```python
from abc import ABCMeta, abstractmethod
import numpy as np
import scipy
import scipy.stats as stats

from im2col_cython import col2im_cython, im2col_cython

zero_init = np.zeros

def variance_scaling_initializer(shape, fan_in, factor=2.0, seed=None):
  sigma = np.sqrt(factor / fan_in)
  return stats.truncnorm(-2, 2, loc=0, scale=sigma).rvs(shape)


# -- ABSTRACT CLASS DEFINITION --
class Layer(metaclass=ABCMeta):
  """Interface for layers"""
  # See documentation of abstract base classes (ABC): https://docs.python.org/3/library/abc.html

  @abstractmethod
  def forward(self, inputs):
    """
    Args:
      inputs: ndarray tensor.
    Returns:
      ndarray tensor, result of the forward pass.
    """
    pass

  @abstractmethod
  def backward_inputs(self, grads):
    """
    Args:
      grads: gradient of the loss with respect to the output of the layer.
    Returns:
      Gradient of the loss with respect to the input of the layer.
    """
    pass

  def backward_params(self, grads):
    """
    Args:
      grads: gradient of the loss with respect to the output of the layer.
    Returns:
      Gradient of the loss with respect to all the parameters of the layer as a list
      [[w0, g0], ..., [wk, gk], self.name] where w are parameter weights and g their gradient.
      Note that wk and gk must have the same shape.
    """
    pass


class Convolution(Layer):
  """N-dimensional convolution layer"""

  def __init__(self, input_layer, num_filters, kernel_size, name, padding='SAME',
               weights_initializer_fn=variance_scaling_initializer,
               bias_initializer_fn=zero_init):
    self.input_shape = input_layer.shape
    N, C, H, W = input_layer.shape
    self.C = C
    self.N = N
    self.num_filters = num_filters
    self.kernel_size = kernel_size

    assert kernel_size % 2 == 1
```

```
    self.padding = padding
    if padding == 'SAME':
      # with zero padding
      self.shape = (N, num_filters, H, W)
      self.pad = (kernel_size - 1) // 2
    else:
      # without padding
      self.shape = (N, num_filters, H - kernel_size + 1, W - kernel_size + 1)
      self.pad = 0

    fan_in = C * kernel_size**2
    self.weights = weights_initializer_fn([num_filters, kernel_size**2 * C], fan_in)
    self.bias = bias_initializer_fn([num_filters])
    # this implementation doesn't support strided convolutions
    self.stride = 1
    self.name = name
    self.has_params = True

  def forward(self, x):
    k = self.kernel_size
    self.x_cols = im2col_cython(x, k, k, self.pad, self.stride)
    res = self.weights.dot(self.x_cols) + self.bias.reshape(-1, 1)
    N, C, H, W = x.shape
    out = res.reshape(self.num_filters, self.shape[2], self.shape[3], N)
    return out.transpose(3, 0, 1, 2)

  def backward_inputs(self, grad_out):
    # nice trick from CS231n, backward pass can be done with just matrix mul and col2im
    grad_out = grad_out.transpose(1, 2, 3, 0).reshape(self.num_filters, -1)
    grad_x_cols = self.weights.T.dot(grad_out)
    N, C, H, W = self.input_shape
    k = self.kernel_size
    grad_x = col2im_cython(grad_x_cols, N, C, H, W, k, k, self.pad, self.stride)
    return grad_x

  def backward_params(self, grad_out):
    grad_bias = np.sum(grad_out, axis=(0, 2, 3))
    grad_out = grad_out.transpose(1, 2, 3, 0).reshape(self.num_filters, -1)
    grad_weights = grad_out.dot(self.x_cols.T).reshape(self.weights.shape)
    return [[self.weights, grad_weights], [self.bias, grad_bias], self.name]


class MaxPooling(Layer):
  def __init__(self, input_layer, name, pool_size=2, stride=2):
    self.name = name
    self.input_shape = input_layer.shape
    N, C, H, W = self.input_shape
    self.stride = stride
    self.shape = (N, C, H // stride, W // stride)
    self.pool_size = pool_size
    assert pool_size == stride, 'Invalid pooling params'
    assert H % pool_size == 0
    assert W % pool_size == 0
    self.has_params = False

  def forward(self, x):
    N, C, H, W = x.shape
    self.input_shape = x.shape
    # with this clever reshaping we can implement pooling where pool_size == stride
    self.x = x.reshape(N, C, H // self.pool_size, self.pool_size,
                       W // self.pool_size, self.pool_size)
    self.out = self.x.max(axis=3).max(axis=4)
    # if you are returning class member be sure to return a copy
    return self.out.copy()

  def backward_inputs(self, grad_out):
```

```
    grad_x = np.zeros_like(self.x)
    out_newaxis = self.out[:, :, :, np.newaxis, :, np.newaxis]
    mask = (self.x == out_newaxis)
    dout_newaxis = grad_out[:, :, :, np.newaxis, :, np.newaxis]
    dout_broadcast, _ = np.broadcast_arrays(dout_newaxis, grad_x)
    # this is almost the same as the real backward pass
    grad_x[mask] = dout_broadcast[mask]
    # in the very rare case that more then one input have the same max value
    # we can aprox the real gradient routing by evenly distributing across multiple inputs
    # but in almost all cases this sum will be 1
    grad_x /= np.sum(mask, axis=(3, 5), keepdims=True)
    grad_x = grad_x.reshape(self.input_shape)
    return grad_x


class Flatten(Layer):
  def __init__(self, input_layer, name):
    self.input_shape = input_layer.shape
    self.N = self.input_shape[0]
    self.num_outputs = 1
    for i in range(1, len(self.input_shape)):
      self.num_outputs *= self.input_shape[i]
    self.shape = (self.N, self.num_outputs)
    self.has_params = False
    self.name = name

  def forward(self, inputs):
    self.input_shape = inputs.shape
