Τσαντίκης Γεώργιος
ΑΕΜ: 10722

Αναφορά 1ης υποχρεωτικής εργασίας στο μάθημα «ΝΕΥΡΩΝΙΚΑ ΔΙΚΤΥΑ- ΒΑΘΙΑ ΜΑΘΗΣΗ»

ΠΕΡΙΕΧΟΜΕΝΑ:

1. Εισαγωγή

Στην παρούσα εργασία πραγματοποιήθηκε η υλοποίηση από την αρχή (**from scratch**), με κώδικα **Python**, ενός πολυεπίπεδου νευρωνικού δικτύου (**Multi-Layer Perceptron** - MLP) που εκπαιδεύεται με τον αλγόριθμο **back-propagation** για την κατηγοριοποίηση εικόνων της βάσης δεδομένων **CIFAR-10**. Η CIFAR-10 περιέχει 60.000 έγχρωμες (RGB) εικόνες μεγέθους 32x32 pixels, κατανεμημένες σε 10 διαφορετικές κατηγορίες /κλάσεις (π.χ., σκύλος, φορτηγό, αεροπλάνο), από τις οποίες 50.000 εικόνες είναι δείγματα εκπαίδευσης και 10.000 δείγματα ελέγχου.

Η μελέτη επικεντρώνεται στην ανάπτυξη και βελτιστοποίηση του MLP χωρίς τη χρήση έτοιμων βιβλιοθηκών (όπως TensorFlow ή PyTorch), με σκοπό την κατανόηση σε βάθος των μηχανισμών εκπαίδευσης και λειτουργίας ενός νευρωνικού δικτύου. Όσον αφορά τις εισόδους, δεν υπέστησαν κάποια επεξεργασία (π.χ., PCA ή Data Augmentation), επιτρέποντας την αξιολόγηση της απόδοσης σε ωμές εικόνες. Χρησιμοποιήθηκαν τεχνικές όπως:

- **Κανονικοποίηση** εισόδων και **One-Hot Encoding** για τα τις κλάσεις.
- Συναρτήσεις ενεργοποίησης **ReLU** (Rectified Linear Unit) και **Softmax**.
- Εκπαίδευση μέσω **Batches** και **Gradient Descent**.
- Ρύθμιση υπερπαραμέτρων εκπαίδευσης (**learning rate, epochs**, **αριθμός** και **πλήθος επιπέδων νευρώνων**, **batch size**)

Στόχοι της εργασίας είναι να διερευνηθεί η επίδραση διαφορετικών ρυθμίσεων (υπεραπαραμέτρους) στην απόδοση του MLP, να παρουσιαστούν παραδείγματα ορθής και εσφαλμένης κατηγοριοποίησης και να συγκριθεί η **απόδοση** του νευρωνικού σε σχέση με την κατηγοριοποίηση πλησιέστερου γείτονα (Nearest Neighbor) και πλησιέστερου κέντρου κλάσης (Nearest Class Centroid) της ***ενδιάμεσης εργασίας*****.**

2. Περιγραφή αλγορίθμων – Ανάλυση Κώδικα

Βιβλιοθήκες

Οι παρακάτω βιβλιοθήκες αποτελούν απαραίτητα εργαλεία για την κάλυψη των απαιτήσεων της εργασίας :

- **import numpy as np** → για την αποδοτική διαχείριση αριθμητικών υπολογισμών και πολυδιάστατων πινάκων (x_train, x_test κτλ)
- **import time** → για την μέτρηση του χρόνου εκπαίδευσης
- **import matplotlib. pyplot as plt** → για τη δημιουργία γραφικών παραστάσεων για την οπτικοποίηση των αποτελεσμάτων (loss per epoch)
- **import pickle** → για την αποθήκευση και ανάκτηση των δεδομένων σε δυαδική μορφή
- **import requests** → για την αποστολή αιτήματος HTTP, με σκοπό την λήψη του dataset CIFAR-10 από το διαδίκτυο
- **import tarfile** → για την αποσυμπίεση των δεδομένων
- **import os**→ για τον χειρισμό λειτουργιών του συστήματος αρχείων (έλεγχος ύπαρξης αρχείων κτλ.)

2.1 – Download και extract του dataset CIFAR-10

Η παρακάτω συνάρτηση εξυπηρετεί τη λήψη από το διαδίκτυο και την αποσυμπίεση του dataset σε περίπτωση που το συμπιεσμένο αρχείο με όνομα *"cifar-10-python.tar.gz"* δεν υπάρχει στον τοπικό φάκελο, στον οποίο εξάγει το περιεχόμενο του σε δυαδική μορφή. Πλέον, τα δεδομένα του CIFAR-10 είναι έτοιμα για περαιτέρω επεξεργασία σε μη συμπιεσμένη μορφή στον φάκελο "cifar-10-batches-py".

```
# Download και extract του dataset CIFAR-10
def download_and_extract_cifar10():
    url = "https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz"
    archive_name = "cifar-10-python.tar.gz"
    if not os.path.exists(archive_name):
        print("Downloading CIFAR-10 dataset...")
        with requests.get(url, stream=True) as r:
            with open(archive_name, "wb") as file: # ανοίγει για γράψιμο
                file.write(r.content)
    if not os.path.exists("cifar-10-batches-py"):
        print("Extracting dataset...")
        with tarfile.open(archive_name, "r:gz") as tar:
            tar.extractall()
```

2.2 – Φόρτωση και προεπεξεργασία του dataset

Η βοηθητική συνάρτηση " *unpickle(file)* " διαβάζει τα δεδομένα από αρχεία που είναι αποθηκευμένα σε δυαδική μορφή και χρησιμοποιείται για τη φόρτωση των δεδομένων του

CIFAR-10 ύστερα από την λήψη και αποθήκευση τους στο φάκελο *"./cifar-10-batches-py"* με χρήση της συνάρτησης " *download_and_extract_cifar10()* " της παραγράφου **2.1**. Τα δεδομένα εκπαίδευσης περιλαμβάνονται σε πέντε batches (αρχεία data_batch_1 έως data_batch_5), με κάθε **batch** (βλ. *2.7*) να περιέχει τις εικόνες και τις αντίστοιχες ετικέτες τους, δηλαδή τις κατηγορίες/κλάσεις στις οποίες ανήκουν. Οι εικόνες φορτώνονται και αποθηκεύονται στον πίνακα ***x*** και οι ετικέτες στον πίνακα ***y***.
Έπειτα πραγματοποιείται η **κανονικοποίηση**, που μετατρέπει τις τιμές των εικόνων από το εύρος [0, 255] στο εύρος [0, 1], διαιρώντας με το μέγιστο (255.0), μειώνοντας τη διακύμανση μεταξύ των εισόδων. Έτσι διευκολύνεται η σύγκλιση κατά την εκπαίδευση, καθώς οι βελτιώσεις στα βάρη γίνονται πιο ομαλά**.** Επιπλέον, με αυτόν τον τρόπο αποφεύγουμε το overflow error στο exponential της **SoftMax** (βλ. *2.4*). Οι NumPy Arrays για τα δείγματα αποθηκεύονται στις μεταβλητές ***x_train*** και ***y_train***. Αντίστοιχη επεξεργασία υφίστανται τα δεδομένα ελέγχου που περιέχονται στο αρχείο "*test_batch*". Τελικά, η συνάρτηση επιστρέφει τις μεταβλητές "*x_train*" (δείγματα εκπαίδευσης), "*y_train*" (ετικέτες των x_train), "*x_test*" (δείγματα ελέγχου), "*y_test*" (ετικέτες των x_test).

```python
# Φόρτωση των δεδομένων του dataset
def load_cifar10():
    def unpickle(file):
        with open(file, 'rb') as fo:
            data = pickle.load(fo, encoding='bytes')
        return data

    download_and_extract_cifar10()
    data_path = "./cifar-10-batches-py/"
    x, y = [], []

    for i in range(1, 6):
        batch = unpickle(os.path.join(data_path, f"data_batch_{i}"))
        x.append(batch[b'data'])
        y.extend(batch[b'labels'])

    x_train = np.vstack(x).astype(np.float32) / 255.0 # Κανονικοποίηση τιμών
    y_train = np.array(y)

    test_batch = unpickle(os.path.join(data_path, "test_batch"))
    x_test = test_batch[b'data'].astype(np.float32) / 255.0
    y_test = np.array(test_batch[b'labels'])

    return x_train, y_train, x_test, y_test
```

*Η **κλάση** "**MLP**" υλοποιεί ένα πολυεπίπεδο νευρωνικό δίκτυο (MLP) το οποίο χρησιμοποιείται για ταξινόμηση δεδομένων. Οι μέθοδοι που παρουσιάζονται στις παραγράφους 2.3 έως 2.8 ανήκουν σε αυτήν την κλάση:*

2.3 – Αρχικοποίηση παραμέτρων

Ο constructor "*def __init__(self, layer_sizes, learning_rate, batch_size)*" χρησιμοποιείται για την αρχικοποίηση παραμέτρων. Η μεταβλητή "*layer_sizes*" ορίζει το πλήθος των επιπέδων και τον αριθμό των νευρώνων σε κάθε επίπεδο :

```
layer_sizes = [input_size] + hidden_sizes + [output_size]
```

με "*input_size*" τον αριθμό νευρώνων που ισούται με την διάσταση της εικόνας (32x32x3) και αποτελεί το επίπεδο εισόδου, "*output_size*" τον αριθμό νευρώνων που ισούται με το πλήθος των κατηγοριών/κλάσεων (10) και "*hidden_sizes*" την λίστα που περιέχει τις πληροφορίες για τους νευρώνες του κρυφού επιπέδου. Επιπρόσθετα, έχουμε την αρχικοποίηση των biases σε μηδενικές τιμές και των βαρών με χρήση της μεθόδου *He Initialization*: $w \sim N(0, \sqrt{\frac{2}{n}})$ που προσαρμόζει τα βάρη ανάλογα με το μέγεθος των επιπέδων (n → ο αριθμός των νευρώνων στο προηγούμενο επίπεδο), είναι κατάλληλη για δίκτυα με συνάρτηση ενεργοποίησης **ReLU** και σκοπεύει στην αποφυγή φαινομένων vanishing ή exploding gradients.

```
def __init__(self, layer_sizes, learning_rate, batch_size):
    self.layer_sizes = layer_sizes
    self.learning_rate = learning_rate # Ρυθμός μάθησης
    self.weights = []
    self.biases = []
    self.batch_size = batch_size # Αριθμός δειγμάτων του batch

    # Αρχικοποίηση βαρών και biases για κάθε επίπεδο
    for i in range(len(layer_sizes) - 1):
        stddev = np.sqrt(2 / layer_sizes[i]) #(He Normal Initialization)
        self.weights.append(np.random.randn(layer_sizes[i], layer_sizes[i + 1]) * stddev)
        self.biases.append(np.zeros((1, layer_sizes[i + 1])))
```

2.4 – Συναρτήσεις ενεργοποίησης

Η μέθοδος "*relu(self, u)*" υλοποιεί την συνάρτηση ενεργοποίησης **ReLU** που χρησιμοποιείται στα κρυφά επίπεδα στο forward pass και επιστρέφει τη μέγιστη τιμή μεταξύ του 0 και της εισόδου u (απλό υπολογιστικά) . Παράλληλα, βοηθά στην αποφυγή του φαινομένου vanishing gradients που προκύπτει όταν η παράγωγος της συνάρτησης ενεργοποίησης γίνεται πολύ μικρή, καθώς η μέθοδος "*relu_derivative(self, u)*" που υλοποιεί την παράγωγο της και χρησιμοποιείται για τον υπολογισμό των σφαλμάτων (delta) κατά το backward pass, για θετικές τιμές είναι σταθερή (1) αλλιώς 0. Επίσης όπως θα παρουσιαστεί στη συνέχεια φάνηκε να εμφανίζει καλύτερες επιδόσεις από την **Sigmoid**.

