ДОДАТОК А
Програмна реалізація

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
from subprocess import Popen, PIPE
import tensorflow as tf
import numpy as np
from scipy import misc
from sklearn.model_selection import KFold
from scipy import interpolate
from tensorflow.python.training import training
import random
import re
from tensorflow.python.platform import gfile
import math
from six import iteritems

def triplet_loss(anchor, positive, negative, alpha):
    """Calculate the triplet loss according to the FaceNet paper

    Args:
      anchor: the embeddings for the anchor images.
      positive: the embeddings for the positive images.
      negative: the embeddings for the negative images.

    Returns:
      the triplet loss according to the FaceNet paper as a float tensor.
    """
    with tf.variable_scope('triplet_loss'):
        pos_dist = tf.reduce_sum(tf.square(tf.subtract(anchor, positive)), 1)
        neg_dist = tf.reduce_sum(tf.square(tf.subtract(anchor, negative)), 1)

        basic_loss = tf.add(tf.subtract(pos_dist,neg_dist), alpha)
        loss = tf.reduce_mean(tf.maximum(basic_loss, 0.0), 0)

    return loss
```

```
def center_loss(features, label, alfa, nrof_classes):
    """Center loss based on the paper "A Discriminative Feature Learning Approach
for Deep Face Recognition"
       (http://ydwen.github.io/papers/WenECCV16.pdf)
    """
    nrof_features = features.get_shape()[1]
    centers    =    tf.get_variable('centers',    [nrof_classes,    nrof_features],
dtype=tf.float32,
        initializer=tf.constant_initializer(0), trainable=False)
    label = tf.reshape(label, [-1])
    centers_batch = tf.gather(centers, label)
    diff = (1 - alfa) * (centers_batch - features)
    centers = tf.scatter_sub(centers, label, diff)
    with tf.control_dependencies([centers]):
        loss = tf.reduce_mean(tf.square(features - centers_batch))
    return loss, centers

def get_image_paths_and_labels(dataset):
    image_paths_flat = []
    labels_flat = []
    for i in range(len(dataset)):
        image_paths_flat += dataset[i].image_paths
        labels_flat += [i] * len(dataset[i].image_paths)
    return image_paths_flat, labels_flat

def shuffle_examples(image_paths, labels):
    shuffle_list = list(zip(image_paths, labels))
    random.shuffle(shuffle_list)
    image_paths_shuff, labels_shuff = zip(*shuffle_list)
    return image_paths_shuff, labels_shuff

def random_rotate_image(image):
    angle = np.random.uniform(low=-10.0, high=10.0)
    return misc.imrotate(image, angle, 'bicubic')

# 1: Random rotate 2: Random crop  4: Random flip  8:  Fixed image standardization
16: Flip
RANDOM_ROTATE = 1
RANDOM_CROP = 2
RANDOM_FLIP = 4
```

```
FIXED_STANDARDIZATION = 8
FLIP = 16
def   create_input_pipeline(input_queue,   image_size,   nrof_preprocess_threads,
batch_size_placeholder):
    images_and_labels_list = []
    for _ in range(nrof_preprocess_threads):
        filenames, label, control = input_queue.dequeue()
        images = []
        for filename in tf.unstack(filenames):
            file_contents = tf.read_file(filename)
            image = tf.image.decode_image(file_contents, 3)
            image = tf.cond(get_control_flag(control[0], RANDOM_ROTATE),
                            lambda:tf.py_func(random_rotate_image,        [image],
tf.uint8),
                            lambda:tf.identity(image))
            image = tf.cond(get_control_flag(control[0], RANDOM_CROP),
                            lambda:tf.random_crop(image, image_size + (3,)),
                            lambda:tf.image.resize_image_with_crop_or_pad(image,
image_size[0], image_size[1]))
            image = tf.cond(get_control_flag(control[0], RANDOM_FLIP),
                            lambda:tf.image.random_flip_left_right(image),
                            lambda:tf.identity(image))
            image = tf.cond(get_control_flag(control[0], FIXED_STANDARDIZATION),
                            lambda:(tf.cast(image, tf.float32) - 127.5)/128.0,
                            lambda:tf.image.per_image_standardization(image))
            image = tf.cond(get_control_flag(control[0], FLIP),
                            lambda:tf.image.flip_left_right(image),
                            lambda:tf.identity(image))
            #pylint: disable=no-member
            image.set_shape(image_size + (3,))
            images.append(image)
        images_and_labels_list.append([images, label])

