ДОДАТОК А

Вихідний код програм

Файл sciGAN.py:

```
from __future__ import print_function, division
import argparse
import os
import numpy as np
import pandas as pd
import sys
import torchvision.transforms as transforms
from torch.autograd import Variable
import torch.nn as nn
import torch
from torch.utils.data import Dataset, DataLoader

GANs_models = opt.outdir+'/GANs_models'
if (job_name == ""):
    job_name=os.path.basename(opt.file_d)+"-"+os.path.basename(opt.file_c)
model_basename = job_name+"-"+str(opt.latent_dim)+"-"+str(opt.n_epochs)+"-
"+str(opt.ncls)
if  os.path.isdir(GANs_models)!=True:
    os.makedirs(GANs_models)

img_shape = (opt.channels, opt.img_size, opt.img_size)

cuda = True if torch.cuda.is_available() else False

class MyDataset(Dataset):

    def __init__(self, d_file, cls_file, transform=None):
        self.data = pd.read_csv(d_file,header=0,index_col=0)
        d = pd.read_csv(cls_file,header=None,index_col=False)
        self.data_cls = pd.Categorical(d.iloc[:,0]).codes
        self.transform = transform
        self.fig_h = opt.img_size ##


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

    def __getitem__(self, idx):
        data =
self.data.iloc[:,idx].values[0:(self.fig_h*self.fig_h),].reshape(self.fig_h,se
lf.fig_h,1).astype('double')  #
        label = np.array(self.data_cls[idx]).astype('int32')
        sample = {'data': data, 'label': label}
        if self.transform:
            sample = self.transform(sample)
        return sample
```

```
class ToTensor(object):

    def __call__(self, sample):
        data,label = sample['data'], sample['label']
        # swap color axis because
        # numpy image: H x W x C
        # torch image: C X H X W
        data = data.transpose((2, 0, 1))

        return {'data': torch.from_numpy(data),
                'label': torch.from_numpy(label)
                }

def one_hot(batch,depth):
    ones = torch.eye(depth)
    return ones.index_select(0,batch)

def weights_init_normal(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        torch.nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm2d') != -1:
        torch.nn.init.normal_(m.weight.data, 1.0, 0.02)
        torch.nn.init.constant_(m.bias.data, 0.0)
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.init_size = opt.img_size // 4
        self.cn1=32
        self.l1 = nn.Sequential(nn.Linear(opt.latent_dim,
self.cn1*(self.init_size**2)))
        self.l1p = nn.Sequential(nn.Linear(opt.latent_dim,
self.cn1*(opt.img_size**2)))



        self.conv_blocks_01p = nn.Sequential(
            nn.BatchNorm2d(self.cn1),
            nn.Conv2d(self.cn1, self.cn1, 3, stride=1, padding=1),
            nn.BatchNorm2d(self.cn1, 0.8),
            nn.ReLU(),
        )



        self.conv_blocks_02p = nn.Sequential(
            nn.Upsample(scale_factor=opt.img_size),#torch.Size([bs, 128, 16,
16])
            nn.Conv2d(max_ncls,  self.cn1//4, 3, stride=1,
padding=1),#torch.Size([bs, 128, 16, 16])
            nn.BatchNorm2d( self.cn1//4),
            nn.ReLU(),
        )


        self.conv_blocks_1 = nn.Sequential(
```

```
            nn.BatchNorm2d(40, 0.8),
            nn.Conv2d(40, self.cn1, 3, stride=1, padding=1),
nn.BatchNorm2d(self.cn1),
            nn.ReLU(),
            nn.Conv2d(self.cn1, opt.channels, 3, stride=1,
padding=1),#torch.Size([bs, 1, 32, 32])
            nn.Sigmoid()
        )
    def forward(self, noise,label_oh):
        out = self.l1p(noise)
        out = out.view(out.shape[0], self.cn1, opt.img_size, opt.img_size)
        out01 = self.conv_blocks_01p(out) #([4, 32, 124, 124])
#
        label_oh=label_oh.unsqueeze(2)
        label_oh=label_oh.unsqueeze(2)
        out02 = self.conv_blocks_02p(label_oh) #([4, 8, 124, 124])


