In [ ]:

```python
import numpy as np


def fc_forward(x, w, b):
    """
    Computes the forward pass for a fully-connected layer.

    The input x has shape (N, Din) and contains a minibatch of N
    examples, where each example x[i] has shape (Din,).

    Inputs:
    - x: A numpy array of shape (N, Din) giving input data
    - w: A numpy array of shape (Din, Dout) giving weights
    - b: A numpy array of shape (Dout,) giving biases

    Returns a tuple of:
    - out: output, of shape (N, Dout)
    - cache: (x, w, b)
    """
    out = None
    ###########################################################################
    # TODO: Implement the forward pass. Store the result in out.              #
    ###########################################################################

    # Ο υπολογισμός out=x·w+b ,
    # out=x·w+b προκύπτει από τον τρόπο με τον οποίο λειτουργεί ένα πλήρως συνδεδεμένο επίπεδο (fully connected layer) σε ένα νευρωνικό δίκτυο.
    out = np.dot(x, w) + b
    ###########################################################################
    #                              END OF YOUR CODE                           #
    ###########################################################################
    cache = (x, w, b)
    return out, cache


def fc_backward(grad_out, cache):
    """
    Computes the backward pass for a fully-connected layer.

    Inputs:
    - grad_out: Numpy array of shape (N, Dout) giving upstream gradients
    - cache: Tuple of:
      - x: A numpy array of shape (N, Din) giving input data
      - w: A numpy array of shape (Din, Dout) giving weights
      - b: A numpy array of shape (Dout,) giving biases

    Returns a tuple of downstream gradients:
    - grad_x: A numpy array of shape (N, Din) of gradient with respect to x
    - grad_w: A numpy array of shape (Din, Dout) of gradient with respect to w
    - grad_b: A numpy array of shape (Dout,) of gradient with respect to b
    """
    x, w, b = cache
    grad_x, grad_w, grad_b = None, None, None
    ###########################################################################
    # TODO: Implement the backward pass for the fully-connected layer         #
    ###########################################################################

    # Ο τύπος για την παράγωγο ως προς x είναι: dLoss/dx = dLoss/dout * dout/dx = dLoss/dout * w^T
    grad_x = np.dot(grad_out, w.T)


    # Ο τύπος για την παράγωγο ως προς w είναι: dLoss/dw = dLoss/dout * dout/dw = x^T * dLoss/dout
    grad_w = np.dot(x.T, grad_out)


    # Ο τύπος για την παράγωγο ως προς b είναι: dLoss/db = Σ ( dLoss/dout_i * dout_i/db )
    grad_b = np.sum(grad_out, axis=0)  # dLoss/db = άθροισμα(dLoss/dout)


    ###########################################################################
    #                              END OF YOUR CODE                           #
    ###########################################################################
    return grad_x, grad_w, grad_b


def relu_forward(x):
    """
    Computes the forward pass for the Rectified Linear Unit (ReLU) nonlinearity

    Input:
    - x: A numpy array of inputs, of any shape

    Returns a tuple of:
    - out: A numpy array of outputs, of the same shape as x
    - cache: x
    """
    out = None
    ###########################################################################
    # TODO: Implement the ReLU forward pass.                                  #
    ###########################################################################
    out = np.zeros_like(x)

    # Εφαρμογή της συνάρτησης ReLU σε κάθε στοιχείο του x
    for i in range(x.shape[0]):
      for j in range(x.shape[1]):
          out[i, j] = max(0, x[i, j])
    ###########################################################################
    #                              END OF YOUR CODE                           #
    ###########################################################################
    cache = x
    return out, cache


def relu_backward(grad_out, cache):
    """
    Computes the backward pass for a Rectified Linear Unit (ReLU) nonlinearity

    Input:
    - grad_out: Upstream derivatives, of any shape
    - cache: Input x, of same shape as dout

    Returns:
    - grad_x: Gradient with respect to x
    """
    grad_x, x = None, cache
    ###########################################################################
    # TODO: Implement the ReLU backward pass.                                 #
    ###########################################################################
    grad_x = np.zeros_like(x)  # Δημιουργία ενός πίνακα μηδενικών με το ίδιο σχήμα με το x

    # Εφαρμογή της συνάρτησης ReLU σε κάθε στοιχείο του x
    for i in range(x.shape[0]):
      for j in range(x.shape[1]):
        if x[i, j] > 0:
            grad_x[i, j] = grad_out[i, j]
        else:
            grad_x[i, j] = 0

    ###########################################################################
    #                              END OF YOUR CODE                           #
    ###########################################################################
    return grad_x


def l2_loss(x, y):
    """
    Computes the loss and gradient of L2 loss.