Η συνάρτηση ενεργοποίησης **SoftMax** που ορίζεται ως: $SoftMax_i(x) = \frac{e^{x_i}}{\sum_j^K e^{x_j}}$, όπου x_i η είσοδος της SoftMax για την κατηγορία i, με K το πλήθος των κατηγοριών, χρησιμοποιείται στο επίπεδο εξόδου σε συνδυασμό με την **Cross-Entropy Loss Function** (βλ. *2.8*) και εξασφαλίζει ότι η έξοδος του μοντέλου μπορεί να ερμηνευθεί ως πιθανότητα για κάθε κατηγορία, ώστε το άθροισμα τους να είναι ίσο με 1. Η κατηγορία με τη μεγαλύτερη πιθανότητα είναι η τελική πρόβλεψη (δηλαδή ένας νευρώνας είναι σωστός κάθε φορά).

```
def relu(self, u):
    return np.maximum(0, u)

def relu_derivative(self, u):
    return (u > 0).astype(float)

def softmax(self, u):
    exp_u = np.exp(u - np.max(u, axis=1, keepdims=True)) #Αποφυγή overflow
    return exp_u / np.sum(exp_u, axis=1, keepdims=True)
```

2.5 – Forward Pass

Η μέθοδος "*forward_pass(self, x)*" υλοποιεί την εμπρόσθια τροφοδότηση για το MLP,ενώ αποθηκεύει κρίσιμες πληροφορίες (u και outputs) για το backward pass. Η μεταβλητή "*self.outputs()*" αποθηκεύει τις εξόδους όλων των επιπέδων και η "*self.u_values()*" τις τιμές (u) πριν την εφαρμογή της συνάρτησης ενεργοποίησης. Μέσω του *for loop* που επαναλαμβάνεται για κάθε επίπεδο υπολογίζεται το u ως : $\boldsymbol{u_i = O_{i-1} \cdot w_i + b_i}$, όπου O_{i-1} είναι η έξοδος του προηγούμενου επιπέδου, w_i τα συναπτικά βάρη και b_i τα biases για το τρέχον επίπεδο. Στο *for loop* περιλαμβάνεται το *if – else,* σύμφωνα με το οποίο αν είναι το εξωτερικό επίπεδο του δικτύου, για τον υπολογισμό της εξόδου της συνάρτησης ενεργοποίησης (outputs) χρησιμοποιείται η SoftMax, ενώ αν είναι κρυφό επίπεδο χρησιμοποιείται η ReLU. Στο τέλος επιστρέφεται η έξοδος του τελευταίου επιπέδου, η οποία είναι η τελική πρόβλεψη του δικτύου (predictions).

```
def forward_pass(self, x):
    self.outputs = [x] #Οι έξοδοι των συναρτήσεων ενργοποίησης (x τα inputs)
    self.u_values = [] # Οι τιμές πριν την συνάρτηση ενεργοποίησης

    for i in range(len(self.weights)):
        u = np.dot(self.outputs[-1], self.weights[i]) + self.biases[i]
        self.u_values.append(u) # αποθήκευση των u
        if i == len(self.weights) - 1:
            output = self.softmax(u)
        else:
            output = self.relu(u)
        self.outputs.append(output)
    return self.outputs[-1]
```

2.6 – Backward Pass

Η μέθοδος "*backward_pass(self, y_true)*" υλοποιεί τον αλγόριθμο **back-propagation** για την εκπαίδευση του MLP που είναι απαραίτητη για την βελτιστοποίηση των παραμέτρων (βάρη και biases) με σκοπό την βελτίωση της απόδοσης του δικτύου.Η μεταβλητή m προσαρμόζεται στο πραγματικό μέγεθος του batch σε κάθε βήμα και η $deltas$ είναι μία λίστα που αποθηκεύει τις κλίσεις/δ ("σφάλματα") για κάθε επίπεδο (*κανονικά η έννοια σφάλμα είναι μόνο*

για το εξωτερικό επίπεδο για το οποίο έχω στόχο). Στο εξωτερικό επίπεδο η κλίση υπολογίζεται ως: $\delta = O_{output} - y_{true}$, όπου O_{output} η τελική πρόβλεψη του δικτύου και y_{true} οι πραγματικές ετικέτες (labels) των δειγμάτων (η SoftMax και η Cross-Entropy Loss έχουν συνδυαστεί με τέτοιο τρόπο ώστε οι παράγωγοι της SoftMax "ενσωματώνονται" φυσικά στους υπολογισμούς). Στο πρώτο *for loop,* όπου η επανάληψη γίνεται από το προτελευταίο επίπεδο προς το επίπεδο εισόδου, η **κλίση** υπολογίζεται ως: $\boldsymbol{\delta_l = \left(\delta_{l+1} \cdot w_{l+1}^T\right) \cdot \varphi'(u_l)}$ αξιοποιώντας τις ιδιότητες των πινάκων που προφέρει η NumPy και είναι ισοδύναμο με το: $\boldsymbol{\delta_i^{(k)}(l) = \varphi'\left(u_i^{(k)}(l)\right) \cdot \sum_{k=1}^{N(l+1)} w_{ki}(l+1) \cdot \delta_i(l+1)}$, που παρουσιάστηκε στο μάθημα. Άρα για τον υπολογισμό της κλίσης στο εσωτερικό επίπεδο l (δ_l) πολλαπλασιάζεται το σύνολο των κλίσεων (επίδραση των σφαλμάτων) του επόμενου επιπέδου (δ_{l+1}) με τα αντίστοιχα συναπτικά βάρη (w_{l+1}), που γυρίζουν πίσω, και την παράγωγο της ReLU. Στο δεύτερο *for loop* που επαναλαμβάνεται για κάθε επίπεδο, υπολογίζονται οι παράγωγοι του συνολικού "σφάλματος" για τα βάρη ως: $\Delta w_l = \frac{1}{m} \cdot (\delta_l \cdot O_{l-1})$ (όπου O_{l-1}, οι έξοδοι του προηγούμενου επιπέδου) και για τα biases ως: $\Delta b_l = \frac{1}{m} \cdot \sum_{k=1}^{m} \delta_k$ (διαιρούμε με m για να υπολογιστεί το μέσο σφάλμα για το batch). Τελικά τα βάρη και τα biases ανανεώνονται σύμφωνα με τον κανόνα Gradient Descent, πολλαπλασιάζοντας τα Δw_l και Δb_l με τον ρυθμό μάθησης (*learning_rate*) και τα αφαιρώ επειδή έχω ελαχιστοποίηση και ακολουθώ την κλίση ανάποδα, αφού δεν έχω συμπεριλάβει το -1 στον υπολογισμό των Δw_l και Δb_l . Ο αντίστοιχος αλγόριθμος που παρουσιάζεται στις διαφάνειες είναι : $w_{ij}(k+1) = w_{ij}(k) + \beta \cdot \delta_i^{(k)} \cdot O_j^{(k)}$

```
def backward_pass(self, y_true): # BP
    m = y_true.shape[0]; # Προσαρμόζεται στο πραγματικό μέγεθος του batch
    deltas = [self.outputs[-1] - y_true] # Σφάλμα στο output layer

    for i in range(len(self.weights) - 2, -1, -1):
        delta = np.dot(deltas[0], self.weights[i + 1].T) * self.relu_derivative(self.u_values[i])
        deltas.insert(0, delta) # Αποθήκευση σφαλμάτων από output έως input layer

    for i in range(len(self.weights)):
        dw = np.dot(self.outputs[i].T, deltas[i]) / m
        db = np.sum(deltas[i], axis=0, keepdims=True) / m
        self.weights[i] -= self.learning_rate * dw
        self.biases[i] -= self.learning_rate * db
```

2.7 – Προβλέψεις

Μέσω της συνάρτησης "*predict(self, x)*" το δίκτυο εκτελέι την μέθοδο "*forward_pass()*" (παρ. 2.5) για όλα τα δείγματα που εμπεριέχονται στο x και αποθηκεύει το αποτέλεσμα στην μεταβλητή "*predictions*" που είναι ένας πίνακας μεγέθους N x C (N: # of samples, C: # of classes) . Κάθε γραμμή του "*predictions*" περιέχει τις πιθανότητες που προβλέπει το δίκτυο για τις κατηγορίες του αντίστοιχου δείγματος. Τελικά η συνάρτηση επιστρέφει έναν μονοδιάστατο πίνακα με N στοιχεία, που περιέχει την πρόβλεψη κατηγορίας (κατηγορία με τη μέγιστη πιθανότητα) για κάθε δείγμα με την "*np.argmax*" να βρίσκει τον δείκτη του μέγιστου στοιχείου σε κάθε γραμμή του πίνακα "*predictions*".

```
def predict(self, x):
    predictions = self.forward_pass(x)
    return np.argmax(predictions, axis=1)
```