        out1=torch.cat((out01,out02),1)
        out1=self.conv_blocks_1(out1)
        return out1


class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.cn1=32
        self.down_size0 = 64
        self.down_size = 32
        self.pre= nn.Sequential(
            nn.Linear(opt.img_size**2,self.down_size0**2),
        )

        # Upsampling
        self.down = nn.Sequential(
            nn.Conv2d(opt.channels, self.cn1, 3, 1, 1),
            nn.MaxPool2d(2,2),
            nn.BatchNorm2d(self.cn1),
            nn.ReLU(),
            nn.Conv2d(self.cn1, self.cn1//2, 3, 1, 1),
            nn.BatchNorm2d(self.cn1//2),
            nn.ReLU(),
        )


        self.conv_blocks02p = nn.Sequential(
            nn.Upsample(scale_factor=self.down_size),
            nn.Conv2d(max_ncls,  self.cn1//4, 3, stride=1, padding=1),
            nn.BatchNorm2d( self.cn1//4),
            nn.ReLU(),
        )

        # Fully-connected layers

        down_dim =24 * (self.down_size)**2
        self.fc = nn.Sequential(
            nn.Linear(down_dim, 16),
```

```
            nn.BatchNorm1d(16, 0.8),
            nn.ReLU(),
            nn.Linear(16, down_dim),
            nn.BatchNorm1d(down_dim),
            nn.ReLU()
        )
        # Upsampling 32X32
        self.up = nn.Sequential(
            nn.Upsample(scale_factor=4),
            nn.Conv2d(24, 16, 3, 1, 1),
            nn.MaxPool2d(2,2),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.Conv2d(16, 8, 3, 1, 1),
            nn.MaxPool2d(2,2),
            nn.BatchNorm2d(8),
            nn.ReLU(),
            nn.Conv2d(8, 4, 3, 1, 1),
            nn.MaxPool2d(2,2),
            nn.BatchNorm2d(4),
            nn.ReLU(),
            nn.Conv2d(4, 4, 3, 1, 1),
            nn.MaxPool2d(2,2),
            nn.BatchNorm2d(4),
            nn.ReLU(),
            nn.Conv2d(4, 4, 3, 1, 1),
            nn.MaxPool2d(2,2),
            nn.BatchNorm2d(4),
            nn.ReLU(),
            nn.Conv2d(4, 4, 3, 1, 1),
            nn.MaxPool2d(2,2),
            nn.BatchNorm2d(4),
            nn.ReLU(),
            nn.Conv2d(4, opt.channels, 2, 1, 0),
            nn.Sigmoid(),
        )

    def forward(self, img,label_oh):

        out00 = self.pre(img.view((img.size()[0],-
1))).view((img.size()[0],1,self.down_size0,self.down_size0))
        out01 = self.down(out00)#([4, 16, 32, 32])

        label_oh=label_oh.unsqueeze(2)
        label_oh=label_oh.unsqueeze(2)
        out02 = self.conv_blocks02p(label_oh)#([4, 16, 32, 32])
        out1=torch.cat((out01,out02),1)
        out = self.fc(out1.view(out1.size(0), -1))
        out = self.up(out.view(out.size(0), 24, self.down_size,
self.down_size))
        return out


#%%
def my_knn_type(data_imp_org_k,sim_out_k,knn_k=10):
        sim_size=sim_out_k.shape[0]
        out=data_imp_org_k.copy()
        q1k = data_imp_org_k.reshape((opt.img_size*opt.img_size,1))
```

```
        q1kl = np.int8(q1k>0) # get which part in cell k is >0
        q1kn = np.repeat(q1k*q1kl,repeats=sim_size,axis=1) # get >0 part of
cell k
        sim_out_tmp=sim_out_k.reshape((sim_size,opt.img_size*opt.img_size)).T
        sim_outn = sim_out_tmp * np.repeat(q1kl,repeats=sim_size,axis=1)
        diff = q1kn-sim_outn #distance of cell k to simmed ones
        diff = diff*diff
        rel = np.sum(diff,axis=0)
        locs = np.where(q1kl==0)[0]
        sim_out_c=np.median(sim_out_tmp[:,rel.argsort()[0:knn_k]],axis=1)
        out[locs]=sim_out_c[locs]
        return out


# Initialize generator and discriminator
generator = Generator()
discriminator = Discriminator()


print(discriminator)
if cuda:
    generator.cuda()
    discriminator.cuda()
    print("scIGANs is runing on GPUs.")