    loss = 0.5 * sum_i (x_i - y_i)**2 / N

    Inputs:
    - x: Input data, of shape (N, D)
    - y: Output data, of shape (N, D)

    Returns a tuple of:
    - loss: Scalar giving the loss
    - grad_x: Gradient of the loss with respect to x
    """
    N = x.shape[0]
    diff = x - y
    loss = 0.5 * np.sum(diff * diff) / N
    grad_x = diff / N
    return loss, grad_x


def softmax_loss(x, y):
    """
    Computes the loss and gradient for softmax (cross-entropy) loss function.

    Inputs:
    - x: Numpy array of shape (N, C) giving predicted class scores, where
      x[i, c] gives the predicted score for class c on input sample i
    - y: Numpy array of shape (N,) giving ground-truth labels, where
      y[i] = c means that input sample i has ground truth label c, where
      0 <= c < C.

    Returns a tuple of:
    - loss: Scalar giving the loss
    - grad_x: Numpy array of shape (N, C) giving the gradient of the loss with
      with respect to x
    """
    # Number of training examples
    N = x.shape[0]

    # Shift the logits by subtracting the maximum value in each row for numerical stability
    x_shifted = x - np.max(x, axis=1, keepdims=True)

    # Compute the softmax scores
    exp_scores = np.exp(x_shifted)
    softmax_scores = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

    # Compute the cross-entropy loss
    correct_class_scores = softmax_scores[np.arange(N), y]
    loss = -np.sum(np.log(correct_class_scores)) / N

    # Compute the gradient of the loss with respect to x
    grad_x = softmax_scores.copy()
    grad_x[np.arange(N), y] -= 1
    grad_x /= N

    return loss, grad_x


def l2_regularization(w, reg):
    """
    Computes loss and gradient for L2 regularization of a weight matrix:

    loss = (reg / 2) * sum_i w_i^2

    Where the sum ranges over all elements of w.

    Inputs:
    - w: Numpy array of any shape
    - reg: float giving the regularization strength

    Returns:
    """
    loss, grad_w = None, None
    ###########################################################################
    # TODO: Implement L2 regularization.                                      #
    ###########################################################################
    loss = (reg / 2) * np.sum(w ** 2)

    grad_w = reg * w
    ###########################################################################
    #                              END OF YOUR CODE                           #
    ###########################################################################
    return loss, grad_w
```

In [ ]:

```
[Running] python -u "c:\Users\hacknet13\Desktop\οραση\assignment2\gradcheck_layers.py"
Running numeric gradient check for fc
  grad_x difference:  5.212257292441791e-10
  grad_w difference:  3.5200753423225706e-10
  grad_b difference:  1.7937495933040282e-10
Running numeric gradient check for relu
  grad_x difference:  9.115641574908295e-11
Running numeric gradient check for softmax loss
  grad_x difference:  1.03187236533131e-10
Running numeric gradient check for L2 regularization
  grad_w difference:  4.354671657991194e-11
```

In [ ]:

```python
import numpy as np
from classifier import Classifier
from layers import fc_forward, fc_backward, relu_forward, relu_backward


class TwoLayerNet(Classifier):
    """
    A neural network with two layers, using a ReLU nonlinearity on its one
    hidden layer. That is, the architecture should be:

    input -> FC layer -> ReLU layer -> FC layer -> scores
    """
    def __init__(self, input_dim=3072, num_classes=10, hidden_dim=512,
                 weight_scale=1e-3):
        """
        Initialize a new two layer network.

        Inputs:
        - input_dim: The number of dimensions in the input.
        - num_classes: The number of classes over which to classify
        - hidden_dim: The size of the hidden layer
        - weight_scale: The weight matrices of the model will be initialized
          from a Gaussian distribution with standard deviation equal to
          weight_scale. The bias vectors of the model will always be
          initialized to zero.