2.8 – Εκπαίδευση

Η μέθοδος "*train(self, x_train, y_train, x_test, y_test, epochs, batch_size, initial_lr, wait, lr_decay)*" υλοποιεί τη διαδικασία εκπαίδευσης του MLP με χρήση των μεθόδων "*forward_pass()*" , "*backward_pass()*" και "*predict()*"από τις παραγράφους *2.5, 2.6* και *2.7* αντίστοιχα. Παράλληλα, σε αυτήν με χρήση *for loop* πραγματοποιείται τυχαία αναδιάταξη των δεδομένων εκπαίδευσης ("shuffle the training data") για κάθε εποχή ώστε να μην υπερεκπαιδεύεται το δίκτυο στα τελευταία δείγματα του batch κάνοντας **batch training**, με σκοπό την αποφυγή καλής προσαρμογής και κατ' επέκταση την γενίκευση του μοντέλου**.** Ακόμη, αρχικοποιούνται τα στατιστικά που θα χρησιμοποιηθούν για την συσσώρευση πληροφοριών κατά τη διάρκεια της εποχής:

```
for epoch in range(epochs):
    # Shuffle the training data
    indices = np.arange(num_samples)  # Δημιουργία πίνακα δεικτών
    np.random.shuffle(indices)  # Τυχαίο shuffling των δεικτών
    x_train = x_train[indices]
    y_train = y_train[indices]

    # Αρχικοποίηση μέσου loss και accuracy για κάθε εποχή
    epoch_train_loss, epoch_train_accuracy = 0, 0
    epoch_test_loss, epoch_test_accuracy = 0, 0
    num_batches =0 # το πλήθος των batches
```

Προηγείται του *for loop* η αρχικοποίηση των μεταβλητών που θα χρειαστούν:

```
self.train_loss_history = [] # Αποθήκευση ιστορικού απωλειών training
self.test_loss_history = [] # και test
self.learning_rate = initial_lr # Αρχικό learning rate
num_samples = x_train.shape[0] # Συνολικός αριθμός δειγμάτων του training set
best_loss = float('inf') # Αρχικοποίηση του καλύτερο loss με άπειρο
no_improvement_epochs = 0 # Αριθμός διαδοχικών εποχών χωρίς βελτίωση στο loss
y_test_onehot = one_hot_encode(y_test, 10) # One-hot encoding για το y_test
```

Μέσα στο *for loop* που φαίνεται παραπάνω και διατρέχει τις εποχές (epochs) υπάρχει ένα *for loop* που αφορά το **training** στο οποίο για κάθε **batch** δεδομένων εκτελείται η "*forward_pass()*" για την πρόβλεψη των πιθανοτήτων εξόδου και η "*backward_pass()*" για την ανανέωση βαρών και biases των δεδομένων του batch. Επιπλέον, υπολογίζονται τα ποσοστά επιτυχίας στα στάδια της εκπαίδευσης (training). Το "*y_batch*" είναι σε μορφή **one-hot encoding** (βλ. *2.9*) και η "*np.argmax*" εξάγει τον δείκτη της κατηγορίας για κάθε δείγμα αποθηκεύοντας τον στο "*train_labels*". Ύστερα, υπολογίζεται ο μέσος όρος των σωστών προβλέψεων και αποθηκεύεται στο "*train_accuracy*". Για τον υπολογισμό της συνάρτησης απώλειας για τα κάθε batch της εποχής χρησιμοποιείται η **Cross-Entropy Loss**, που είναι κατάλληλη για προβλήματα ταξινόμησης με SoftMax στο επίπεδο εξόδου και ορίζεται ως:

$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij} \cdot \log(y_{pred,ij})$. Η SoftMax μετατρέπει τις εξόδους του δικτύου σε πιθανότητες που αθροίζονται σε 1. Η Cross-Entropy Loss αξιολογεί πόσο καλά αυτές οι πιθανότητες συμφωνούν με τις πραγματικές ετικέτες και αν η πρόβλεψη πλησιάζει την πραγματική κατηγορία (υψηλή πιθανότητα για το σωστό label), το loss είναι μικρό (επιβραβεύει τις σωστές προβλέψεις), ενώ αν η πρόβλεψη αποκλίνει από την πραγματική κατηγορία, το loss είναι μεγάλο(τιμωρεί τις λανθασμένες). Το μέσο *"train_loss"* και *"train_accuracy"* υπολογίζονται διαιρώντας τα συνολικά αποτελέσματα με τον αριθμό των batches. Αντίστοιχη διαδικασία ακολουθείται και για το test set με τη διαφορά ότι δεν χρησιμοποιούνται batches, καθώς υπολογίζονται στο σύνολο αξιολόγησης (*x_test, y_test*) για την αξιολόγηση της γενίκευσης του μοντέλου. Επιπρόσθετα εφαρμόζεται **δυναμική προσαρμογή** του *"**learning_rate**"* σύμφωνα με την οποία αν το *"train_loss"* της εποχής είναι μικρότερο από το *"best_loss"*, αυτό ενημερώνεται, και ο μετρητής *"no_improvement_epochs"* μηδενίζεται. Αν το *"train_loss"* δεν βελτιωθεί για αριθμό εποχών ίσο με *"wait"* και $learning\_rate > 10^{-6}$, το *"learning_rate"* μειώνεται πολλαπλασιάζοντάς το με *"lr_decay"*.

```
for i in range(0, num_samples, batch_size):
    x_batch = x_train[i:i + batch_size] # Batch δεδομένων εισόδου
    y_batch = y_train[i:i + batch_size] # Batch ετικετών

    # Υπολογισμός train loss και accuracy για το batch
    train_loss = -np.mean(np.sum(y_batch * np.log(self.forward_pass(x_batch) + 1e-8), axis=1))
    epoch_train_loss += train_loss

    train_pred = self.predict(x_batch)
    train_labels = np.argmax(y_batch, axis=1)
    train_accuracy = np.mean(train_pred == train_labels)
    epoch_train_accuracy += train_accuracy

    # Backward pass για το batch
    self.backward_pass(y_batch)
    num_batches += 1

# Υπολογισμός συνολικού train loss και accuracy για την εποχή
epoch_train_loss /= num_batches
epoch_train_accuracy /= num_batches

# Υπολογισμός του test loss και accuracy για την εποχή
epoch_test_loss = -np.mean(np.sum(y_test_onehot * np.log(self.forward_pass(x_test) + 1e-8), axis=1))

test_pred = self.predict(x_test)
test_accuracy = np.mean(test_pred == y_test)

# Αποθήκευση των τιμών των απωλειών
self.train_loss_history.append(epoch_train_loss)
self.test_loss_history.append(epoch_test_loss)

# Αν το train loss βελτιώθηκε, ενημέρωση του καλύτερου loss
if epoch_train_loss < best_loss:
    best_loss = epoch_train_loss
    no_improvement_epochs = 0
else:
    no_improvement_epochs += 1
    # Μείωση του learning rate αν δεν βελτιωθεί για αριθμό εποχών = wait
    if no_improvement_epochs >= wait and self.learning_rate > 1e-6:
            self.learning_rate *= lr_decay
            no_improvement_epochs = 0  # Επαναφορά του μετρητή
```

Τελικά, εκτυπώνονται τα στατιστικά (*train loss και accuracy, test loss και accuracy, τρέχον learning rate*) στο τέλος κάθε εποχής:

```
print(f"Epoch {epoch+1}, Train loss: {epoch_train_loss:.4f},  Test Loss: {epoch_test_loss:.4f},"
f" Learning Rate: {self.learning_rate:.6f},"
f"Train Acc: {epoch_train_accuracy * 100:.2f}%, Test Acc: {test_accuracy * 100:.2f}%")
```

Οι συναρτήσεις που παρουσιάζονται στις παραγράφους 2.9 έως 2.11 δεν ανήκουν στην κλάση MLP:

2.9 – One-Hot Encoding

Η συνάρτηση "*one_hot_encode(labels, num_classes)*" μετατρέπει τα labels σε one-hot encoded διανύσματα, όπου για την περίπτωση μας είναι διανύσματα 10 στοιχείων (# of classes = 10) που είναι γεμάτα με μηδενικά εκτός από τον δείκτη της κλάσης στην οποία ανήκει το δείγμα (στόχος). Επομένως, είναι κατάλληλη για προβλήματα ταξινόμησης, ειδικά με την παράλληλη χρήση της SoftMax η οποία παράγει πιθανότητες για κάθε κατηγορία, καθώς επιτρέπει τη σωστή σύγκριση της εξόδου του μοντέλου με τις πραγματικές κατηγορίες (κλάσεις). Έτσι, επιστρέφει έναν πίνακαν με διαστάσεις *len(labels)* x *num_classes*, όπου κάθε γραμμή είναι το one-hot encoded διάνυσμα της αντίστοιχης κατηγορίας.

```
# Το στοιχείο του πίνακα που αντιστοιχεί στο label είναι 1, ΕΝΩ όλα τα άλλα 0
def one_hot_encode(labels, num_classes):
    return np.eye(num_classes)[labels] # Δημιουργία ταυτοτικού πίνακα
```

2.10 – Παραδείγματα ορθής και εσφαλμένης κατηγοριοποίησης

Η συνάρτηση "*display_examples(x, y, y_pred, num_examples)*" εμφανίζει χαρακτηριστικά παραδείγματα σωστής και λανθασμένης κατηγοριοποίησης. Τα παραδείγματα εμφανίζονται ως one-hot encoded διανύσματα με χρήση της συνάρτησης "*one_hot_encode(labels, num_classes)*" της παραγράφου *2.9*, για την πρόβλεψη και την ετικέτα. Η μεταβλητή "*correct_indices*" υπολογίζει τους δείκτες των παραδειγμάτων όπου η πρόβλεψη του μοντέλου είναι σωστή (δηλ. η ετικέτα "*y_true*" είναι ίση με την πρόβλεψη "*y_pred*"). Αντίστοιχα υπάρχει η "*in*correct_*indices*" για τις εσφαλμένες προβλέψεις.

```python
def display_examples(y_true, y_pred, num_examples):
    print("\nΣωστές Κατηγοριοποιήσεις:")
    correct_indices = np.where(y_true == y_pred)[0]
    for i in correct_indices[:num_examples]: # έως num_examples
        print(f"Παράδειγμα {i}:")
        print(f"Πραγματική Ετικέτα (One-hot): {one_hot_encode([y_true[i]], 10)[0]}")
        print(f"Πρόβλεψη (One-hot): {one_hot_encode([y_pred[i]], 10)[0]}")
        print("-" * 30)

    print("\nΛανθασμένες Κατηγοριοποιήσεις:")
    incorrect_indices = np.where(y_true != y_pred)[0]
    for i in incorrect_indices[:num_examples]:
        print(f"Παράδειγμα {i}:")
        print(f"Πραγματική Ετικέτα (One-hot): {one_hot_encode([y_true[i]], 10)[0]}")
        print(f"Πρόβλεψη (One-hot): {one_hot_encode([y_pred[i]], 10)[0]}")
        print("-" * 30)
```

```
Σωστές Κατηγοριοποιήσεις:
Παράδειγμα 0:
Πραγματική Ετικέτα (One-hot): [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
Πρόβλεψη (One-hot): [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
------------------------------
Παράδειγμα 3:
Πραγματική Ετικέτα (One-hot): [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
Πρόβλεψη (One-hot): [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
------------------------------
Παράδειγμα 5:
Πραγματική Ετικέτα (One-hot): [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
Πρόβλεψη (One-hot): [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
------------------------------

Λανθασμένες Κατηγοριοποιήσεις:
Παράδειγμα 1:
Πραγματική Ετικέτα (One-hot): [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
Πρόβλεψη (One-hot): [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
------------------------------
Παράδειγμα 2:
Πραγματική Ετικέτα (One-hot): [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
Πρόβλεψη (One-hot): [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
------------------------------
Παράδειγμα 4:
Πραγματική Ετικέτα (One-hot): [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
Πρόβλεψη (One-hot): [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
------------------------------
```