else:
    print("scIGANs is runing on CPUs.")
# Initialize weights
generator.apply(weights_init_normal)
discriminator.apply(weights_init_normal)


# Configure data loader
transformed_dataset = MyDataset(d_file=opt.file_d,
                                cls_file=opt.file_c,

transform=transforms.Compose([ToTensor()]))
dataloader = DataLoader(transformed_dataset, batch_size=opt.batch_size,
                        shuffle=True, num_workers=0,drop_last=True)

# Optimizers
optimizer_G = torch.optim.Adam(generator.parameters(), lr=opt.lr,
betas=(opt.b1, opt.b2))
optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=opt.lr,
betas=(opt.b1, opt.b2))

Tensor = torch.cuda.FloatTensor if cuda else torch.FloatTensor
```

Файл SAVER-X.R:

```
autoencode <- function(x,
                       python.module,
                       main,
                       test.x = NULL,
                       pretrain_file = "",
                       nonmissing_indicator = 1,
                       n_human=21183L,
                       n_mouse=21122L,
```

```
                      shared_size=15494L,
                      model.species = NULL,
                      out_dir = ".",
                      batch_size = 32L,
                      write_output_to_tsv = F,
                      ...) {

 if (pretrain_file == "")
   pretrain <- F
 else
   pretrain <- T

 if (pretrain)
   api <- python.module$api_pretrain
 else
   api <- python.module$api


 gnames <- rownames(x)
 cnames <- colnames(x)
 x <- Matrix::Matrix(x, sparse = T)
 mtx_file <- paste0(out_dir, "/SAVERX_temp.mtx")
 Matrix::writeMM(x, file = mtx_file)
 rm(x)
# x <- api$anndata$AnnData(t(x))
# main$x <- x
 nonmissing_indicator <- api$np$asarray(nonmissing_indicator)
 gc()

 if (!is.null(test.x)) {
   gnames <- rownames(test.x)
   cnames <- colnames(test.x)
   test.x <- Matrix::Matrix(test.x, sparse = T)
   test_mtx_file <- paste0(out_dir, "/SAVERX_temp_test.mtx")
   Matrix::writeMM(test.x, file = test_mtx_file)
   rm(test.x)
   gc()
 } else
   test_mtx_file <- NULL


 if (!pretrain)
   main$result <- api$autoencode(mtx_file = mtx_file,
                                 pred_mtx_file = test_mtx_file,
                                 nonmissing_indicator = nonmissing_indicator,
                                 out_dir = out_dir,
                                 batch_size = batch_size,
                                 write_output_to_tsv = write_output_to_tsv,
                                 ...)
 else
   main$result <- api$autoencode(n_inoutnodes_human=n_human,
                                 n_inoutnodes_mouse=n_mouse,
                                 shared_size=shared_size,
                                # adata = x,
                                 mtx_file = mtx_file,
                                 pred_mtx_file = test_mtx_file,
                                 species=model.species,
                                 nonmissing_indicator = nonmissing_indicator,
```

```
                                    initial_file = pretrain_file,
                                    out_dir = out_dir,
                                    batch_size = batch_size,
                                    write_output_to_tsv = write_output_to_tsv,
                                    ...)