        """
        #######################################################################
        # TODO: Initialize the weights and biases of a two-layer network.     #
        #######################################################################


        self.W1 = weight_scale * np.random.randn(input_dim, hidden_dim)
        self.b1 = np.zeros(hidden_dim)
        self.W2 = weight_scale * np.random.randn(hidden_dim, num_classes)
        self.b2 = np.zeros(num_classes)
        self.params = self.parameters()
        #######################################################################
        #                          END OF YOUR CODE                           #
        #######################################################################

    def parameters(self):
        params = None
        #######################################################################
        # TODO: Build a dict of all learnable parameters of this model.       #
        #######################################################################

        params = {
            'W1':self.W1,
            'b1':self.b1,
            'W2':self.W2,
            'b2':self.b2,
        }

        #######################################################################
        #                          END OF YOUR CODE                           #
        #######################################################################
        return params

    def forward(self, X):
        scores, cache = None, None
        #######################################################################
        # TODO: Implement the forward pass to compute classification scores   #
        # for the input data X. Store into cache any data that will be needed #
        # during the backward pass.                                           #
        #######################################################################
        hiden_layer , hiden_layer_cache  = fc_forward(X,self.params['W1'],self.params['b1'])
        relu_layer,relu_cache = relu_forward(hiden_layer)
        scores,fc2_cache = fc_forward(relu_layer,self.params['W2'],self.params['b2'])

        # Αποθήκευση των cache σε ένα dictionary
        cache = {
            'hiden_layer_cache': hiden_layer_cache,
            'relu_cache': relu_cache,
            'fc2_cache': fc2_cache
        }
        #######################################################################
        #                          END OF YOUR CODE                           #
        #######################################################################
        return scores, cache

    def backward(self, grad_scores, cache):
        grads = None
        #######################################################################
        # TODO: Implement the backward pass to compute gradients for all      #
        # learnable parameters of the model, storing them in the grads dict   #
        # above. The grads dict should give gradients for all parameters in   #
        # the dict returned by model.parameters().                            #
        #######################################################################

        # input <- FC layer <- ReLU layer <- FC layer <- scores

        hiden_layer_cache = cache['hiden_layer_cache']
        relu_cache = cache['relu_cache']
        fc2_cache = cache['fc2_cache']


        drelu_layer,grad_W2,grad_b2 = fc_backward(grad_scores,fc2_cache)
        dhidden_layer  = relu_backward(drelu_layer,relu_cache)
        grad_f2,grad_W1,grad_b1 = fc_backward(dhidden_layer,hiden_layer_cache)

        grads = {
            'W1': grad_W1,
            'b1': grad_b1,
            'W2': grad_W2,
            'b2': grad_b2,
        }

        #######################################################################
        #                          END OF YOUR CODE                           #
        #######################################################################
        return grads
```

In [ ]:

```
[Running] python -u "c:\Users\hacknet13\Desktop\οραση\assignment2\gradcheck_classifier.py"
Running numeric gradient check for LinearClassifier
  Max diff for grad_W:  1.734723475976807e-13
  Max diff for grad_b:  1.5076828674409626e-13
Running numeric gradient check for TwoLayerNet
  Max diff for grad_W1:  7.745540320236444e-16
  Max diff for grad_b1:  6.336077496005288e-16
  Max diff for grad_W2:  5.126107871511465e-16
  Max diff for grad_b2:  4.440892098500626e-16

[Done] exited with code=0 in 0.264 seconds
```

In [ ]:

```python
import argparse
import numpy as np
import matplotlib.pyplot as plt

from data import load_cifar10, DataSampler
from linear_classifier import LinearClassifier
from two_layer_net import TwoLayerNet
from optim import SGD
from layers import softmax_loss, l2_regularization
from utils import check_accuracy


parser = argparse.ArgumentParser()
parser.add_argument(
    '--plot-file',
    default='plot.pdf',
    help='File where loss and accuracy plot should be saved')
parser.add_argument(
    '--checkpoint-file',
    default='checkpoint.pkl',
    help='File where trained model weights should be saved')
parser.add_argument(
    '--print-every',
    type=int,
    default=25,
    help='How often to print losses during training')


def main(args):
    # How much data to use for training
    num_train = 25000

    # Model architecture hyperparameters.
    hidden_dim = 128 #64

    # Optimization hyperparameters.
    batch_size = 64 # 64
    num_epochs = 15 #30
    learning_rate = 0.1 #0.01
    reg = 0.001 #0.01

    ###########################################################################
    # TODO: Set hyperparameters for training your model. You can change any   #
    # of the hyperparameters above.                                           #
    ###########################################################################
     # How much data to use for training
    # Ολες οι τιμες βρεθηκαν πειραματικα /τυχαια
    num_train = 25000

    # Model architecture hyperparameters.
    hidden_dim = 128