    image_batch, label_batch = tf.train.batch_join(
        images_and_labels_list, batch_size=batch_size_placeholder,
        shapes=[image_size + (3,), ()], enqueue_many=True,
        capacity=4 * nrof_preprocess_threads * 100,
        allow_smaller_final_batch=True)

    return image_batch, label_batch
```

```
def get_control_flag(control, field):
    return tf.equal(tf.mod(tf.floor_div(control, field), 2), 1)

def _add_loss_summaries(total_loss):
    """Add summaries for losses.

    Generates moving average for all losses and associated summaries for
    visualizing the performance of the network.

    Args:
      total_loss: Total loss from loss().
    Returns:
      loss_averages_op: op for generating moving averages of losses.

    # Compute the moving average of all individual losses and the total loss.
    loss_averages = tf.train.ExponentialMovingAverage(0.9, name='avg')
    losses = tf.get_collection('losses')
    loss_averages_op = loss_averages.apply(losses + [total_loss])

    # Attach a scalar summmary to all individual losses and the total loss; do the
    # same for the averaged version of the losses.
    for l in losses + [total_loss]:
        # Name each loss as '(raw)' and name the moving average version of the
loss
        # as the original loss name.
        tf.summary.scalar(l.op.name +' (raw)', l)
        tf.summary.scalar(l.op.name, loss_averages.average(l))

    return loss_averages_op

def train(total_loss, global_step, optimizer, learning_rate, moving_average_decay,
update_gradient_vars, log_histograms=True):
    # Generate moving averages of all losses and associated summaries.
    loss_averages_op = _add_loss_summaries(total_loss)

    # Compute gradients.
    with tf.control_dependencies([loss_averages_op]):
        if optimizer=='ADAGRAD':
            opt = tf.train.AdagradOptimizer(learning_rate)
        elif optimizer=='ADADELTA':
            opt = tf.train.AdadeltaOptimizer(learning_rate, rho=0.9, epsilon=1e-6)
```

```
        elif optimizer=='ADAM':
            opt = tf.train.AdamOptimizer(learning_rate, beta1=0.9, beta2=0.999,
epsilon=0.1)
        elif optimizer=='RMSPROP':
            opt = tf.train.RMSPropOptimizer(learning_rate, decay=0.9,
momentum=0.9, epsilon=1.0)
        elif optimizer=='MOM':
            opt = tf.train.MomentumOptimizer(learning_rate, 0.9,
use_nesterov=True)
        else:
            raise ValueError('Invalid optimization algorithm')

        grads = opt.compute_gradients(total_loss, update_gradient_vars)

    # Apply gradients.
    apply_gradient_op = opt.apply_gradients(grads, global_step=global_step)

    # Add histograms for trainable variables.
    if log_histograms:
        for var in tf.trainable_variables():
            tf.summary.histogram(var.op.name, var)

    # Add histograms for gradients.
    if log_histograms:
        for grad, var in grads:
            if grad is not None:
                tf.summary.histogram(var.op.name + '/gradients', grad)

    # Track the moving averages of all trainable variables.
    variable_averages = tf.train.ExponentialMovingAverage(
        moving_average_decay, global_step)
    variables_averages_op = variable_averages.apply(tf.trainable_variables())

    with tf.control_dependencies([apply_gradient_op, variables_averages_op]):
        train_op = tf.no_op(name='train')

    return train_op

def prewhiten(x):
    mean = np.mean(x)
    std = np.std(x)
```

```
    std_adj = np.maximum(std, 1.0/np.sqrt(x.size))
    y = np.multiply(np.subtract(x, mean), 1/std_adj)
    return y

def crop(image, random_crop, image_size):
    if image.shape[1]>image_size:
        sz1 = int(image.shape[1]//2)
        sz2 = int(image_size//2)
        if random_crop:
            diff = sz1-sz2
            (h, v) = (np.random.randint(-diff, diff+1), np.random.randint(-diff,
diff+1))
        else:
            (h, v) = (0,0)
        image = image[(sz1-sz2+v):(sz1+sz2+v),(sz1-sz2+h):(sz1+sz2+h),:]
    return image