    inputs_flat = inputs.reshape(self.input_shape[0], -1)
    self.shape = inputs_flat.shape
    return inputs_flat

  def backward_inputs(self, grads):
    return grads.reshape(self.input_shape)


class FC(Layer):
  def __init__(self, input_layer, num_outputs, name,
           weights_initializer_fn=variance_scaling_initializer,
           bias_initializer_fn=zero_init):
    """
    Args:
      input_layer: layer below
      num_outputs: number of neurons in this layer
      weights_initializer_fn: initializer function for weights,
      bias_initializer_fn: initializer function for biases
    """

    self.input_shape = input_layer.shape
    self.N = self.input_shape[0]
    self.shape = (self.N, num_outputs)
    self.num_outputs = num_outputs

    self.num_inputs = 1
    for i in range(1, len(self.input_shape)):
      self.num_inputs *= self.input_shape[i]
    # weights.shape = out x in
    self.weights = weights_initializer_fn([num_outputs, self.num_inputs], fan_in=self.num_inputs)
    self.bias = bias_initializer_fn([num_outputs])
    self.name = name
    self.has_params = True

  def forward(self, inputs):
    """
    Args:
      inputs: ndarray of shape (N, num_inputs)
```

```python
    Returns:
      An ndarray of shape (N, num_outputs)
    """
    self.inputs = inputs
    outputs = inputs @ self.weights.T + self.bias
    return outputs

  def backward_inputs(self, grads):
    """
    Args:
      grads: ndarray of shape (N, num_outputs)
    Returns:
      An ndarray of shape (N, num_inputs)
    """
    return grads @ self.weights

  def backward_params(self, grads):
    """
    Args:
      grads: ndarray of shape (N, num_outputs)
    Returns:
      List of params and gradient pairs.
    """
    grad_weights = grads.T @ self.inputs
    grad_bias = np.sum(grads, axis=0)
    return [[self.weights, grad_weights], [self.bias, grad_bias], self.name]


class ReLU(Layer):
  def __init__(self, input_layer, name):
    self.shape = input_layer.shape
    self.name = name
    self.has_params = False

  def forward(self, inputs):
    """
    Args:
      inputs: ndarray of shape (N, C, H, W).
    Returns:
      ndarray of shape (N, C, H, W).
    """
    self.inputs = inputs
    return np.maximum(0, inputs)

  def backward_inputs(self, grads):
    """
    Args:
      grads: ndarray of shape (N, C, H, W).
    Returns:
      ndarray of shape (N, C, H, W).
    """
    # TODO vidit jel smijem tu mijenjat dobivene grads
    grads[self.inputs < 0] = 0
    return grads


def softmax(x):
  exp_x_shifted = np.exp(x - np.max(x))
  exp_sums = np.sum(exp_x_shifted, axis=1).reshape((-1, 1))
  probs = exp_x_shifted / exp_sums
  return probs


class SoftmaxCrossEntropyWithLogits():
  def __init__(self):
    self.has_params = False
```

```python
    def forward(self, x, y):
        """
        Args:
          x: ndarray of shape (N, num_classes).
          y: ndarray of shape (N, num_classes) - one-hot encoded obv
        Returns:
          Scalar, average loss over N examples.
          It is better to compute average loss here instead of just sum
          because then learning rate and weight decay won't depend on batch size.

        """
        probs = softmax(x)
        log_probs = np.log(probs)
        loss = -np.mean(log_probs * y)
        return loss

    def backward_inputs(self, x, y):
        """
        Args:
          x: ndarray of shape (N, num_classes).
          y: ndarray of shape (N, num_classes).
        Returns:
          Gradient with respect to the x, ndarray of shape (N, num_classes).
        """
        # Hint: don't forget that we took the average in the forward pass
        return (softmax(x) - y) / x.shape[0]


class L2Regularizer():
    def __init__(self, weights, weight_decay, name):
        """
        Args:
          weights: parameters which will be regularizerized
          weight_decay: lambda, regularization strength
          name: layer name
        """
        # this is still a reference to original tensor so don't change self.weights
        self.weights = weights
        self.weight_decay = weight_decay
        self.name = name

    def forward(self):
        """
         Returns:
          Scalar, loss due to the L2 regularization.
        """
        return np.linalg.norm(self.weights) * self.weight_decay

    def backward_params(self):
        """
        Returns:
          Gradient of the L2 loss with respect to the regularized weights.
        """
        grad_weights = self.weight_decay * self.weights
        return [[self.weights, grad_weights], self.name]


class RegularizedLoss():
    def __init__(self, data_loss, regularizer_losses):
        self.data_loss = data_loss
        self.regularizer_losses = regularizer_losses
        self.has_params = True
        self.name = 'RegularizedLoss'

    def forward(self, x, y):
```

```
    loss_val = self.data_loss.forward(x, y)
    for loss in self.regularizer_losses:
      loss_val += loss.forward()
    return loss_val

  def backward_inputs(self, x, y):
    return self.data_loss.backward_inputs(x, y)

  def backward_params(self):
    grads = []
    for loss in self.regularizer_losses:
      grads += [loss.backward_params()]
    return grads
```

```python
import numpy as np
import layers

def rel_error(x, y):
  """ returns relative error """
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))

def eval_numerical_gradient(f, x, df, h=1e-5):
  """
  Evaluate a numeric gradient for a function that accepts a numpy
  array and returns a numpy array.