    # Optimization hyperparameters.
    batch_size = 64
    num_epochs = 15
    learning_rate = 0.1
    reg = 0.001
    ###########################################################################
    #                           END OF YOUR CODE                              #
    ###########################################################################

    data = load_cifar10(num_train=num_train)
    train_sampler = DataSampler(data['X_train'], data['y_train'], batch_size)
    val_sampler = DataSampler(data['X_val'], data['y_val'], batch_size)

    # Set up the model and optimizer
    model = TwoLayerNet(hidden_dim=hidden_dim)
    optimizer = SGD(model.parameters(), learning_rate=learning_rate)

    stats = {
        't': [],
        'loss': [],
        'train_acc': [],
        'val_acc': [],
    }

    for epoch in range(1, num_epochs + 1):
        print(f'Starting epoch {epoch} / {num_epochs}')
        for i, (X_batch, y_batch) in enumerate(train_sampler):
            loss, grads = training_step(model, X_batch, y_batch, reg)
            optimizer.step(grads)
            if i % args.print_every == 0:
                print(f'  Iteration {i} / {len(train_sampler)}, loss = {loss}')
            stats['t'].append(i / len(train_sampler) + epoch - 1)
            stats['loss'].append(loss)

        print('Checking accuracy')
        train_acc = check_accuracy(model, train_sampler)
        print(f'  Train: {train_acc:.2f}')
        val_acc = check_accuracy(model, val_sampler)
        print(f'  Val:   {val_acc:.2f}')
        stats['train_acc'].append(train_acc)
        stats['val_acc'].append(val_acc)

    print(f'Saving plot to {args.plot_file}')
    plot_stats(stats, args.plot_file)
    print(f'Saving model checkpoint to {args.checkpoint_file}')
    model.save(args.checkpoint_file)


def training_step(model, X_batch, y_batch, reg):
    """
    Compute the loss and gradients for a single training iteration of a model
    given a minibatch of data. The loss should be a sum of a cross-entropy loss
    between the model predictions and the ground-truth image labels, and
    an L2 regularization term on all weight matrices in the fully-connected
    layers of the model. You should not regularize the bias vectors.

    Inputs:
    - model: A Classifier instance
    - X_batch: A numpy array of shape (N, D) giving a minibatch of images
    - y_batch: A numpy array of shape (N,) where 0 <= y_batch[i] < C is the
      ground-truth label for the image X_batch[i]
    - reg: A float giving the strength of L2 regularization to use.

    Returns a tuple of:
    - loss: A float giving the loss (data loss + regularization loss) for the
      model on this minibatch of data
    - grads: A dictionary giving gradients of the loss with respect to the
      parameters of the model. In particular grads[k] should be the gradient
      of the loss with respect to model.parameters()[k].
    """
    loss, grads = None, {}
    ###########################################################################
    # TODO: Compute the loss and gradient for one training iteration.         #
    ###########################################################################

     # Forward pass
    scores, cache = model.forward(X_batch)

    # data loss με softmax
    loss_softmax,grad_x = softmax_loss(scores,y_batch)

    # regularization loss
    reg_loss = 0
    for param in model.parameters():
        if param.startswith('W'):
            loss_l2_regularization,grad_w = l2_regularization( model.parameters()[param],reg)
            reg_loss += loss_l2_regularization

    # Total loss