2.11 – Κύριο πρόγραμμα

Η συνάρτηση *"main()"* αποτελεί το κύριο πρόγραμμα για την εκπαίδευση και αξιολόγηση ενός πλήρους συνδεδεμένου **MLP** πάνω στο dataset **CIFAR-10**. Συνοπτικά:

- Φορτώνονται τα δεδομένα εκπαίδευσης (x_train, y_train) και ελέγχου (x_test, y_test) μέσα από τη συνάρτηση *"load_cifar10()"*
- Οι ετικέτες *"y_train"* και *"y_test"* μετατρέπονται σε μορφή one-hot encoding, ώστε να είναι συμβατές με την έξοδο του νευρωνικού δικτύου.
- Ορίζονται τιμές για τις υπερπαραμέτρους του δικτύου
- Δημιουργείται ένα αντικείμενο του μοντέλου MLP με τις δομές επιπέδων που ορίζονται στις *"layer_sizes"*, το μοντέλο εκπαιδεύεται χρησιμοποιώντας την μέθοδο *"train()"* και καταγράφεται ο χρόνος εκπαίδευσης
- Δημιουργείται ένα γράφημα που δείχνει την εξέλιξη της απώλειας (loss) κατά τη διάρκεια της εκπαίδευσης για το training και το test set.
- Χρησιμοποιείται η μέθοδος *"predict()"* για να προβλεφθούν οι ετικέτες των δεδομένων δοκιμών (**x_test**) και η *"display_examples()"* εμφανίζει τρία παραδείγματα σωστών και λανθασμένων κατηγοριοποιήσεων.

```
def main():
    # Φόρτωση όλων των δεδομένων
    x_train, y_train, x_test, y_test = load_cifar10()
    print(f"Φόρτωση {len(x_train)} δεδομένων εκπαίδευσης και {len(x_test)} δεδομένων ελέγχου")

    # One-hot encoding
    y_train_onehot = one_hot_encode(y_train, 10)
    y_test_onehot = one_hot_encode(y_test, 10)

    # Ορισμός υπερπαραμέτρων δικτύου
    input_size = x_train.shape[1] # 3072 στοιχεία
    hidden_sizes = [256, 128]  # δύο κρυφά επίπεδα με 256 και 128 νευρώνες
    output_size = 10  # αριθμός κατηγοριών
    initial_lr = 0.01 # αρχικό leraning rate
    epochs = 250
    batch_size = 128
    wait = 1 # αριθμός εποχών χωρίς βελτίωση πριν τη μείωση του lr
    lr_decay = 0.5 # ο συντέλεστης μείωσης του lr

    # Δημιουργία και εκπαίδευση του MLP
    layer_sizes = [input_size] + hidden_sizes + [output_size]
    model = MLP(layer_sizes, initial_lr, batch_size)
    start_time = time.time()
    model.train(x_train, y_train_onehot, x_test, y_test, epochs, batch_size, initial_lr, wait, lr_decay)
    training_time = time.time() - start_time
    print(f"Χρόνος εκπαίδευσης: {training_time:.2f} δευτερόλεπτα")

    # Plot των Training και Test Loss
    plt.plot(range(epochs), model.train_loss_history, label="Training Loss", color="blue")
    plt.plot(range(epochs), model.test_loss_history, label="Test Loss", color="red")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.title("Loss per Epoch")
    plt.legend()
    plt.grid(True)
    plt.show()

    y_test_pred = model.predict(x_test)
    # Εμφάνιση παραδειγμάτων ορθής και εσφαλμένης κατηγοριοποίησης
    display_examples(y_test, y_test_pred, num_examples=3)

if __name__ == "__main__":
    main()
```

3. Πειραματισμοί και Αποτελέσματα

Στις ακόλουθες παραγράφους παρουσιάζονται τα αποτελέσματα και διάφοροι πειραματισμοί που πραγματοποιήθηκαν κατά τη διάρκεια υλοποίησης (from scratch) του κώδικα σε Python για την κατασκευή, εκπαίδευση και αξιολόγηση του νευρωνικού δικτύου MLP**.** Σε αυτό το σημείο αξίζει να αναφερθεί ότι τα **test loss και test accuracy** που αναφέρονται, στην ουσία είναι **validation loss και validation accuracy** (δεν υπάρχει test error αλλά προσομοιάζεται με το validation error). Για το **test** κανονικά τα **δεδομένα** πρέπει να είναι τελείως **άγνωστα**. Τέλος, γενικά το **training accuracy** αντικατοπτρίζει την ικανότητα του μοντέλου να προσαρμόζεται στα δεδομένα, ενώ το **test accuracy** την ικανότητα του για γενίκευση.

3.1- Αρχικοποίηση Βαρών και Συναρτήσεις Ενεργοποίησης

Τα αρχικά χαρακτηριστικά του δικτύου μου ήταν:

- **Κανονικοποίηση** στις τιμές των εικόνων
- Χρήση της συνάρτησης ενεργοποίησης **Sigmoid** για τα κρυφά επίπεδα και **SoftMax** για το επίπεδο εξόδου
- Συνάρτηση απώλειας σφάλματος η **Cross-Entropy**
- **Απουσία batch training** και **δυναμικής προσαρμογής του learning rate (σταθερό learning rate** = 0.1**)**
- **Αριθμός νευρώνων** = 100 (μοναδικό κρυφό επίπεδο)

Αντιμετωπίστηκε πρόβλημα με την αρχικοποίηση των βαρών της συνάρτησης ενεργοποίησης. Δοκίμασα να χρησιμοποιήσω την συνάρτηση "*np.random.randn()*" η οποία δίνει τιμές στο εύρος [0,1] το οποίο έχει μεγάλη διασπορά και τα βάρη παρουσιάζονταν πολύ μεγάλα σε σχέση με τις εισόδους, κάτι το οποίο συνεπάγεται προβλήματα στη σύγκλιση λόγω κορεσμού των συναρτήσεων ενεργοποίησης με φαινόμενα vanishing ή exploding gradients καθιστώντας δύσκολη την εκπαίδευση του μοντέλου. Έπειτα δοκίμασα να πολλαπλασιάσω την "*np.random.randn()*" με διάφορες τιμές (scale_values) και παρατηρήθηκαν καλύτερα αποτελέσματα στην απόδοση του δικτύου. Για τις δοκιμές έτρεξα **ολόκληρο** το **dataset** για **50 εποχές**. Παρακάτω παρουσιάζονται οι τιμές των **train losses** ανά 10 εποχές, ο **χρόνος εκπαίδευσης**, τα **training** και **testing accuracies** για διαφορετικές τιμές των scale values:

scale_value	-	0.001	0.01	0.1
Epoch 1 (loss)	2.7635	2.3118	2.3064	2.0094
Epoch 10 (loss)	2.3554	2.0622	1.7922	1.9384
Epoch 20 (loss)	2.2094	2.0382	1.7691	1.6711
Epoch 30 (loss)	2.0654	1.8579	1.6507	1.4282
Epoch 40 (loss)	1.9610	1.7025	1.5586	1.2959
Epoch 50 (loss)	1.8746	1.7762	1.4682	1.1906
Χρόνος εκπαίδευσης	1793.81 sec	1842.45 sec	1799.47 sec	1885.68 sec
Training accuracy	34.59%	34.95%	47.88%	57.3%
Testing accuracy	30.8%	33.93%	44.24%	46.46%

Από τα παραπάνω, αποφάσισα να κρατήσω για το **3.2** το **scale_value = 0.01 ,** καθώς για τις εποχές που άφησα το δίκτυο να εκπαιδευτεί αν και το train loss του είχε αρκετή διαφορά με αυτό του scale_value = 0.1, φάνηκε να υπάρχει πιθανότητα να οδηγηθεί δυσκολότερα από αυτό σε **overfitting** ,αν και κανονικά για να το παρατηρήσω πρέπει να δω αν ενώ το train loss συνεχίζει να μειώνεται, το test loss να αρχίζει να αυξάνεται (δεν ήλεγχα το test loss όπως θα έπρεπε και αποφάσισα να το πάω με γνώμονα την διαφορά training και test accuracy).

<u>Επιστροφή στο πρόβλημα του 3.1, ύστερα από τις δοκιμές του 3.2</u>

Τα χαρακτηριστικά του δικτύου μου είναι:

- **Κανονικοποίηση** στις τιμές των εικόνων
- Χρήση της συνάρτησης ενεργοποίησης **Sigmoid** για τα κρυφά επίπεδα και **SoftMax** για το επίπεδο εξόδου (+ **δοκιμή χρήσης ReLU**)
- Συνάρτηση απώλειας σφάλματος η **Cross-Entropy**
- **Απουσία batch training** και **δυναμικής προσαρμογής του learning rate (σταθερό learning rate** = 0.1**)**
- **Scale_value =0.01,** για την αρχικοποίηση των βαρών
- **Δομή κρυφών επιπέδων [128, 64]**

Δοκίμασα να χρησιμοποιήσω το **Xavier Initialization** του οποίου η χρήση συνηθίζεται για δίκτυα με συνάρτηση ενεργοποίησης **Sigmoid** και σύμφωνα με το οποίο τα βάρη αρχικοποιούνται ως: $w \sim N(0, \frac{1}{n})$ όπου n →ο αριθμός των νευρώνων του προηγούμενου επιπέδου. Το προαναφερθέν υλοποιήθηκε με τον κώδικα:

```
for i in range(len(layer_sizes) - 1):
    self.weights.append(np.random.randn(layer_sizes[i], layer_sizes[i + 1]) * 0.01)
    limit = np.sqrt(6 / (layer_sizes[i] + layer_sizes[i + 1]))
```