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient at
  """
  grad = np.zeros_like(x)
  it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
  while not it.finished:
    ix = it.multi_index

    oldval = x[ix]
    x[ix] = oldval + h
    # evaluate f(x + h)
    pos = f(x.copy()).copy()
    x[ix] = oldval - h
    # evaluate f(x - h)
    neg = f(x.copy()).copy()
    x[ix] = oldval

    # compute the partial derivative with centered formula
    grad[ix] = np.sum((pos - neg) * df) / (2 * h)
    # step to next dimension
    it.iternext()
  return grad

def check_grad_inputs(layer, x, grad_out):
  """
  Args:
    layer: Layer object
    x: ndarray tensor input data
    grad_out: ndarray tensor gradient from the next layer
  """
  grad_x_num = eval_numerical_gradient(layer.forward, x, grad_out)
  grad_x = layer.backward_inputs(grad_out)
  print("Relative error = ", rel_error(grad_x_num, grad_x))
  print("Error norm = ", np.linalg.norm(grad_x_num - grad_x))

def check_grad_params(layer, x, w, b, grad_out):
  """
  Args:
    layer: Layer object
    x: ndarray tensor input data
    w: ndarray tensor layer weights
    b: ndarray tensor layer biases
    grad_out: ndarray tensor gradient from the next layer
  """
  func = lambda params: layer.forward(x)
  grad_w_num = eval_numerical_gradient(func, w, grad_out)
  grad_b_num = eval_numerical_gradient(func, b, grad_out)
  grads = layer.backward_params(grad_out)
  grad_w = grads[0][1]
  grad_b = grads[1][1]
  print("Check weights:")
  print("Relative error = ", rel_error(grad_w_num, grad_w))
  print("Error norm = ", np.linalg.norm(grad_w_num - grad_w))
  print("Check biases:")
  print("Relative error = ", rel_error(grad_b_num, grad_b))
```

```
  print("Error norm = ", np.linalg.norm(grad_b_num - grad_b))

print("Convolution")
x = np.random.randn(4, 3, 5, 5)
grad_out = np.random.randn(4, 2, 5, 5)
conv = layers.Convolution(x, 2, 3, "conv1")
print("Check grad wrt input")
check_grad_inputs(conv, x, grad_out)
print("Check grad wrt params")
check_grad_params(conv, x, conv.weights, conv.bias, grad_out)

print("\nMaxPooling")
x = np.random.randn(5, 4, 8, 8)
grad_out = np.random.randn(5, 4, 4, 4)
pool = layers.MaxPooling(x, "pool", 2, 2)
print("Check grad wrt input")
check_grad_inputs(pool, x, grad_out)

print("\nReLU")
x = np.random.randn(4, 3, 5, 5)
grad_out = np.random.randn(4, 3, 5, 5)
relu = layers.ReLU(x, "relu")
print("Check grad wrt input")
check_grad_inputs(relu, x, grad_out)

print("\nFC")
x = np.random.randn(20, 40)
grad_out = np.random.randn(20, 30)
fc = layers.FC(x, 30, "fc")
print("Check grad wrt input")
check_grad_inputs(fc, x, grad_out)
print("Check grad wrt params")
check_grad_params(fc, x, fc.weights, fc.bias, grad_out)

print("\nSoftmaxCrossEntropyWithLogits")
x = np.random.randn(50, 20)
y = np.zeros([50, 20])
y[:,0] = 1
loss = layers.SoftmaxCrossEntropyWithLogits()
grad_x_num = eval_numerical_gradient(lambda x: loss.forward(x, y), x, 1)
out = loss.forward(x, y)
grad_x = loss.backward_inputs(x, y)
print("Relative error = ", rel_error(grad_x_num, grad_x))
print("Error norm = ", np.linalg.norm(grad_x_num - grad_x))

print("\nL2Regularizer")
x = np.random.randn(5, 4, 8, 8)
grad_out = np.random.randn(5, 4, 4, 4)
l2reg = layers.L2Regularizer(x, 1e-2, 'L2reg')
print("Check grad wrt params")
func = lambda params: l2reg.forward()
grad_num = eval_numerical_gradient(func, l2reg.weights, 1)
grads = l2reg.backward_params()
grad = grads[0][1]
print("Relative error = ", rel_error(grad_num, grad))
print("Error norm = ", np.linalg.norm(grad_num - grad))
```

```
# The MIT License (MIT)
#
# Copyright (c) 2015 Andrej Karpathy
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all
# copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.

# build with:
# python setup_cython.py build_ext --inplace

import numpy as np
cimport numpy as np
cimport cython

# DTYPE = np.float64
# ctypedef np.float64_t DTYPE_t

ctypedef fused DTYPE_t:
    np.float32_t
    np.float64_t

def im2col_cython(np.ndarray[DTYPE_t, ndim=4] x, int field_height,
                  int field_width, int padding, int stride):
    cdef int N = x.shape[0]
    cdef int C = x.shape[1]
    cdef int H = x.shape[2]
    cdef int W = x.shape[3]

    cdef int HH = (H + 2 * padding - field_height) / stride + 1
    cdef int WW = (W + 2 * padding - field_width) / stride + 1

    cdef int p = padding
    cdef np.ndarray[DTYPE_t, ndim=4] x_padded = np.pad(x,
            ((0, 0), (0, 0), (p, p), (p, p)), mode='constant')

    cdef np.ndarray[DTYPE_t, ndim=2] cols = np.zeros(
            (C * field_height * field_width, N * HH * WW),
            dtype=x.dtype)