    loss = loss_softmax + reg_loss


    # Backward pass
    grads = model.backward(grad_x, cache)

    # Add regularization gradient
    for param in model.parameters():
        if param.startswith('W'):
            grads[param] += l2_regularization(model.parameters()[param],reg)[1]
    ###########################################################################
    #                              END OF YOUR CODE                           #
    ###########################################################################
    return loss, grads


def plot_stats(stats, filename):
    plt.subplot(1, 2, 1)
    plt.plot(stats['t'], stats['loss'], 'o', alpha=0.5, ms=4)
    plt.title('Loss')
    plt.xlabel('Epoch')
    loss_xlim = plt.xlim()

    plt.subplot(1, 2, 2)
    epoch = np.arange(1, 1 + len(stats['train_acc']))
    plt.plot(epoch, stats['train_acc'], '-o', label='train')
    plt.plot(epoch, stats['val_acc'], '-o', label='val')
    plt.xlim(loss_xlim)
    plt.title('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(loc='upper left')

    plt.gcf().set_size_inches(12, 4)
    plt.savefig(filename, bbox_inches='tight')
    plt.clf()


if __name__ == '__main__':
    main(parser.parse_args())
```

In [ ]:

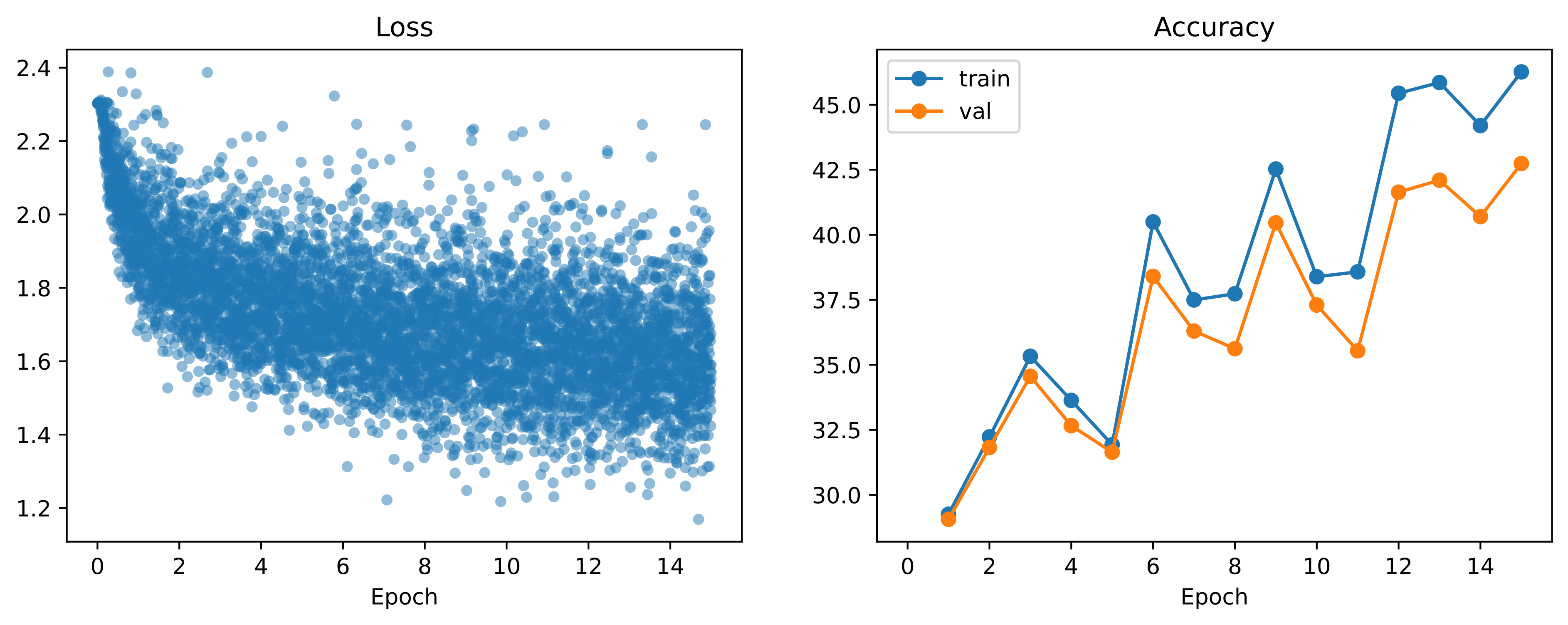
```
[Running] python -u "c:\Users\hacknet13\Desktop\οραση\assignment2\train.py"
Starting epoch 1 / 15
  Iteration 0 / 390, loss = 2.302812487130162
  Iteration 25 / 390, loss = 2.302247769744271
  Iteration 50 / 390, loss = 2.2769513000500488
  Iteration 75 / 390, loss = 2.1337484288711472
  Iteration 100 / 390, loss = 2.152143450545982
  Iteration 125 / 390, loss = 2.034679016074625
  Iteration 150 / 390, loss = 2.1002640195030917
  Iteration 175 / 390, loss = 2.0849698447365896
  Iteration 200 / 390, loss = 2.098860917372953
  Iteration 225 / 390, loss = 2.0030333154738393
  Iteration 250 / 390, loss = 2.080701750827813
  Iteration 275 / 390, loss = 2.055137069912502
  Iteration 300 / 390, loss = 2.0478431945407576
  Iteration 325 / 390, loss = 2.16213865784662
  Iteration 350 / 390, loss = 1.8438728764669037
  Iteration 375 / 390, loss = 1.9404651979371248
Checking accuracy
  Train: 29.26
  Val:   29.06
Starting epoch 2 / 15
  Iteration 0 / 390, loss = 1.802015293972788
  Iteration 25 / 390, loss = 1.9062247385467046
  Iteration 50 / 390, loss = 1.9875072008268875
  Iteration 75 / 390, loss = 1.965407012359769
  Iteration 100 / 390, loss = 1.8729739746418415
  Iteration 125 / 390, loss = 1.9537316797469524
  Iteration 150 / 390, loss = 1.8265086920000944
  Iteration 175 / 390, loss = 1.8304660164972633
  Iteration 200 / 390, loss = 1.9540957803071093
  Iteration 225 / 390, loss = 1.8423138563360673
  Iteration 250 / 390, loss = 1.7215526622897306
  Iteration 275 / 390, loss = 1.8436482889086725
  Iteration 300 / 390, loss = 1.6603242287774551
  Iteration 325 / 390, loss = 2.0085837537719935
  Iteration 350 / 390, loss = 1.807436407651696
  Iteration 375 / 390, loss = 1.7790535171477353
Checking accuracy
  Train: 32.23
  Val:   31.82
Starting epoch 3 / 15
  Iteration 0 / 390, loss = 1.8760393846586774
  Iteration 25 / 390, loss = 1.8089585216756998
  Iteration 50 / 390, loss = 1.901058349091459
  Iteration 75 / 390, loss = 1.8266456046848578
  Iteration 100 / 390, loss = 1.7455485520704463
  Iteration 125 / 390, loss = 1.8098830472875742
  Iteration 150 / 390, loss = 1.6378087970846789
  Iteration 175 / 390, loss = 1.6997427591750895
  Iteration 200 / 390, loss = 1.6208364886815334
  Iteration 225 / 390, loss = 1.7630047082363633
  Iteration 250 / 390, loss = 1.9792444083934253
  Iteration 275 / 390, loss = 1.6822973888730706