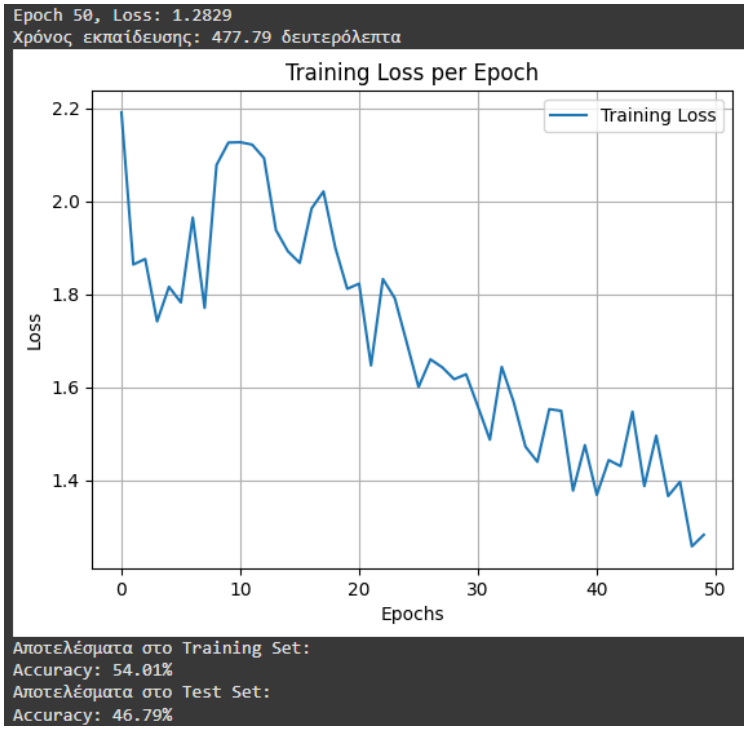
Επίσης, δοκίμασα να χρησιμοποιήσω την αρχικοποίηση των βαρών από **ομοιόμορφη κατανομή [-0.5, 0.5]** που υλοποιήθηκε με τον παρακάτω κώδικα:

```
for i in range(len(layer_sizes) - 1):
    self.weights.append(np.random.uniform(low=-0.5, high=0.5, size=(layer_sizes[i], layer_sizes[i + 1])))
```

Τα αποτελέσματα για τις **50 εποχές** που έτρεξε ο κώδικας φαίνονται στον παρακάτω πίνακα:

	Ομοιόμορφη κατανομή	**Xavier Initialization**
Epoch 1 (loss)	1.9042	1.9257
Epoch 10 (loss)	1.5981	1.8415
Epoch 20 (loss)	1.4667	1.5748
Epoch 30 (loss)	1.3736	1.4674
Epoch 40 (loss)	1.2985	1.3456
Epoch 50 (loss)	1.2463	1.3038
Χρόνος εκπαίδευσης	496.16 sec	487.9 sec
Training accuracy	55.65%	53.07%
Testing accuracy	48.2%	46.16%

Επιπλέον δοκίμασα να χρησιμοποιήσω την **συνάρτηση ενεργοποίησης ReLU** (Rectified Linear Unit) με αρχικοποίηση βαρών μέσω **He Initialization**, λεπτομέρειες για τα οποία δίνονται στις παραγράφους ***2.4 και 2.3*** αντίστοιχα**.** Τα αποτελέσματα παρουσιάζονται στην παρακάτω εικόνα:



Παρατηρώ πολλές αυξομειώσεις στον train loss που συνεπάγεται με αποτέλεσμα η αν και συγκλίνει μακροπρόθεσμα αυτό να μην γίνεται καθόλου ομαλά.

Έπειτα από τα παραπάνω αποτελέσματα αποφάσισα να χρησιμοποιήσω την **ομοιόμορφη κατανομή,** όπως παρουσιάστηκε προηγουμένως , με **Sigmoid** σε συνδυασμό με το αποτέλεσμα από την ***3.2***, για την ενδεχομένως πιο αποδοτική δομή δικτύου [3072, 256, 128, 10] και κατέληξα στα εξής:

	Ομοιόμορφη κατανομή και [256,128]
Epoch 1 (loss)	1.9249
Epoch 10 (loss)	1.7236
Epoch 20 (loss)	1.5625
Epoch 30 (loss)	1.4104
Epoch 40 (loss)	1.2700
Epoch 50 (loss)	1.1777
Χρόνος εκπαίδευσης	912.26 sec
Training accuracy	57.39%
Testing accuracy	48.96%

3.2- Αριθμός νευρώνων και πλήθος επιπέδων

Τα χαρακτηριστικά του δικτύου μου είναι:

- **Κανονικοποίηση** στις τιμές των εικόνων
- Χρήση της συνάρτησης ενεργοποίησης **Sigmoid** για τα κρυφά επίπεδα και **SoftMax** για το επίπεδο εξόδου
- Συνάρτηση απώλειας σφάλματος η **Cross-Entropy**
- **Απουσία batch training** και **δυναμικής προσαρμογής του learning rate (σταθερό learning rate** = 0.1**)**
- **Scale_value =0.01,** για την αρχικοποίηση των βαρών

Δοκίμασα να εκπαιδεύσω το δίκτυο με διαφορετικό αριθμό νευρώνων και πλήθος κρυφών επιπέδων. Όσον αφορά το επίπεδο εισόδου και το επίπεδο εξόδου έχουν fixed τιμές , 3072 (διάσταση εικόνας) και 10 (πλήθος κατηγοριών/κλάσεων). Για τις δοκιμές έτρεξα **ολόκληρο** το **dataset** για **50 εποχές**. Παρακάτω παρουσιάζονται οι τιμές των **train losses** ανά 10 εποχές,

ο **χρόνος εκπαίδευσης**, τα **training** και **testing accuracies** για τις αντίστοιχες δοκιμές (σε αγκύλες συμβολίζονται οι αριθμοί των νευρώνων των κρυφών επιπέδων και το πλήθος των κρυφών επιπέδων):

	[256]	**[512]**	**[128,64]**
Epoch 1 (loss)	1.9508	1.9795	2.3095
Epoch 10 (loss)	1.7462	1.7306	1.8718
Epoch 20 (loss)	1.5099	1.5530	1.6964
Epoch 30 (loss)	1.3683	1.4256	1.6861
Epoch 40 (loss)	1.2817	1.3246	1.5663
Epoch 50 (loss)	1.2110	1.2476	1.5049
Χρόνος εκπαίδευσης	913.91 sec	1700.91 sec	513.78 sec
Training accuracy	47.64%	55.32%	46.71%
Testing accuracy	43.63%	46.84%	43.36%

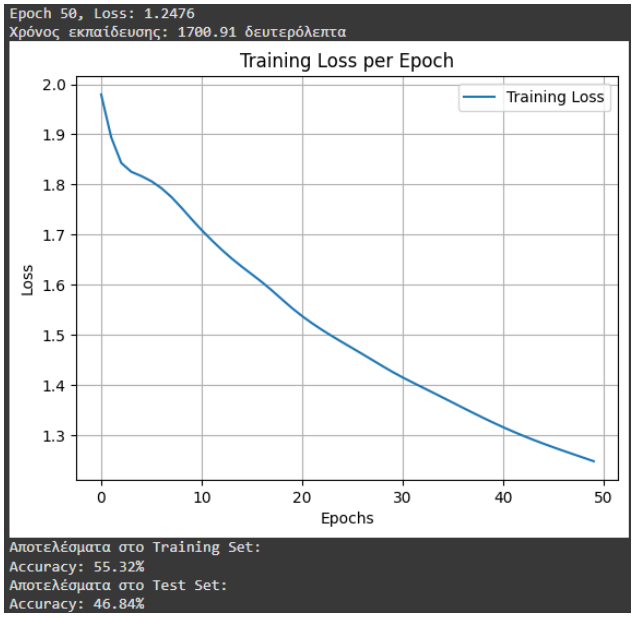
	[256, 128]	**[256, 128, 64]**	**[512, 256, 128]**
Epoch 1 (loss)	2.3113	1.8956	1.9855
Epoch 10 (loss)	1.8012	1.6315	1.8646
Epoch 20 (loss)	1.7173	1.4668	1.6177
Epoch 30 (loss)	1.6559	1.3546	1.4726
Epoch 40 (loss)	1.5149	1.2996	1.3514
Epoch 50 (loss)	1.3613	1.2708	1.2111
Χρόνος εκπαίδευσης	941.27 sec	982.54 sec	1927.11 sec
Training accuracy	51.24%	54.35%	57.67%
Testing accuracy	43.63%	46.84%	43.6%

Όπως αναμενόταν, πιο πολύπλοκα δίκτυα (περισσότερα κρυφά επίπεδα) ή με μεγαλύτερο αριθμό νευρώνων που δίνουν τη δυνατότητα να μαθαίνει πιο σύνθετα πρότυπα, οδηγούν σε υψηλότερη ακρίβεια στο training set (καλύτερη προσαρμογή) και απαιτούν περισσότερο χρόνο εκπαίδευσης Επιπλέον, στη δομή **[512, 256, 128]** παρατηρήθηκε μεταξύ των εποχώ 40-50 μια απότομη αύξηση για το train loss στο 1.9124

Αποφάσισα για τη δομή του δικτύου να κρατήσω το [3072, 256, 128, 10] ως το πιο αποδοτικό με μικρότερο ενδεχόμενο για overfitting με την ίδια λογική με προηγουμένως, καθώς επιτυγχάνει την καλύτερη ισορροπία μεταξύ ακρίβειας και γενίκευσης. Ωστόσο, πρώτα έκανα κάποιες δοκιμές, για διαφορετικούς τρόπους αρχικοποίησης των βαρών (επιστρέφοντας έτσι

στο ζήτημα τις παραγράφου ***3.1)*** στο δίκτυο με δομή κρυφού επιπέδου [128, 64], διότι είχε πολύ μικρότερο χρόνο εκπαίδευσης συγκριτικά με τα υπόλοιπα.