    # Moving the inner loop to a C function with no bounds checking works, but does
    # not seem to help performance in any measurable way.

    im2col_cython_inner(cols, x_padded, N, C, H, W, HH, WW,
                        field_height, field_width, padding, stride)
    return cols



@cython.boundscheck(False)
cdef int im2col_cython_inner(np.ndarray[DTYPE_t, ndim=2] cols,
                             np.ndarray[DTYPE_t, ndim=4] x_padded,
                             int N, int C, int H, int W, int HH, int WW,
                             int field_height, int field_width, int padding, int stride) except? -1:
```

```
    cdef int c, ii, jj, row, yy, xx, i, col

    for c in range(C):
        for yy in range(HH):
            for xx in range(WW):
                for ii in range(field_height):
                    for jj in range(field_width):
                        row = c * field_width * field_height + ii * field_height + jj
                        for i in range(N):
                            col = yy * WW * N + xx * N + i
                            cols[row, col] = x_padded[i, c, stride * yy + ii, stride * xx + jj]



def col2im_cython(np.ndarray[DTYPE_t, ndim=2] cols, int N, int C, int H, int W,
                  int field_height, int field_width, int padding, int stride):
    cdef np.ndarray x = np.empty((N, C, H, W), dtype=cols.dtype)
    cdef int HH = (H + 2 * padding - field_height) / stride + 1
    cdef int WW = (W + 2 * padding - field_width) / stride + 1
    cdef np.ndarray[DTYPE_t, ndim=4] x_padded = np.zeros((N, C, H + 2 * padding, W + 2 * padding),
                                        dtype=cols.dtype)

    # Moving the inner loop to a C-function with no bounds checking improves
    # performance quite a bit for col2im.
    col2im_cython_inner(cols, x_padded, N, C, H, W, HH, WW,
                        field_height, field_width, padding, stride)
    if padding > 0:
        return x_padded[:, :, padding:-padding, padding:-padding]
    return x_padded


@cython.boundscheck(False)
cdef int col2im_cython_inner(np.ndarray[DTYPE_t, ndim=2] cols,
                             np.ndarray[DTYPE_t, ndim=4] x_padded,
                             int N, int C, int H, int W, int HH, int WW,
                             int field_height, int field_width, int padding, int stride) except? -1:
    cdef int c, ii, jj, row, yy, xx, i, col

    for c in range(C):
        for ii in range(field_height):
            for jj in range(field_width):
                row = c * field_width * field_height + ii * field_height + jj
                for yy in range(HH):
                    for xx in range(WW):
                        for i in range(N):
                            col = yy * WW * N + xx * N + i
                            x_padded[i, c, stride * yy + ii, stride * xx + jj] += cols[row, col]
```

```python
import time
from pathlib import Path

import numpy as np
from torchvision.datasets import MNIST

import nn
import layers
import layers

DATA_DIR = Path(__file__).parent / 'data'
SAVE_DIR = Path(__file__).parent / 'out'

config = {}
config['max_epochs'] = 8
config['batch_size'] = 50
config['save_dir'] = SAVE_DIR
config['weight_decay'] = 1e-1
config['lr_policy'] = {
    1: {'lr': 1e-1},
    3: {'lr': 1e-2},
    5: {'lr': 1e-3},
    7: {'lr': 1e-4}
}

def dense_to_one_hot(y, class_count):
    return np.eye(class_count)[y]

#np.random.seed(100)
np.random.seed(int(time.time() * 1e6) % 2**31)

ds_train, ds_test = MNIST(DATA_DIR, train=True, download=False), MNIST(DATA_DIR, train=False)
train_x = ds_train.data.reshape([-1, 1, 28, 28]).numpy().astype(np.float) / 255
train_y = ds_train.targets.numpy()
train_x, valid_x = train_x[:55000], train_x[55000:]
train_y, valid_y = train_y[:55000], train_y[55000:]
test_x = ds_test.data.reshape([-1, 1, 28, 28]).numpy().astype(np.float) / 255
test_y = ds_test.targets.numpy()
train_mean = train_x.mean()
train_x, valid_x, test_x = (x - train_mean for x in (train_x, valid_x, test_x))
train_y, valid_y, test_y = (dense_to_one_hot(y, 10) for y in (train_y, valid_y, test_y))

weight_decay = config['weight_decay']
net = []
regularizers = []
inputs = np.random.randn(config['batch_size'], 1, 28, 28)
net += [layers.Convolution(inputs, 16, 5, "conv1")]
regularizers += [layers.L2Regularizer(net[-1].weights, weight_decay, 'conv1_l2reg')]
net += [layers.MaxPooling(net[-1], "pool1")]
net += [layers.ReLU(net[-1], "relu1")]
net += [layers.Convolution(net[-1], 32, 5, "conv2")]
regularizers += [layers.L2Regularizer(net[-1].weights, weight_decay, 'conv2_l2reg')]
net += [layers.MaxPooling(net[-1], "pool2")]
net += [layers.ReLU(net[-1], "relu2")]
## 7x7
net += [layers.Flatten(net[-1], "flatten3")]
net += [layers.FC(net[-1], 512, "fc3")]
regularizers += [layers.L2Regularizer(net[-1].weights, weight_decay, 'fc3_l2reg')]
net += [layers.ReLU(net[-1], "relu3")]
net += [layers.FC(net[-1], 10, "logits")]

data_loss = layers.SoftmaxCrossEntropyWithLogits()
loss = layers.RegularizedLoss(data_loss, regularizers)

nn.train(train_x, train_y, valid_x, valid_y, net, loss, config)
nn.evaluate("Test", test_x, test_y, net, loss, config)
```

```python
import os
import math

import numpy as np
import skimage as ski
import skimage.io


def forward_pass(net, inputs):
  output = inputs
  for layer in net:
    output = layer.forward(output)
  return output


def backward_pass(net, loss, x, y):
  grads = []
  grad_out = loss.backward_inputs(x, y)
  if loss.has_params:
    grads += loss.backward_params()
  for layer in reversed(net):
    grad_inputs = layer.backward_inputs(grad_out)
    if layer.has_params:
      grads += [layer.backward_params(grad_out)]
    grad_out = grad_inputs
  return grads

def sgd_update_params(grads, config):
  lr = config['lr']
  for layer_grads in grads:
    for i in range(len(layer_grads) - 1):
      params = layer_grads[i][0]
      grads = layer_grads[i][1]
      #print(layer_grads[-1], " -> ", grads.sum())
      params -= lr * grads


def draw_conv_filters(epoch, step, weights, save_dir, layer_name, C):
  # layer.C TODO fix al trebam si ipak iskopirat i napravit svoju verziju
  w = weights
  num_filters = w.shape[0]
  k = int(np.sqrt(w.shape[1] / C))
  print(f"reshaping to: ({num_filters}, {C}, {k}, {k})")
  w = w.reshape(num_filters, C, k, k)
  w -= w.min()
  w /= w.max()
  border = 1
  cols = 8
  rows = math.ceil(num_filters / cols)
  width = cols * k + (cols-1) * border
  height = rows * k + (rows-1) * border
  #for i in range(C):
  for i in range(1):
    img = np.zeros([height, width])
    for j in range(num_filters):
      r = int(j / cols) * (k + border)
      c = int(j % cols) * (k + border)
      img[r:r+k, c:c+k] = w[j, i]
    filename = '%s_epoch_%02d_step_%06d_input_%03d.png' % (layer_name, epoch, step, i)
    ski.io.imsave(os.path.join(save_dir, filename), img)


def train(train_x, train_y, valid_x, valid_y, net, loss, config):
  lr_policy = config['lr_policy']
  batch_size = config['batch_size']
  max_epochs = config['max_epochs']
```

```python
  save_dir = config['save_dir']
  num_examples = train_x.shape[0]
  assert num_examples % batch_size == 0
  num_batches = num_examples // batch_size
  for epoch in range(1, max_epochs+1):
    if epoch in lr_policy:
      solver_config = lr_policy[epoch]
    cnt_correct = 0
    #for i in range(num_batches):
    # shuffle the data at the beggining of each epoch
    permutation_idx = np.random.permutation(num_examples)
    train_x = train_x[permutation_idx]
    train_y = train_y[permutation_idx]
    #for i in range(100):
    for i in range(num_batches):
      # store mini-batch to ndarray
      batch_x = train_x[i*batch_size:(i+1)*batch_size, :]
      batch_y = train_y[i*batch_size:(i+1)*batch_size, :]
      logits = forward_pass(net, batch_x)
      loss_val = loss.forward(logits, batch_y)
      # compute classification accuracy
      yp = np.argmax(logits, 1)
      yt = np.argmax(batch_y, 1)
      cnt_correct += (yp == yt).sum()
      grads = backward_pass(net, loss, logits, batch_y)
      sgd_update_params(grads, solver_config)

      if i % 5 == 0:
        print("epoch %d, step %d/%d, batch loss = %.2f" % (epoch, i*batch_size, num_examples, loss_val))
      if i % 100 == 0:
        print(f"C = {net[0].C}")
        draw_conv_filters(epoch, i*batch_size, net[0].weights, save_dir, net[0].name, net[0].C)
        #draw_conv_filters(epoch, i*batch_size, net[3])
      if i > 0 and i % 50 == 0:
        print("Train accuracy = %.2f" % (cnt_correct / ((i+1)*batch_size) * 100))
    print("Train accuracy = %.2f" % (cnt_correct / num_examples * 100))
    evaluate("Validation", valid_x, valid_y, net, loss, config)
  return net


def evaluate(name, x, y, net, loss, config):
  print("\nRunning evaluation: ", name)
  batch_size = config['batch_size']
  num_examples = x.shape[0]
  assert num_examples % batch_size == 0
  num_batches = num_examples // batch_size
  cnt_correct = 0
  loss_avg = 0
  for i in range(num_batches):
    batch_x = x[i*batch_size:(i+1)*batch_size, :]
    batch_y = y[i*batch_size:(i+1)*batch_size, :]
    logits = forward_pass(net, batch_x)
    yp = np.argmax(logits, 1)
    yt = np.argmax(batch_y, 1)
    cnt_correct += (yp == yt).sum()
    loss_val = loss.forward(logits, batch_y)
    loss_avg += loss_val
    #print("step %d / %d, loss = %.2f" % (i*batch_size, num_examples, loss_val / batch_size))
  valid_acc = cnt_correct / num_examples * 100
  loss_avg /= num_batches
  print(name + " accuracy = %.2f" % valid_acc)
  print(name + " avg loss = %.2f\n" % loss_avg)
```

```
import torch
import nn
import math
from torch.utils import data
from torchvision.datasets import MNIST
from typing import Dict
import argparse
import numpy as np
from pathlib import Path
import skimage as ski
import skimage.io
import os
from sklearn import metrics
import matplotlib.pyplot as plt


class ConvollutionalModel(torch.nn.Module):

    def __init__(self):
        super(ConvollutionalModel, self).__init__()
        self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=16, kernel_size=5,
                        stride=1, padding=2, padding_mode='replicate')
        self.pool1 = torch.nn.MaxPool2d(kernel_size=2, stride=2)
        # self.relu1 = nn.ReLU
        self.conv2 = torch.nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5,
                        stride=1, padding=2, padding_mode='replicate')
        self.pool2 = torch.nn.MaxPool2d(kernel_size=2, stride=2)
        # self.relu2 = nn.ReLU
        self.flatten = torch.nn.Flatten(start_dim=1, end_dim=-1)
        self.fc1 = torch.nn.Linear(in_features=1568, out_features=512, bias=True)
        # self.relu3 = nn.ReLu
        self.fc2 = torch.nn.Linear(in_features=self.fc1.out_features, out_features=10, bias=True)

    def forward(self, x):
        s1 = self.conv1(x)
        a1 = self.pool1(s1)
        h1 = torch.relu(a1)
        s2 = self.conv2(h1)
        a2 = self.pool2(s2)
        h2 = torch.relu(a2)
        f1 = self.flatten(h2)
        s3 = self.fc1(f1)
        h3 = torch.relu(s3)
        s4 = self.fc2(h3)
        return s4


class MyDataset(data.Dataset):
    def __init__(self, x, y):
        super(MyDataset, self).__init__()
        assert len(x) == len(y), "Lengths of x and y must match."