  if (!write_output_to_tsv) {
    x.autoencoder <- t(reticulate::py_to_r(main$result$obsm[['X_dca']]))
    colnames(x.autoencoder) <- cnames
    rownames(x.autoencoder) <- gnames
  } else
    x.autoencoder <- NULL
  reticulate::py_run_string("
del result
import gc
gc.collect()")
  return(x.autoencoder)
}

computePrediction <- function(out.dir,
                              input.file.name = NULL,
                              data.matrix = NULL,
                              data.species = c("Human", "Mouse", "Others"),
                              use.pretrain = F,
                              pretrained.weights.file = "",
                              model.species = c("Human", "Mouse", "Joint"),
                              model.nodes.ID = NULL,
                              is.large.data = F,
                              clearup.python.session = T,
                              batch_size = NULL,
                              ...) {

  if (!is.null(input.file.name)) {
    format <- strsplit(input.file.name, '[.]')[[1]]
    format <- paste(".", format[length(format)], sep = "")
    if (format != ".txt" && format != ".csv" && format != ".rds")
      stop("Input file must be in .txt or .csv or .rds form", call.=FALSE)
    print(paste("Input file is:", input.file.name))
  } else
    print("Input is a data matrix")
  if (use.pretrain)
    temp <- "Yes"
  else
    temp <- "No"
  print(paste("Use a pretrained model:", temp))

   data.species <- match.arg(data.species, c("Human", "Mouse", "Others"))
  if (use.pretrain)
    print(paste("Data species is:", data.species))

   if (use.pretrain) {
      if (data.species == "Others")
         stop("For pretrained model, the data.species can only be Human or
Mouse")
      if (!file.exists(pretrained.weights.file))
         stop("Can not find the pretrained model. Please make sure that
pretrained.weights.file exists")
      print(paste("Pretrained weights file is:", pretrained.weights.file))
```

```
      model.species <- match.arg(model.species, c("Human", "Mouse", "Joint"))
    if (model.species == "Joint")
      model.species <- data.species
      print(paste("Model species is:", model.species))

  }
  ######

  dir.create(out.dir, showWarnings = F)

  ### preprocess data  ###
  if (use.pretrain && is.null(model.nodes.ID)) {
     if (model.species == "Human") {
  #     data(human_nodes_ID)
        model.nodes.ID <- human_nodes_ID
     } else {
  #     data(mouse_nodes_ID)
        model.nodes.ID <- mouse_nodes_ID
     }
  }
  preprocessDat(out.dir, input.file.name,
         data.matrix,
             data.species = data.species,
             model.species = model.species,
             model.nodes.ID = model.nodes.ID)



  print("Data preprocessed ...")

  sctransfer <- reticulate::import("sctransfer", convert = F)
  main <- reticulate::import_main(convert = F)
  print("Python module sctransfer imported ...")

  if (is.large.data)
    write_output_to_tsv <- T
  else
    write_output_to_tsv <- F

  if (use.pretrain) {

    data <- readRDS(paste0(out.dir, "/tmpdata.rds"))

    est.mu <- Matrix::rowMeans(Matrix::t(Matrix::t(data$mat) /
Matrix::colSums(data$mat)) * 10000)

    n.genes <- nrow(data$mat)
    err.autoencoder <- rep(NA, n.genes)
    err.const <- rep(NA, n.genes)
    n.cells <- ncol(data$mat)

    if (data.species == model.species)
      ID.use <- data$rowdata$internal_ID
    else
      ID.use <- data$rowdata$other_species_internal_ID
    rm(data)
    gc()
```

```
      x <- Matrix::readMM(paste0(out.dir, "/tmpdata.mtx"))
    if (is.null(batch_size))
      batch_size <- as.integer(max(ncol(x) / 50, 32))
    else
      batch_size <- as.integer(batch_size)

      nonmissing_indicator <- read.table(paste0(out.dir,
"/tmpdata_nonmissing.txt"))$V1
    used.time <- system.time(result <- autoFilterCV(x,
                                                    sctransfer,
                                                    main,
                                                    pretrain_file =
pretrained.weights.file,
                                                    nonmissing_indicator =
nonmissing_indicator,
                                                    model.species =
model.species,
                                                    out_dir = out.dir,
                                                    batch_size = batch_size,
                                                    write_output_to_tsv =
write_output_to_tsv,
                                                    ...))