Παράδειγμα απεικόνισης μίας εκ των παραπάνω δοκιμών:



3.3- Batch training

Τα χαρακτηριστικά του δικτύου μου είναι:

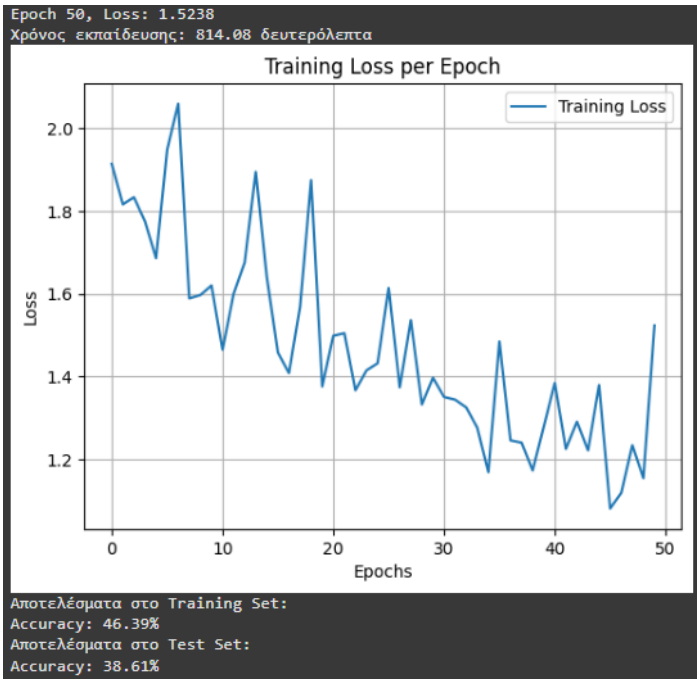
- **Κανονικοποίηση** στις τιμές των εικόνων
- Χρήση της συνάρτησης ενεργοποίησης **Sigmoid** για τα κρυφά επίπεδα και **SoftMax** για το επίπεδο εξόδου
- Συνάρτηση απώλειας σφάλματος η **Cross-Entropy**
- **Batch training** με **batch size = 64**
- Απουσία **δυναμικής προσαρμογής του learning rate (σταθερό learning rate** με τιμές 0.1 και 0.01**)**
- Χρήση **ομοιόμορφης κατανομής [-0.5, 0.5]** για την αρχικοποίηση των βαρών
- **Δομή κρυφών επιπέδων [256, 128]**

Αποφάσισα να χρησιμοποιήσω **batch training** που παρουσιάζεται στην παρ. ***2.8*** για καλύτερη διαχείριση των υπολογιστικών πόρων και της μνήμης αφού δεν χρησιμοποιείται ολόκληρο το dataset ταυτόχρονα αλλά χωρίζεται σε batches συγκεκριμένου μεγέθους. Το δίκτυο εκπαιδεύτηκε για **50 εποχές** και η απόδοση ήταν πολύ χαμηλή με πολλαπλές αυξομειώσεις στο train loss λόγω του υψηλού ***learning rate = 0.1***, για αυτό δοκίμασα και ***learning rate = 0.01*** :

Learning rate	**0.1**	**0.01**
Epoch 1 (loss)	1.9143	2.0986
Epoch 10 (loss)	1.6205	1.7892
Epoch 20 (loss)	1.3764	1.7007
Epoch 30 (loss)	1.3975	1.6449
Epoch 40 (loss)	1.2790	1.5965
Epoch 50 (loss)	1.5238	1.5627
Χρόνος εκπαίδευσης	814.08 sec	877.92 sec
Training accuracy	46.39%	44.66%
Testing accuracy	38.61%	43.41%

Ορισμένα παραδείγματα αυξομειώσεων για ***learning rate = 0.1*** είναι:

i. 2.05 μεταξύ των εποχών 1-10
ii. 1.89 μεταξύ των εποχών 10-20
iii. 1.61 μεταξύ των εποχών 20-30
iv. 1.48 μεταξύ των εποχών 30-40
v. 1.08 μεταξύ των εποχών 40-50



3.4- Δυναμική προσαρμογή learning rate και αλλαγή batch size

Τα χαρακτηριστικά του δικτύου μου είναι:

- **Κανονικοποίηση** στις τιμές των εικόνων
- Χρήση της συνάρτησης ενεργοποίησης **Sigmoid** ή **ReLU** για τα κρυφά επίπεδα και **SoftMax** για το επίπεδο εξόδου
- Συνάρτηση απώλειας σφάλματος η **Cross-Entropy**
- **Batch training** με διαφορετικές τιμές **batch size**
- **Δυναμική προσαρμογή του learning rate** με **αρχικό learning rate = 0.1** και διαφορετικές τιμές για **lr_decay** και **wait**
- Χρήση **ομοιόμορφης κατανομής [-0.5, 0.5]** για την αρχικοποίηση των βαρών και **He Initialization**
- **Δομή κρυφών επιπέδων [256, 128]**

Παρακάτω χρησιμοποιήθηκε **δυναμική προσαρμογή του learning rate** όπως παρουσιάζεται στην παράγραφο ***2.8.*** Αυτό έγινε με σκοπό την βελτίωση της απόδοσης της εκπαίδευσης και την καλύτερη γενίκευση του μοντέλου, προσπαθώντας για αποφυγή αστάθειας του train loss, αργής σύγκλισης λόγω χαμηλού learning rate και υπερκπαίδευσης λόγω υψηλού learning rate. Το δίκτυο εκπαιδεύτηκε για **50 εποχές** με **lr_decay = 0.5 και batch size =64:**

wait	**5**	**2**
Epoch 1 (loss)	1.8656	2.0475
Epoch 10 (loss)	1.6187	1.4500
Epoch 20 (loss)	1.3415	1.3729
Epoch 30 (loss)	1.3066	1.3431
Epoch 40 (loss)	1.2802	1.3198
Epoch 50 (loss)	1.2414	1.3140
Χρόνος εκπαίδευσης	809.64 sec	805.02 sec
Training accuracy	56.27%	53.69%
Testing accuracy	47.33%	47.84%

i. Με **wait = 5** το learning rate άλλαξε μόνο 2 φορές με τελική τιμή 0.05 από την εποχή 19 και το train loss μειώνεται πιο σταθερά. Επομένως, επιτρέπει περισσότερες εποχές πριν μειώσει το learning rate βοηθώντας στη μείωση του train loss, αλλά μπορεί να ενισχύει το overfitting, καθώς το test accuracy δεν βελτιώνεται.

ii. Με **wait = 2** το learning rate άλλαξε 5 φορές με τελική τιμή 0.003125 από την εποχή 42. Έχει χαμηλότερο training accuracy, οπότε και μικρότερη ικανότητα

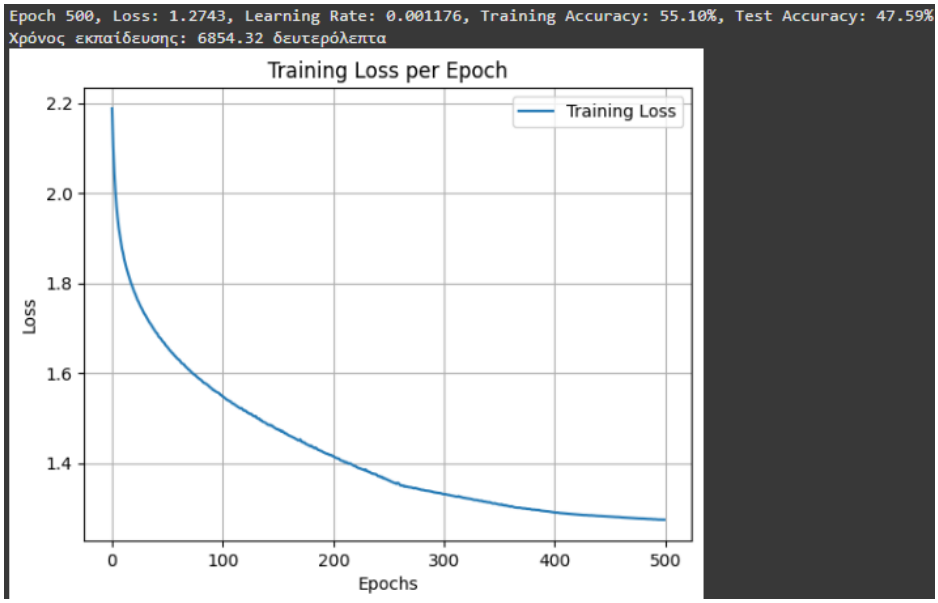
εκμάθησης πιθανώς λόγω της πιο γρήγορης μείωσης του learning rate και αποδίδει παρόμοια με το wait = 5 στο test set, με ελαφρώς καλύτερη απόδοση υποδηλώνοντας ότι ενδεχομένως να αποφύγει καλύτερα το onerfitting (καλύτερη γενίκευση).

Στη συνέχεια δοκιμάστηκε η εκπαίδευση του δικτύου για **50 εποχές** με **lr_decay = 0.5 και batch size = 128** και παρατηρήθηκαν διαφορές τόσο στην **δυναμική αλλαγή** του **learning rate** όσο και ως προς το **χρόνο εκπαίδευσης,** συγκριτικά με **batch size = 64.** Πλέον η ανανέωση των βαρών βασίζεται σε μεγαλύτερο υποσύνολο δεδομένων και αυτό μειώνει τις τυχαίες διακυμάνσεις (όσο μεγαλύτερο είναι το batch size τόσο καλύτερα το gradient αντιπροσωπεύει την πραγματική κατεύθυνση μείωσης του loss) και κάνει το loss να μεταβάλλεται πιο αργά και ομαλά. Επίσης, ο χρόνος εκπαίδευσης αυξάνεται γιατί κάθε βήμα εκπαίδευσης απαιτεί περισσότερους υπολογισμούς:

wait	**5**	**2**
Epoch 1 (loss)	1.8949	1.9347
Epoch 10 (loss)	1.6105	1.6347
Epoch 20 (loss)	1.5267	1.4827
Epoch 30 (loss)	1.3911	1.4017
Epoch 40 (loss)	1.3291	1.3492
Epoch 50 (loss)	1.2433	1.3305
Χρόνος εκπαίδευσης	935.69 sec	939.5 sec
Training accuracy	55.84%	52.83%
Testing accuracy	47.97%	47.49%

i. Με **wait = 5** το learning rate **δεν** άλλαξε καθόλου με τελική τιμή 0.1
ii. Με **wait = 2** το learning rate άλλαξε μόνο μία φορά με τελική τιμή 0.05 από την εποχή 24

Παρακάτω απεικονίζεται για **wait = 2,** αρχικό **learning rate = 0.01, lr_decay = 0.7** και **500 εποχές:**



Έπειτα δοκιμάστηκε η εκπαίδευση του δικτύου για **50 εποχές** με **lr_decay = 0.5, wait = 2** (βοηθά το δίκτυο να σταθεροποιηθεί γρηγορότερα στις βέλτιστες περιοχές του χώρου παραμέτρων) και χρήση της **συνάρτησης ενεργοποίησης ReLU** (Rectified Linear Unit) με αρχικοποίηση βαρών μέσω **He Initialization**, λεπτομέρειες για τα οποία δίνονται στις παραγράφους ***2.4 και 2.3*** αντίστοιχα. Ο συνδυασμός των παραπάνω και για λόγους που παρουσιάζονται στις προαναφερθείσες παραγράφους, οδήγησε στη σημαντική βελτίωση της επίδοσης του δικτύου στο test set:

batch size	**64**	**128**
Epoch 1 (loss)	1.8933	1.8852
Epoch 10 (loss)	1.3990	1.4216
Epoch 20 (loss)	1.2212	1.2004
Epoch 30 (loss)	1.1341	1.1186
Epoch 40 (loss)	1.1121	1.0967
Epoch 50 (loss)	1.1028	1.0608
Χρόνος εκπαίδευσης	1136.71 sec	1205.37 sec
Training accuracy	61.17%	62.85%
Testing accuracy	52.33%	53.93%

i. Με **batch size = 64** το learning rate άλλαξε 6 φορές με τελική τιμή 0.001563 από την εποχή 39

ii. Με **batch size = 128** το learning rate άλλαξε 4 φορές με τελική τιμή 0.00625 από την εποχή 30

ΣΗΜΕΙΩΣΗ : Με **αρχικό learning rate = 0.01** μειώθηκαν αρκετά τα ποσοστά στα accuracies με training ~ 56% και testing ~ 51%, ενώ αυξήθηκε ελαφρώς το train loss

Με **wait =5** για τα ποσοστά στα accuracies έχουμε training ~ 67% και testing ~ 52%, ενώ μειώθηκε περεταίρω train loss ~ 0.92 .