        self.x = x
        self.y = y

    def __len__(self):
        return len(self.x)

    def __getitem__(self, idx):
        return self.x[idx], self.y[idx]


def train(train_x, train_y, valid_x, valid_y, model: torch.nn.Module,
      loss, optimizer, scheduler=None, config=dict(), callbacks=[]):

    batch_size = config.get('batch_size', 64)
```

```
    max_epochs = config.get('max_epochs', 5)
    verbose = config.get('verbose', False)
    print_frequency = config.get('print_frequency', 100)

    train_dataset = MyDataset(train_x, train_y)
    train_dataloader = data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

    for epoch in range(max_epochs):
        print(f"Epoch {epoch}")
        train_loss = 0.0
        batch_count = 0
        for batch, batch_data in enumerate(train_dataloader):
            batch_count += 1
            train_x_batch, train_y_batch = batch_data
            logits = model.forward(train_x_batch)
            batch_loss = loss(logits, train_y_batch)
            batch_loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            train_loss += batch_loss

            if verbose and batch % print_frequency == 0:
                print(f"epoch: {epoch} batch: {batch} loss: {batch_loss}")

        if scheduler is not None:
            scheduler.step()

        train_loss /= batch_count
        if len(callbacks) > 0:
            with torch.no_grad():
                logits = model.forward(valid_x)
                validation_loss = loss(logits, valid_y)
                stop_training = [cb(epoch, -1, train_loss, validation_loss, model) for cb in callbacks]
                if any(stop_training):
                    break


def evaluate(name, x, yt, model, loss):
    print(f"\nRunning evaluation: {name}")
    with torch.no_grad():
        logits = model.forward(x)
        avg_loss = loss(logits, yt)
        yp = np.argmax(logits, axis=1)
        accuracy = metrics.accuracy_score(y_true=yt, y_pred=yp)
        print(f"{name} accuracy: {accuracy}")
        print(f"{name} avg loss: {avg_loss}")


def draw_conv_filters(epoch, step, weights, save_dir, layer_name):
    w = weights
    C = w.shape[1]
    num_filters = w.shape[0]
    k = w.shape[2]
    w = w.reshape(num_filters, C, k, k)
    w -= w.min()
    w /= w.max()
    border = 1
    cols = 8
    rows = math.ceil(num_filters / cols)
    width = cols * k + (cols-1) * border
    height = rows * k + (rows-1) * border
    #for i in range(C):
    for i in range(1):
        img = np.zeros([height, width])
        for j in range(num_filters):
            r = int(j / cols) * (k + border)
```

```python
        c = int(j % cols) * (k + border)
        img[r:r+k, c:c+k] = w[j, i]
    filename = '%s_epoch_%02d_step_%06d_input_%03d.png' % (layer_name, epoch, step, i)
    # img = (img*255).astype(np.uint8)
    ski.io.imsave(os.path.join(save_dir, filename), img)


class EarlyStoppingCallback:

    def __init__(self, patience):
        self.best_validation_loss = None
        self.patience = patience
        self.count = 0

    def __call__(self, epoch, batch, train_loss, validation_loss, model=None):
        if validation_loss is None:
            return True

        validation_loss = validation_loss.detach().numpy()

        if self.best_validation_loss is None or validation_loss < self.best_validation_loss:
            self.best_validation_loss = validation_loss
            self.count = 0
            return False
        elif validation_loss > self.best_validation_loss:
            self.count += 1
            print(f"patience = {self.patience} count = {self.count} {validation_loss} > {self.best_validation_loss}"
                  f" returning {self.count >= self.patience}")
            return self.count >= self.patience


class SaveFiltersImageCallback:

    def __init__(self, save_dir):
        self.save_dir = save_dir

    def __call__(self, epoch, batch, train_loss, validation_loss, model=None):
        if batch >= 0:
            draw_conv_filters(epoch, batch, model.conv1.weight.detach().numpy(), self.save_dir, "conv1")


class LossTracerCallback:

    def __init__(self):
        self.train_loss_trace = []
        self.validation_loss_trace = []

    def __call__(self, epoch, batch, train_loss, validation_loss, model=None):
        self.validation_loss_trace.append(validation_loss.detach().numpy())
        self.train_loss_trace.append(train_loss.detach().numpy())


def dense_to_one_hot(y, class_count):
    return np.eye(class_count)[y]


def parse_arguments():
    default_data_dir = Path(__file__).parent / 'data'
    default_save_dir = Path(__file__).parent / 'out'
    parser = argparse.ArgumentParser()
    parser.add_argument("-bs", "--batch_size", help="batch size", type=int, default=64)
    parser.add_argument("-me", "--max_epochs", help="max epochs", type=int, default=10)
    parser.add_argument("-lr", "--learning_rate", help="learning rate", type=float, default=0.01)
    parser.add_argument("-g", "--gamma", help="gamma idk really", type=float, default=1 - 1e-4)
    parser.add_argument("-wd", "--weight_decay", help="L2 regularization factor", type=float, default=1e-2)
    parser.add_argument("-sd", "--save_dir", help="dir where save filters", type=str, default=default_save_dir)