      print(paste("Autoencoder total computing time is:", used.time[3],
"seconds"))

      idx <- nonmissing_indicator == 1
      print(paste("Number of predictive genes is", sum(result$err.const[idx] >
result$err.autoencoder[idx])))


      temp <- table(ID.use)
      id.not.unique <- names(temp[temp > 1])

      rownames(result$x.autoencoder) <- model.nodes.ID

      names(result$err.autoencoder) <- names(result$err.const) <-
rownames(result$x.autoencoder)

      idx <- ID.use %in% model.nodes.ID[nonmissing_indicator == 1]
    result$x.autoencoder <- result$x.autoencoder[ID.use[idx], ]
    rownames(result$x.autoencoder) <- names(est.mu)[idx]
    tt <- est.mu[!idx] %*% t(rep(1, n.cells))
    rownames(tt) <- names(est.mu)[!idx]
    result$x.autoencoder <- rbind(result$x.autoencoder, tt)
    rm(tt)
    gc()
    result$x.autoencoder <- result$x.autoencoder[names(est.mu), , drop = F]

#     est.mu[idx, ] <- result$x.autoencoder[ID.use[idx], ]
      err.autoencoder[idx] <- result$err.autoencoder[ID.use[idx]]
    err.const[idx] <- result$err.const[ID.use[idx]]
      tmp <- suppressWarnings(file.remove(paste0(out.dir,
"/tmpdata_nonmissing.txt")))
      tmp <- suppressWarnings(file.remove(paste0(out.dir, "/tmpdata.mtx")))
 #   result$x.autoencoder <- est.mu
    result$err.autoencoder <- err.autoencoder
      result$err.const <- err.const
   } else {
```

```
      data <- readRDS(paste0(out.dir, "/tmpdata.rds"))
    if (is.null(batch_size))
      batch_size <- as.integer(max(ncol(data$mat) / 50, 32))
    else
      batch_size <- as.integer(batch_size)

      used.time <- system.time(result <- autoFilterCV(data$mat,
                                    sctransfer,
                                    main,
                                    out_dir = out.dir,
                           batch_size = batch_size,
                           write_output_to_tsv = write_output_to_tsv,
                                    ...))
    rm(data)
    gc()

      print(paste("Autoencoder total computing time is:", used.time[3],
"seconds"))

      print(paste("Number of predictive genes is", sum(result$err.const >
result$err.autoencoder)))
   }

  if (clearup.python.session) {
    reticulate::py_run_string("
import sys
sys.modules[__name__].__dict__.clear()")
    print("Python module cleared up.")
  }


  if (!use.pretrain || data.species == model.species) {
    temp.name <- paste0(out.dir, "/prediction.rds")
      saveRDS(result, file = temp.name)
    print(paste("Predicted + filtered results saved as:", temp.name))
   } else {
    temp.name <- paste0(out.dir, "/other_species_prediction.rds")
      saveRDS(result, file = temp.name)
    print(paste("Predicted + filtered results saved as:", temp.name))
   }
  tmp <- suppressWarnings(file.remove(paste0(out.dir, "/SAVERX_temp.mtx")))
  tmp <- suppressWarnings(file.remove(paste0(out.dir,
"/SAVERX_temp_test.mtx")))
  if (is.large.data) {
    tmp <- suppressWarnings(file.remove(paste0(out.dir,
"/SAVERX_temp_mean_norm.tsv")))
    tmp <- suppressWarnings(file.remove(paste0(out.dir,
"/SAVERX_temp_pred_mean_norm.tsv")))
  }
   if (!use.pretrain)
      tmp <- suppressWarnings(file.remove(paste0(out.dir,
"/SAVERX_temp_dispersion.tsv")))
}
```

Файл ResNets.py:

```
import os.path
```

```
import keras.optimizers
from keras.layers import Input, Dense, merge, Activation, add
from keras.models import Model
from keras import callbacks as cb
import numpy as np
import matplotlib
from keras.layers.normalization import BatchNormalization