def flip(image, random_flip):
    if random_flip and np.random.choice([True, False]):
        image = np.fliplr(image)
    return image

def to_rgb(img):
    w, h = img.shape
    ret = np.empty((w, h, 3), dtype=np.uint8)
    ret[:, :, 0] = ret[:, :, 1] = ret[:, :, 2] = img
    return ret

def    load_data(image_paths,    do_random_crop,    do_random_flip,    image_size,
do_prewhiten=True):
    nrof_samples = len(image_paths)
    images = np.zeros((nrof_samples, image_size, image_size, 3))
    for i in range(nrof_samples):
        img = misc.imread(image_paths[i])
        if img.ndim == 2:
            img = to_rgb(img)
        if do_prewhiten:
            img = prewhiten(img)
        img = crop(img, do_random_crop, image_size)
        img = flip(img, do_random_flip)
        images[i,:,:,:] = img
```

```
    return images

def get_label_batch(label_data, batch_size, batch_index):
    nrof_examples = np.size(label_data, 0)
    j = batch_index*batch_size % nrof_examples
    if j+batch_size<=nrof_examples:
        batch = label_data[j:j+batch_size]
    else:
        x1 = label_data[j:nrof_examples]
        x2 = label_data[0:nrof_examples-j]
        batch = np.vstack([x1,x2])
    batch_int = batch.astype(np.int64)
    return batch_int

def get_batch(image_data, batch_size, batch_index):
    nrof_examples = np.size(image_data, 0)
    j = batch_index*batch_size % nrof_examples
    if j+batch_size<=nrof_examples:
        batch = image_data[j:j+batch_size,:,:,:]
    else:
        x1 = image_data[j:nrof_examples,:,:,:]
        x2 = image_data[0:nrof_examples-j,:,:,:]
        batch = np.vstack([x1,x2])
    batch_float = batch.astype(np.float32)
    return batch_float

def get_triplet_batch(triplets, batch_index, batch_size):
    ax, px, nx = triplets
    a = get_batch(ax, int(batch_size/3), batch_index)
    p = get_batch(px, int(batch_size/3), batch_index)
    n = get_batch(nx, int(batch_size/3), batch_index)
    batch = np.vstack([a, p, n])
    return batch

def get_learning_rate_from_file(filename, epoch):
    with open(filename, 'r') as f:
        for line in f.readlines():
            line = line.split('#', 1)[0]
            if line:
                par = line.strip().split(':')
                e = int(par[0])
```

```
                if par[1]=='-':
                    lr = -1
                else:
                    lr = float(par[1])
                if e <= epoch:
                    learning_rate = lr
                else:
                    return learning_rate

class ImageClass():
    "Stores the paths to images for a given class"
    def __init__(self, name, image_paths):
        self.name = name
        self.image_paths = image_paths

    def __str__(self):
        return self.name + ', ' + str(len(self.image_paths)) + ' images'

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

def get_dataset(path, has_class_directories=True):
    dataset = []
    path_exp = os.path.expanduser(path)
    classes = [path for path in os.listdir(path_exp) \
                    if os.path.isdir(os.path.join(path_exp, path))]
    classes.sort()
    nrof_classes = len(classes)
    for i in range(nrof_classes):
        class_name = classes[i]
        facedir = os.path.join(path_exp, class_name)
        image_paths = get_image_paths(facedir)
        dataset.append(ImageClass(class_name, image_paths))
    return dataset
def get_image_paths(facedir):
    image_paths = []
    if os.path.isdir(facedir):
        images = os.listdir(facedir)
        image_paths = [os.path.join(facedir,img) for img in images]
    return image_paths
```

ДОДАТОК Б
Відомість атестаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	86 с.
	Інші документи	
2.	Презентаційні матеріали	__ плакатів

.	.	Прізвище та ініціали.	Підп.	Дата	
Розробив		Крят В.В.			Дослідження згорткових нейронних мереж в задачі розпізнавання об'єктів на статичних зображеннях
Перевірив		Кулішова Н.Є			
Н.контр.		Малєєва І.А.			
Затв.		Філатов В.О.			

Шифр групи	Код напр./спец.
СШІм-19-1	122
ХНУРЕ кафедра ШІ	