```

```python
    parser.add_argument("-dd", "--data_dir", help="mnist data directory", type=str, default=default_data_dir)
    parser.add_argument("-v", "--verbose", action="store_true", default=False)
    parser.add_argument("-pf", "--print_frequency", help="not really frequency", type=int, default=100)
    parser.add_argument("-es", "--early_stopping", action="store_true", default=False)
    parser.add_argument("--patience", help="patience for early stopping", type=int, default=1)
    parser.add_argument("-nt", "--no_trace", help="don't trace loss?", action='store_true', default=False)
    args = parser.parse_args()
    return args


if __name__ == "__main__":

    args = parse_arguments()
    config = vars(args)
    # get the data
    ds_train, ds_test = MNIST(args.data_dir, train=True, download=False), MNIST(args.data_dir, train=False)
    train_x = ds_train.data.reshape([-1, 1, 28, 28]) / 255
    train_y = ds_train.targets
    train_x, valid_x = train_x[:55000], train_x[55000:]
    train_y, valid_y = train_y[:55000], train_y[55000:]
    test_x = ds_test.data.reshape([-1, 1, 28, 28]) / 255
    test_y = ds_test.targets
    train_mean = train_x.mean()
    train_x, valid_x, test_x = (x - train_mean for x in (train_x, valid_x, test_x))
    train_y_oh, valid_y_oh, test_y_oh = (dense_to_one_hot(y, 10) for y in (train_y, valid_y, test_y))
    # build the model and other stuff
    model = ConvollutionalModel()
    optimizer = torch.optim.SGD(model.parameters(), lr=args.learning_rate, weight_decay=args.weight_decay)
    scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=args.gamma)
    loss = torch.nn.CrossEntropyLoss()
    # construct callbacks
    callbacks = []
    if args.early_stopping is True:
        callbacks.append(EarlyStoppingCallback(args.patience))
    if args.save_dir is not None:
        callbacks.append(SaveFiltersImageCallback(args.save_dir))
    loss_tracer_callback = None
    if args.no_trace is not True:
        loss_tracer_callback = LossTracerCallback()
        callbacks.append(loss_tracer_callback)
    # train the model
    for k, v in config.items():
        print(f"{k}: {v}")

    train(train_x, train_y, valid_x, valid_y, model, loss, optimizer, scheduler, config, callbacks)
    evaluate("Test", test_x, test_y, model, loss)
    if loss_tracer_callback is not None:
        train_loss_trace = np.array(loss_tracer_callback.train_loss_trace)
        validation_loss_trace = np.array(loss_tracer_callback.validation_loss_trace)
        epochs = np.arange(0, len(train_loss_trace))
        plt.plot(epochs, train_loss_trace, label="train")
        plt.plot(epochs, validation_loss_trace, label="validation")
        plt.title = "Loss"
        plt.show()
```

```python
import torch
from torch.utils import data
import numpy as np
import matplotlib.pyplot as plt
import os
import pickle
from pathlib import Path
import argparse
import convolutional_model as cm
import skimage as ski
import math


def shuffle_data(data_x, data_y):
  indices = np.arange(data_x.shape[0])
  np.random.shuffle(indices)
  shuffled_data_x = np.ascontiguousarray(data_x[indices])
  shuffled_data_y = np.ascontiguousarray(data_y[indices])
  return shuffled_data_x, shuffled_data_y

def unpickle(file):
  fo = open(file, 'rb')
  dict = pickle.load(fo, encoding='latin1')
  fo.close()
  return dict


def evaluate(Y, Y_):
    pr = []
    n = max(Y_)+1
    M = np.bincount(n * Y_ + Y, minlength=n*n).reshape(n, n)
    for i in range(n):
        tp_i = M[i, i]
        fn_i = np.sum(M[i, :]) - tp_i
        fp_i = np.sum(M[:, i]) - tp_i
        tn_i = np.sum(M) - fp_i - fn_i - tp_i
        recall_i = tp_i / (tp_i + fn_i)
        precision_i = tp_i / (tp_i + fp_i)
        pr.append( (recall_i, precision_i) )

    accuracy = np.trace(M) / np.sum(M)
    return accuracy, pr, M


def draw_conv_filters(epoch, step, weights, save_dir):
    w = weights.copy()
    num_filters = w.shape[0]
    num_channels = w.shape[1]
    k = w.shape[2]
    assert w.shape[3] == w.shape[2]
    w = w.transpose(2, 3, 1, 0)
    w -= w.min()
    w /= w.max()
    border = 1
    cols = 8
    rows = math.ceil(num_filters / cols)
    width = cols * k + (cols-1) * border
    height = rows * k + (rows-1) * border
    img = np.zeros([height, width, num_channels])
    for i in range(num_filters):
        r = int(i / cols) * (k + border)
        c = int(i % cols) * (k + border)
        img[r:r+k, c:c+k, :] = w[:, :, :, i]

    img = (img * 255).astype(np.uint8)
    filename = 'epoch_%02d_step_%06d.png' % (epoch, step)
```

```python
    ski.io.imsave(os.path.join(save_dir, filename), img)


def get_lr(optimizer):
    for param_group in optimizer.param_groups:
        return param_group['lr']


def train(train_x, train_y, valid_x, valid_y, model: torch.nn.Module,
          loss, optimizer, scheduler=None, config=dict()):

    batch_size = config.get('batch_size', 64)
    max_epochs = config.get('max_epochs', 5)
    verbose = config.get('verbose', False)
    print_frequency = config.get('print_frequency', 100)

    train_dataset = cm.MyDataset(train_x, train_y)
    train_dataloader = data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    # stvari koje treba pratit
    lrs = []
    train_losses = []