Τα αποτελέσματα είναι λογικά σύμφωνα με όσα έχουν αναφερθεί και σε προηγούμενες παραγράφους.

Ακόμη, δοκιμάστηκε η εκπαίδευση του δικτύου για **100 εποχές** με αρχικό **learning rate = 0.01 batch size = 128** και χρήση της **συνάρτησης ενεργοποίησης ReLU** (Rectified Linear Unit) με αρχικοποίηση βαρών μέσω **He Initialization:**

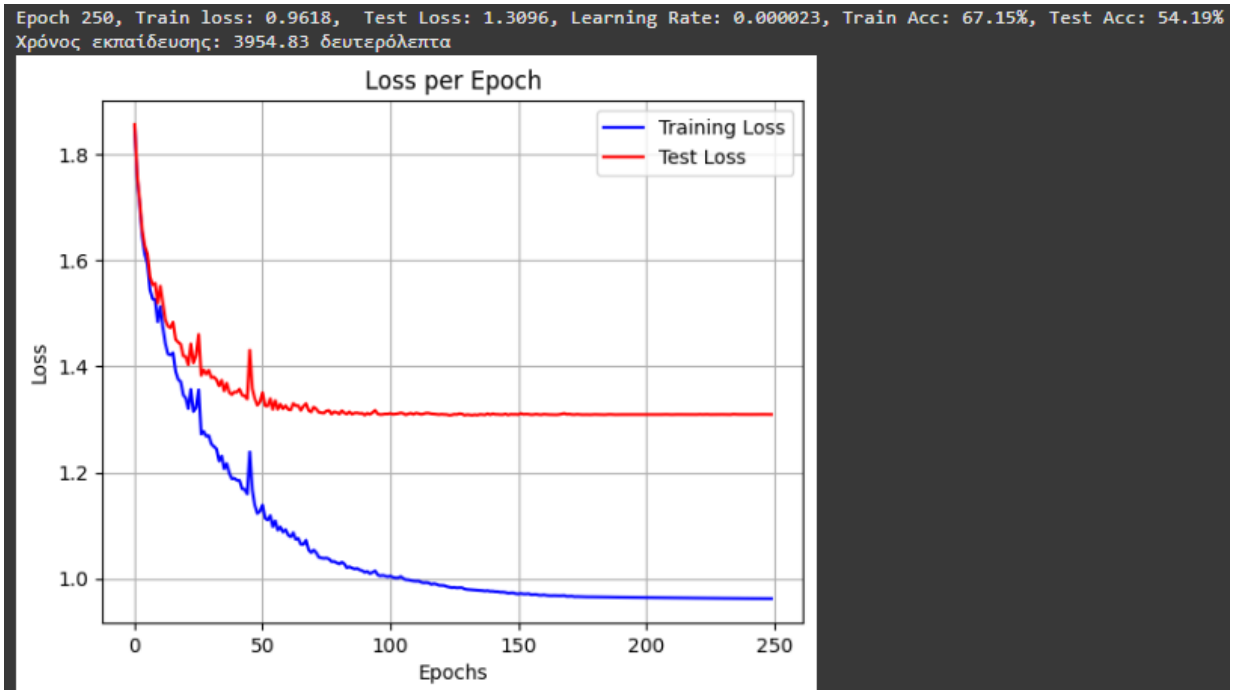
lr_decay	**0.5**	**0.7**
wait	**5**	**2**
Epoch 1 (loss)	1.8465	1.8382
Epoch 50 (loss)	1.1272	1.1462
Epoch 100 (loss)	0.8112	1.0548
Χρόνος εκπαίδευσης	1307.7 sec	1311.37 sec
Training accuracy	72.37%	63.71%
Testing accuracy	54.53%	53.36%

i. Με **lr_decay = 0.5** και **wait = 5** το learning rate άλλαξε μόνο μία φορά με τελική τιμή 0.05 . Προσφέρει καλύτερη απόδοση στην εκπαίδευση (training accuracy) και ταχύτερη σύγκλιση (μικρότερο loss) χωρίς να επηρεάζει αρνητικά τη γενίκευση (testing accuracy) αλλά φαίνεται να οδηγείται ευκολότερα σε onerfitting.
ii. Με **lr_decay = 0.7** και **wait = 2** το learning rate άλλαξε 8 φορές με τελική τιμή 0.000576. Περιορίζει την απόδοση εκπαίδευσης, αλλά διατηρεί αντίστοιχη απόδοση στο testing και φαίνεται να οδηγείται δυσκολότερα σε onerfitting.
Για **150 εποχές** , ο χρόνος εκπαίδευσης ήταν ~ 2263 sec , με training accuracy ~ 68.1%, testing accuracy ~ 54.4% και train loss ~ 0.9353

3.5- Επιπρόσθετες μετρήσεις για μεγαλύτερο αριθμό εποχών και τυχαία αναδιάταξη των batches

Παρακάτω απεικονίζονται τα αποτελέσματα για χαρακτηριστικά δικτύου:

- **Κανονικοποίηση** στις τιμές των εικόνων
- Χρήση της συνάρτησης ενεργοποίησης **ReLU** για τα κρυφά επίπεδα και **SoftMax** για το επίπεδο εξόδου
- Συνάρτηση απώλειας σφάλματος η **Cross-Entropy**
- **Batch training** με **batch size = 128**
- **Δυναμική προσαρμογή του learning rate** με **αρχικό learning rate = 0.01**, **lr_decay =0.7** και **wait = 2**
- Χρήση **He Initialization** για την αρχικοποίηση των βαρών
- **Δομή κρυφών επιπέδων [256, 128]**
- **150 εποχές**



i. Υψηλότερη τιμή για το testing accuracy στην εποχή 155 με ποσοστό 54.68%

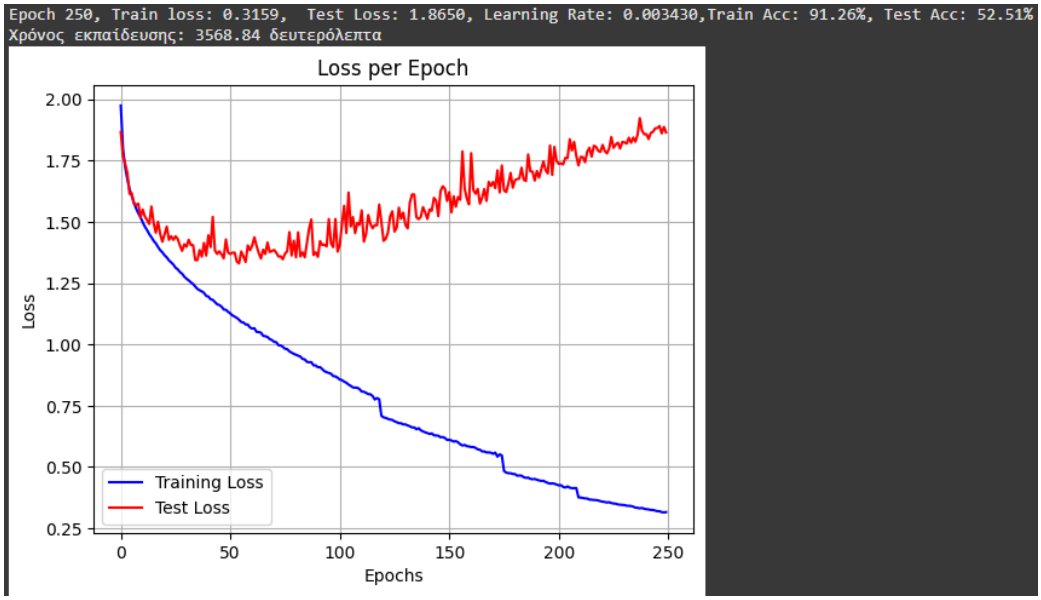
Τέλος για το **batch training** εφάρμοσα **τυχαία αναδιάταξη** των δεδομένων (“shuffle the training data”) για τους λόγους που παρουσιάζεται στην παρ. ***2.8*** . Όλα οι παρακάτω εικόνες είναι για χαρακτηριστικά δικτύου:

- **Κανονικοποίηση** στις τιμές των εικόνων
- Χρήση της συνάρτησης ενεργοποίησης **ReLU** για τα κρυφά επίπεδα και **SoftMax** για το επίπεδο εξόδου
- Συνάρτηση απώλειας σφάλματος η **Cross-Entropy**
- Χρήση **He Initialization** για την αρχικοποίηση των βαρών
- **Δομή κρυφών επιπέδων [256, 128]**
- **250 εποχές**

Οπότε θα αναφέρονται μόνο τα χαρακτηριστικά που διαφοροποιούν τα δίκτυα.