  Iteration 300 / 390, loss = 1.878922105140713
  Iteration 325 / 390, loss = 1.6891645220846179
  Iteration 350 / 390, loss = 1.6686017745088293
  Iteration 375 / 390, loss = 2.114635009302929
Checking accuracy
  Train: 35.33
  Val:   34.56
Starting epoch 4 / 15
  Iteration 0 / 390, loss = 1.6704925379509143
  Iteration 25 / 390, loss = 1.5847052709807135
  Iteration 50 / 390, loss = 1.9757798832638391
  Iteration 75 / 390, loss = 1.8557867905217305
  Iteration 100 / 390, loss = 1.8562783436583388
  Iteration 125 / 390, loss = 1.610880810842005
  Iteration 150 / 390, loss = 1.8130651794849755
  Iteration 175 / 390, loss = 1.7523245578081912
  Iteration 200 / 390, loss = 1.6779966202733687
  Iteration 225 / 390, loss = 1.7528849997759397
  Iteration 250 / 390, loss = 1.6118381176725796
  Iteration 275 / 390, loss = 1.809316906818264
  Iteration 300 / 390, loss = 1.794000295599542
  Iteration 325 / 390, loss = 1.859250917812006
  Iteration 350 / 390, loss = 1.7258625759305561
  Iteration 375 / 390, loss = 1.6871158114100246
Checking accuracy
  Train: 33.63
  Val:   32.66
Starting epoch 5 / 15
  Iteration 0 / 390, loss = 1.5928608520360699
  Iteration 25 / 390, loss = 1.787552911753
  Iteration 50 / 390, loss = 1.7573229335895237
  Iteration 75 / 390, loss = 1.8490103284593342
  Iteration 100 / 390, loss = 1.797686405846781
  Iteration 125 / 390, loss = 1.7036989263406934
  Iteration 150 / 390, loss = 1.6489186440697705
  Iteration 175 / 390, loss = 1.5801980109890839
  Iteration 200 / 390, loss = 1.7294566703118919
  Iteration 225 / 390, loss = 1.6290947741135584
  Iteration 250 / 390, loss = 1.8656518138695086
  Iteration 275 / 390, loss = 1.7758916320671068
  Iteration 300 / 390, loss = 1.8635795389529337
  Iteration 325 / 390, loss = 1.9051931585743622
  Iteration 350 / 390, loss = 1.9244196428779632
  Iteration 375 / 390, loss = 1.7015692438784589
Checking accuracy
  Train: 31.93
  Val:   31.64
Starting epoch 6 / 15
  Iteration 0 / 390, loss = 2.018971675070397
  Iteration 25 / 390, loss = 1.6288238234650003
  Iteration 50 / 390, loss = 1.5987655000103203
  Iteration 75 / 390, loss = 1.8642156536008316
  Iteration 100 / 390, loss = 1.558825593634985
  Iteration 125 / 390, loss = 1.8216544400473058
  Iteration 150 / 390, loss = 1.5889963929394977
  Iteration 175 / 390, loss = 1.7274310735139724
  Iteration 200 / 390, loss = 1.6514306174061157
  Iteration 225 / 390, loss = 1.658133446729895
  Iteration 250 / 390, loss = 1.831959617186306
  Iteration 275 / 390, loss = 1.7354058951557765
  Iteration 300 / 390, loss = 1.9110894773991243
  Iteration 325 / 390, loss = 1.6938985418342616
  Iteration 350 / 390, loss = 1.704590119404481
  Iteration 375 / 390, loss = 1.7229911910029323
Checking accuracy
  Train: 40.50
  Val:   38.40
Starting epoch 7 / 15
  Iteration 0 / 390, loss = 1.5789177113297865
  Iteration 25 / 390, loss = 1.7499493484462632
  Iteration 50 / 390, loss = 1.639532651514128
  Iteration 75 / 390, loss = 1.8339197295746337
  Iteration 100 / 390, loss = 1.6308760607029213
  Iteration 125 / 390, loss = 1.7167796093312104
  Iteration 150 / 390, loss = 1.6465026165481789
  Iteration 175 / 390, loss = 2.0868139599089934
  Iteration 200 / 390, loss = 1.7371645538504243
  Iteration 225 / 390, loss = 1.7074315498395776
  Iteration 250 / 390, loss = 1.6692927524353038
  Iteration 275 / 390, loss = 1.6566710886558593
  Iteration 300 / 390, loss = 1.7627764499923462
  Iteration 325 / 390, loss = 1.6040125806343504
  Iteration 350 / 390, loss = 2.002236782620586
  Iteration 375 / 390, loss = 1.5892360550142361
Checking accuracy