    valid_losses = []
    avg_train_accuracies = []
    avg_valid_accuracies = []

    for epoch in range(max_epochs):
        print(f"Epoch {epoch}")
        for batch, batch_data in enumerate(train_dataloader):
            train_x_batch, train_y_batch = batch_data
            logits = model.forward(train_x_batch)
            batch_loss = loss(logits, train_y_batch)
            batch_loss.backward()
            optimizer.step()
            optimizer.zero_grad()

            if batch % print_frequency == 0:
                print(f"epoch: {epoch} batch: {batch} loss: {batch_loss}")
                weights = model[0].weight.detach().numpy()
                draw_conv_filters(epoch, batch_size*batch, weights, config['save_dir'])

        if scheduler is not None:
            scheduler.step()

        with torch.no_grad():
            learning_rate = get_lr(optimizer)
            lrs.append(learning_rate)

            train_loss = 0.0
            batch_count = 0
            train_y_pred = []
            for batch_x, batch_y in train_dataloader:
                batch_count += 1
                batch_logits = model.forward(batch_x)
                batch_loss = loss(batch_logits, batch_y)
                train_loss += batch_loss
                train_y_pred.append(torch.argmax(batch_logits, axis=1))

            train_loss /= batch_count
            train_y_pred = torch.hstack(train_y_pred)
            train_accuracy, pr, train_conf_matrix = evaluate(train_y, train_y_pred)
            train_losses.append(train_loss)
            avg_train_accuracies.append(train_accuracy)

            valid_logits = model.forward(valid_x)
            valid_loss = loss(valid_logits, valid_y)
            valid_y_pred = torch.argmax(valid_logits, axis=1)
```

```
            valid_accuracy, pr, valid_conf_matrix = evaluate(valid_y, valid_y_pred)
            valid_losses.append(valid_loss)
            avg_valid_accuracies.append(valid_accuracy)

    return lrs, train_losses, avg_train_accuracies, valid_losses, avg_valid_accuracies


DATA_DIR = default_data_dir = Path(__file__).parent / 'data' / 'cifar-10-batches-py'

img_height = 32
img_width = 32
num_channels = 3
num_classes = 10

train_x = np.ndarray((0, img_height * img_width * num_channels), dtype=np.float32)
train_y = []
for i in range(1, 6):
    subset = unpickle(os.path.join(DATA_DIR, 'data_batch_%d' % i))
    train_x = np.vstack((train_x, subset['data']))
    train_y += subset['labels']
train_x = train_x.reshape((-1, num_channels, img_height, img_width)).transpose(0, 2, 3, 1)
train_y = np.array(train_y, dtype=np.long)

subset = unpickle(os.path.join(DATA_DIR, 'test_batch'))
test_x = subset['data'].reshape((-1, num_channels, img_height, img_width)).transpose(0, 2, 3, 1).astype(np.float32)
test_y = np.array(subset['labels'], dtype=np.long)

valid_size = 5000
train_x, train_y = shuffle_data(train_x, train_y)
valid_x = train_x[:valid_size, ...]
valid_y = train_y[:valid_size, ...]
train_x = train_x[valid_size:, ...]
train_y = train_y[valid_size:, ...]
data_mean = train_x.mean((0, 1, 2))
data_std = train_x.std((0, 1, 2))

train_x = (train_x - data_mean) / data_std
valid_x = (valid_x - data_mean) / data_std
test_x = (test_x - data_mean) / data_std

train_x = torch.from_numpy(train_x.transpose(0, 3, 1, 2))
valid_x = torch.from_numpy(valid_x.transpose(0, 3, 1, 2))
test_x = torch.from_numpy(test_x.transpose(0, 3, 1, 2))

train_y = torch.from_numpy(train_y)
valid_y = torch.from_numpy(valid_y)
test_y = torch.from_numpy(test_y)
# =================== ARGS  stuff =================
args = cm.parse_arguments()
config = vars(args)
# =================== MODEL stuff =================
model = torch.nn.Sequential(
    torch.nn.Conv2d(in_channels=3, out_channels=16, kernel_size=5, stride=1, padding=2, padding_mode='replicate'),
    torch.nn.ReLU(),
    torch.nn.MaxPool2d(kernel_size=3, stride=2),
    torch.nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5, stride=1, padding=2, padding_mode='replicate'),
    torch.nn.ReLU(),
    torch.nn.MaxPool2d(kernel_size=3, stride=2),
    torch.nn.Flatten(start_dim=1, end_dim=-1),
    torch.nn.Linear(in_features=1568, out_features=256, bias=True),
    torch.nn.ReLU(),
    torch.nn.Linear(in_features=256, out_features=128, bias=True),
    torch.nn.ReLU(),
    torch.nn.Linear(in_features=128, out_features=10, bias=True)
)

optimizer = torch.optim.SGD(model.parameters(), lr=args.learning_rate, weight_decay=args.weight_decay)
```

```
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=args.gamma)
loss = torch.nn.CrossEntropyLoss()

results = train(train_x, train_y, valid_x, valid_y, model, loss, optimizer, scheduler, config)
lrs, train_losses, avg_train_accuracies, valid_losses, avg_valid_accuracies = results
epochs = np.arange(0, len(lrs))

fig, (ax1, ax2, ax3) = plt.subplots(3)

ax1.plot(epochs, lrs, label='learning rate')
ax1.set_title('Learning rate')
ax1.legend()

ax2.plot(epochs, train_losses, label='train')
ax2.plot(epochs, valid_losses, label='validation')
ax2.set_title('Cross-entropy loss')
ax2.legend()

ax3.plot(epochs, avg_train_accuracies, label='train')
ax3.plot(epochs, avg_valid_accuracies, label='validation')
ax3.set_title('Average class accuracy')
ax3.legend()

plt.show()
```