Παρακάτω απεικονίζονται τα αποτελέσματα για χαρακτηριστικά δικτύου:

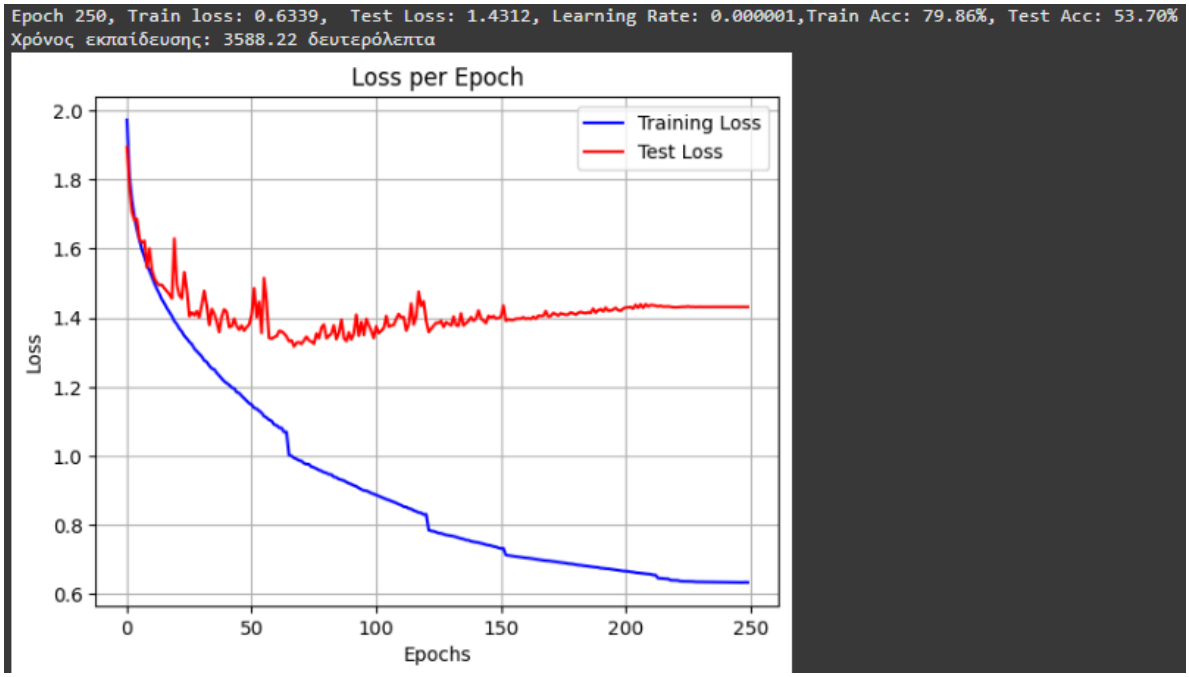
- **Batch training** με **τυχαία αναδιάταξη** των **batches** και **batch size = 128**
- **Δυναμική προσαρμογή του learning rate** με **αρχικό learning rate = 0.01**, **lr_decay =0.7** και **wait = 2**



i. Υψηλότερη τιμή για το testing accuracy στην εποχή 111 με ποσοστό 54.84%. Το overfitting είναι εμφανέστατο και έντονο

Παρακάτω απεικονίζονται τα αποτελέσματα για χαρακτηριστικά δικτύου:

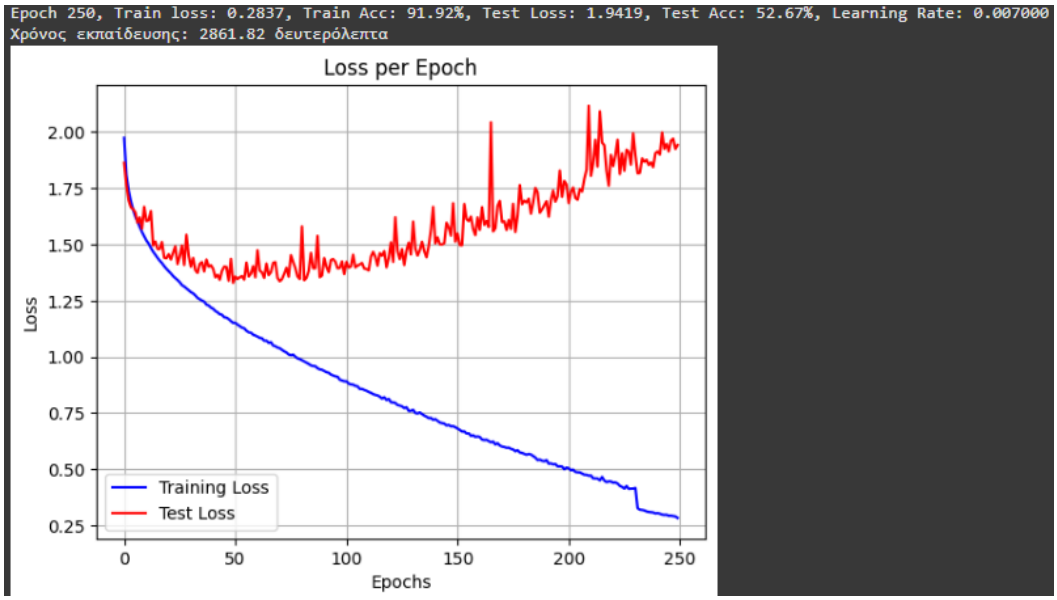
- **Batch training** με **τυχαία αναδιάταξη** των **batches** και **batch size = 128**
- **Δυναμική προσαρμογή του learning rate** με **αρχικό learning rate = 0.01**, **lr_decay =0.5** και **wait = 1**



i. Υψηλότερη τιμή για το testing accuracy στην εποχή 89 με ποσοστό 54.23%. Παρατηρείται overfitting.

Παρακάτω απεικονίζονται τα αποτελέσματα για χαρακτηριστικά δικτύου:

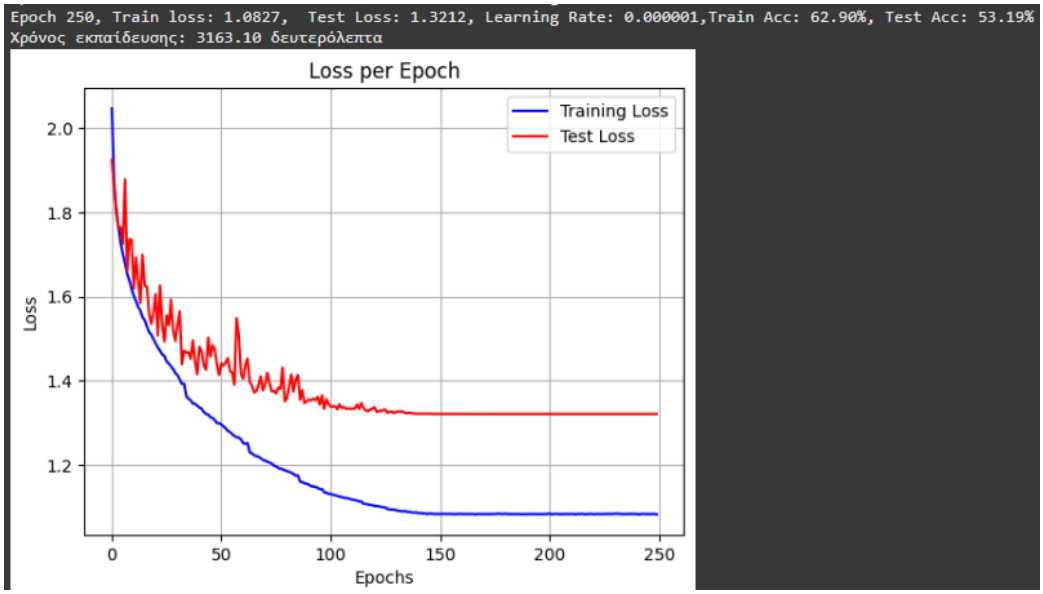
- **Batch training** με **τυχαία αναδιάταξη** των **batches** και **batch size = 128**
- **Δυναμική προσαρμογή του learning rate** με **αρχικό learning rate = 0.01**, **lr_decay =0.7** και **wait = 3**



i. Υψηλότερη τιμή για το testing accuracy στην εποχή 110 με ποσοστό 53.82%. Το overfitting είναι εμφανέστατο και έντονο.

Παρακάτω απεικονίζονται τα αποτελέσματα για χαρακτηριστικά δικτύου:

- **Batch training** με **τυχαία αναδιάταξη** των **batches** και **batch size = 256**
- **Δυναμική προσαρμογή του learning rate** με **αρχικό learning rate = 0.01**, **lr_decay =0.7** και **wait = 1**



i. Υψηλότερη τιμή για το testing accuracy στην εποχή 135 με ποσοστό 53.20%. Ήδη μετά την εποχή 145 το learning rate ήταν ίσο με 0.000138 και μέχρι την εποχή 160 λόγω συνεχών αυξήσεων του train loss έφτασε το όριο το οποίο έθεσα και ήταν ίσο με 0.000001.

4. Σύγκριση με την ενδιάμεση εργασία

Τα αποτελέσματα της ενδιάμεσης εργασίας φαίνονται παρακάτω:

Κατηγοριοποιητής	Ακρίβεια	Χρόνος (προσωπική υλοποίηση)	Χρόνος (έτοιμη υλοποίηση)
1 πλησιέστερου γείτονα (1-NN)	0.35	~9566 seconds	~74.3 seconds
3 πλησιέστερων γειτόνων (3-NN)	0.33	~9695 seconds	~78.4 seconds
Πλησιέστερου κέντρου (NCC)	0.28	~4.8 seconds	~0.9 seconds

Το CIFAR-10 περιέχει σύνθετα δεδομένα, με σημαντική επικάλυψη μεταξύ των κλάσεων (π.χ., γάτες και σκύλοι). Αυτό το καθιστά απαιτητικό για απλούς κατηγοριοποιητές, όπως ο **Nearest Neighbor (NN)** και ο **Nearest Class Centroid (NCC)**, που υπολογίζουν τις αποστάσεις των σημείων του test set από τα σημεία του training set. Οι κατηγοριοποιητές NN και NCC δυσκολεύονται να συλλάβουν την πολυπλοκότητα των χαρακτηριστικών των εικόνων, κάτι που φαίνεται από τις χαμηλές ακρίβειές τους. Το **νευρωνικό δίκτυο** μπορεί να εκμεταλλευτεί τη δομή των εικόνων, προσαρμόζοντας τα βάρη του για να αναγνωρίζει σχέσεις και μοτίβα μέσω της εκπαίδευσης και μπορεί να μοντελοποιήσει μη γραμμικά όρια απόφασης για την κατηγοριοποίηση των εικόνων.

Λαμβάνοντας υπόψη τα αποτελέσματα για το νευρωνικό δίκτυο από το **κεφάλαιο 3** συνάγουμε τα παρακάτω συμπεράσματα:

- Το **νευρωνικό δίκτυο** υπερέχει σημαντικά σε ακρίβεια, με ακρίβεια ~ **54%** στο test set. Αυτή είναι **καλύτερη κατά ~19-26%** σε σχέση με τον 1-NN (35%) και τον 3-NN (33%) και πολύ υψηλότερη από τον NCC (28%), καθιστώντας το την καλύτερη επιλογή για πιο αξιόπιστη κατηγοριοποίηση.
- Ο χρόνος εκπαίδευσης για το νευρωνικό δίκτυο εξαρτάται από τον αριθμό των εποχών για τις οποίες εκπαιδεύεται, αν και ήδη από τις πρώτες εποχές, δηλαδή μετά από μερικά δευτερόλεπτα, έχει ήδη καλύτερη επίδοση από NN και NCC και είναι πολύ πιο γρήγορο από τις προσωπικές υλοποιήσεις των κατηγοριοποιητών (για τις έτοιμες υλοποιήσεις ειδικά για τον NCC είναι αρκετά μικροί οι χρόνοι κάτι που οφείλεται στην απουσία εκπαίδευσης).