  Train: 37.50
  Val:   36.30
Starting epoch 8 / 15
  Iteration 0 / 390, loss = 1.9760729420058074
  Iteration 25 / 390, loss = 1.9931983405070504
  Iteration 50 / 390, loss = 1.631982365791676
  Iteration 75 / 390, loss = 1.9430158332765148
  Iteration 100 / 390, loss = 1.6461918157372748
  Iteration 125 / 390, loss = 1.7710908939936902
  Iteration 150 / 390, loss = 1.6029703227507315
  Iteration 175 / 390, loss = 1.4929179341675538
  Iteration 200 / 390, loss = 1.6977633198939475
  Iteration 225 / 390, loss = 1.6388774520081462
  Iteration 250 / 390, loss = 1.5631993299885654
  Iteration 275 / 390, loss = 1.5877519842992653
  Iteration 300 / 390, loss = 1.8620947661241007
  Iteration 325 / 390, loss = 1.6282771086896874
  Iteration 350 / 390, loss = 1.505497689363071
  Iteration 375 / 390, loss = 1.5151486167473414
Checking accuracy
  Train: 37.73
  Val:   35.62
Starting epoch 9 / 15
  Iteration 0 / 390, loss = 1.6141198434818722
  Iteration 25 / 390, loss = 1.3583345726687934
  Iteration 50 / 390, loss = 1.719798818779352
  Iteration 75 / 390, loss = 1.5732085584837379
  Iteration 100 / 390, loss = 1.654123869170234
  Iteration 125 / 390, loss = 1.768664064651189
  Iteration 150 / 390, loss = 1.6021421086081842
  Iteration 175 / 390, loss = 1.6124270489358414
  Iteration 200 / 390, loss = 1.9140391702235207
  Iteration 225 / 390, loss = 1.630508425596144
  Iteration 250 / 390, loss = 1.7330043114713827
  Iteration 275 / 390, loss = 1.85326560323153
  Iteration 300 / 390, loss = 1.511888984285417
  Iteration 325 / 390, loss = 1.7814128705273315
  Iteration 350 / 390, loss = 1.7296180494976787
  Iteration 375 / 390, loss = 1.4960425883463022
Checking accuracy
  Train: 42.53
  Val:   40.46
Starting epoch 10 / 15
  Iteration 0 / 390, loss = 1.5760547658296882
  Iteration 25 / 390, loss = 1.3634582717979664
  Iteration 50 / 390, loss = 1.8124273288999135
  Iteration 75 / 390, loss = 1.7634297632024611
  Iteration 100 / 390, loss = 1.667300663709414
  Iteration 125 / 390, loss = 1.462076859856392
  Iteration 150 / 390, loss = 1.845326081063442
  Iteration 175 / 390, loss = 1.433952855153633
  Iteration 200 / 390, loss = 1.4376373928948059
  Iteration 225 / 390, loss = 1.6197659963060462
  Iteration 250 / 390, loss = 1.5826671356924047
  Iteration 275 / 390, loss = 1.7436626517095375
  Iteration 300 / 390, loss = 1.4613603570975855
  Iteration 325 / 390, loss = 1.5747363930530214
  Iteration 350 / 390, loss = 1.5625465303260189
  Iteration 375 / 390, loss = 1.5043938628386915
Checking accuracy
  Train: 38.39
  Val:   37.30
Starting epoch 11 / 15
  Iteration 0 / 390, loss = 1.7805730983372947
  Iteration 25 / 390, loss = 1.4359908638079077
  Iteration 50 / 390, loss = 1.4543102368798633
  Iteration 75 / 390, loss = 1.4665410788740048
  Iteration 100 / 390, loss = 1.6267336538718833
  Iteration 125 / 390, loss = 2.013311726212984
  Iteration 150 / 390, loss = 1.853382646349985
  Iteration 175 / 390, loss = 1.5236054612555725
  Iteration 200 / 390, loss = 1.5621146030788526
  Iteration 225 / 390, loss = 1.6167487304655013
  Iteration 250 / 390, loss = 1.7114455984049195
  Iteration 275 / 390, loss = 1.7693567400891286
  Iteration 300 / 390, loss = 1.7592221096252971
  Iteration 325 / 390, loss = 1.7281106211467412
  Iteration 350 / 390, loss = 1.5429931927788334
  Iteration 375 / 390, loss = 1.5155853765572913
Checking accuracy
  Train: 38.58
  Val:   35.54
Starting epoch 12 / 15
  Iteration 0 / 390, loss = 1.672019214776769
  Iteration 25 / 390, loss = 1.4178743843016541
  Iteration 50 / 390, loss = 1.6083405830806476
  Iteration 75 / 390, loss = 1.6651226741047722
