps["learning_rate"],
                DEVICE
            )
            user_updates.append(updated_params)

            n_samples = user_data_sizes[u]
            total_local_loss += local_loss * n_samples
            total_local_samples += n_samples

        # FedAvg
        new_global_state = fedavg(global_model, user_updates,
user_data_sizes)
        global_model.load_state_dict(new_global_state)

        # Training loss
        global_train_loss = total_local_loss / total_local_samples

        # Evaluate on global test set
        test_loss, test_acc = evaluate(global_model, test_loader,
DEVICE)

        global_train_losses.append(global_train_loss)
        global_test_losses.append(test_loss)
        global_test_accs.append(test_acc)

        print(f"   Global Train Loss: {global_train_loss:.4f}")
        print(f"   Test Loss: {test_loss:.4f}, Test Acc:
{test_acc:.2f}%")

    rounds = range(1, GLOBAL_ROUNDS+1)
    plt.figure(figsize=(12,4))

    # Global Training Loss
    plt.subplot(1,3,1)
    plt.plot(rounds, global_train_losses, marker='o')
    plt.title(f"Global Training Loss\n(Non-IID:
Batch={ps['batch_size']}, LR={ps['learning_rate']})")
```

```
    plt.xlabel("Global Round")
    plt.ylabel("Train Loss")

    # Test Loss
    plt.subplot(1,3,2)
    plt.plot(rounds, global_test_losses, marker='o', color='green')
    plt.title("Test Loss")
    plt.xlabel("Global Round")
    plt.ylabel("Loss")

    # Test Accuracy
    plt.subplot(1,3,3)
    plt.plot(rounds, global_test_accs, marker='o', color='orange')
    plt.title("Test Accuracy")
    plt.xlabel("Global Round")
    plt.ylabel("Accuracy (%)")

    plt.tight_layout()
    plt.show()
```

Συμπέρασμα: Το μικρό batch size με το μεγάλο learning rate επιφέρει τα καλυτέρα αποτελέσματα λόγω των συχνών update των weight. Προφανώς, καλυτέρα αποτελέσματα έχουμε με κεντρική μάθηση, μετα με federal iid και τέλος με federal non-iid.