  Iteration 100 / 390, loss = 1.6417718406286657
  Iteration 125 / 390, loss = 1.5844939787797412
  Iteration 150 / 390, loss = 1.6517690689480526
  Iteration 175 / 390, loss = 1.477691961408688
  Iteration 200 / 390, loss = 1.5351247649882553
  Iteration 225 / 390, loss = 1.4364514625713412
  Iteration 250 / 390, loss = 1.5690248740028903
  Iteration 275 / 390, loss = 1.6772870506825914
  Iteration 300 / 390, loss = 1.606994074734868
  Iteration 325 / 390, loss = 1.5690210568093024
  Iteration 350 / 390, loss = 1.8113758611218524
  Iteration 375 / 390, loss = 1.6132928442234198
Checking accuracy
  Train: 45.44
  Val:   41.64
Starting epoch 13 / 15
  Iteration 0 / 390, loss = 1.7672957415404034
  Iteration 25 / 390, loss = 1.6845141710153995
  Iteration 50 / 390, loss = 1.519767357149524
  Iteration 75 / 390, loss = 1.4980187617996552
  Iteration 100 / 390, loss = 1.757012945058896
  Iteration 125 / 390, loss = 2.007309470515392
  Iteration 150 / 390, loss = 1.5536774627378642
  Iteration 175 / 390, loss = 1.5982265788658836
  Iteration 200 / 390, loss = 1.7925597899488912
  Iteration 225 / 390, loss = 1.6416416796220075
  Iteration 250 / 390, loss = 1.6843829673603163
  Iteration 275 / 390, loss = 1.470577960330473
  Iteration 300 / 390, loss = 1.729254596129988
  Iteration 325 / 390, loss = 1.7655198677505806
  Iteration 350 / 390, loss = 1.5082243520146845
  Iteration 375 / 390, loss = 1.6367702604470626
Checking accuracy
  Train: 45.86
  Val:   42.10
Starting epoch 14 / 15
  Iteration 0 / 390, loss = 1.701548879146913
  Iteration 25 / 390, loss = 1.796310704885206
  Iteration 50 / 390, loss = 1.721824609441506
  Iteration 75 / 390, loss = 1.7004960729059686
  Iteration 100 / 390, loss = 1.6307895092768872
  Iteration 125 / 390, loss = 1.6045074342857566
  Iteration 150 / 390, loss = 1.524937915890564
  Iteration 175 / 390, loss = 1.513859465563515
  Iteration 200 / 390, loss = 1.4852022882335423
  Iteration 225 / 390, loss = 1.7141548264631286
  Iteration 250 / 390, loss = 1.6891448636423547
  Iteration 275 / 390, loss = 1.3393094175006732
  Iteration 300 / 390, loss = 1.5876369133151458
  Iteration 325 / 390, loss = 1.6046729707683982
  Iteration 350 / 390, loss = 1.6622850774177451
  Iteration 375 / 390, loss = 1.7697793687411818
Checking accuracy
  Train: 44.20
  Val:   40.70
Starting epoch 15 / 15
  Iteration 0 / 390, loss = 1.565263919204722
  Iteration 25 / 390, loss = 1.5543777412316548
  Iteration 50 / 390, loss = 1.95167582504843
  Iteration 75 / 390, loss = 1.6262069919310354
  Iteration 100 / 390, loss = 1.3792817677993923
  Iteration 125 / 390, loss = 1.623381304965383
  Iteration 150 / 390, loss = 1.5356331851296028
  Iteration 175 / 390, loss = 1.5133314440506487
  Iteration 200 / 390, loss = 1.5076493581919979
  Iteration 225 / 390, loss = 1.5232863184031862
  Iteration 250 / 390, loss = 1.8096131326107996
  Iteration 275 / 390, loss = 1.6415491051862516
  Iteration 300 / 390, loss = 2.005980115338229
  Iteration 325 / 390, loss = 1.6466109043539137
  Iteration 350 / 390, loss = 1.5498422022465042
  Iteration 375 / 390, loss = 1.4972999950058532
Checking accuracy
  Train: 46.26
  Val:   42.74
Saving plot to plot.pdf
Saving model checkpoint to checkpoint.pkl

[Done] exited with code=0 in 145.073 seconds
```

In [ ]:

```
[Running] python -u "c:\Users\hacknet13\Desktop\οραση\assignment2\test.py"
Loading model from checkpoint.pkl
```

```
Test-set accuracy: 42.57
```

In [ ]:


Loss
2.4
2.2
2.0
1.8
1.6
1.4
1.2
0
2
4
6
8
10
12
14
Epoch
Accuracy
train
val
45.0
42.5
40.0
37.5
35.0
32.5
30.0
0
2
4
6
8
10
12
14
Epoch