import os
from keras.regularizers import l2
from sklearn import decomposition
from keras.callbacks import LearningRateScheduler
import math
import ScatterHist as sh
from keras import initializers
from numpy import genfromtxt
import sklearn.preprocessing as prep
import tensorflow as tf
import keras.backend as K

def squaredDistance(X, Y):
    r = K.expand_dims(X, axis=1)
    return K.sum(K.square(r - Y), axis=-1)


class MMD:
    MMDTargetTrain = None
    MMDTargetTrainSize = None
    MMDTargetValidation = None
    MMDTargetValidationSize = None
    MMDTargetSampleSize = None
    kernel = None
    scales = None
    weights = None

    def __init__(self,
                 MMDLayer,
                 MMDTargetTrain,
                 MMDTargetValidation_split=0.1,
                 MMDTargetSampleSize=1000,
                 n_neighbors=25,
                 scales=None,
                 weights=None):
        if scales == None:
            print("setting scales using KNN")
            med = np.zeros(20)
            for ii in range(1, 20):
                sample =
MMDTargetTrain[np.random.randint(MMDTargetTrain.shape[0],
size=MMDTargetSampleSize), :]
                nbrs = NearestNeighbors(n_neighbors=n_neighbors).fit(sample)
                distances, dummy = nbrs.kneighbors(sample)
                med[ii] = np.median(distances[:, 1:n_neighbors])
            med = np.median(med)
            scales = [med / 2, med, med * 2]  # CyTOF
            print(scales)
        scales = K.variable(value=np.asarray(scales))
        if weights == None:
            print("setting all scale weights to 1")
```

```
            weights = K.eval(K.shape(scales)[0])
        weights = K.variable(value=np.asarray(weights))
        self.MMDLayer = MMDLayer
        MMDTargetTrain, MMDTargetValidation = train_test_split(MMDTargetTrain,
test_size=MMDTargetValidation_split,

random_state=42)
        self.MMDTargetTrain = K.variable(value=MMDTargetTrain)
        self.MMDTargetTrainSize = K.eval(K.shape(self.MMDTargetTrain)[0])
        self.MMDTargetValidation = K.variable(value=MMDTargetValidation)
        self.MMDTargetValidationSize =
K.eval(K.shape(self.MMDTargetValidation)[0])
        self.MMDTargetSampleSize = MMDTargetSampleSize
        self.kernel = self.RaphyKernel
        self.scales = scales
        self.weights = weights

    def RaphyKernel(self, X, Y):
        sQdist = K.expand_dims(squaredDistance(X, Y), 0)
        self.scales = K.expand_dims(K.expand_dims(self.scales, -1), -1)
        self.weights = K.expand_dims(K.expand_dims(self.weights, -1), -1)
        return K.sum(self.weights * K.exp(-sQdist / (K.pow(self.scales, 2))),
0)

    def cost(self, source, target):
        xx = self.kernel(source, source)
        xy = self.kernel(source, target)
        yy = self.kernel(target, target)
        MMD = K.mean(xx) - 2 * K.mean(xy) + K.mean(yy)
        return K.sqrt(MMD);

    def KerasCost(self, y_true, y_pred):
        sample =
K.cast(K.round(K.random_uniform_variable(shape=tuple([self.MMDTargetSampleSize
]), low=0,

high=self.MMDTargetTrainSize - 1)), IntType)
        MMDTargetSampleTrain = K.gather(self.MMDTargetTrain, sample)
        sample =
K.cast(K.round(K.random_uniform_variable(shape=tuple([self.MMDTargetSampleSize
]), low=0,

high=self.MMDTargetValidationSize - 1)), IntType)
        MMDTargetSampleValidation = K.gather(self.MMDTargetValidation, sample)
        MMDtargetSample = K.in_train_phase(MMDTargetSampleTrain,
MMDTargetSampleValidation)
        ret = self.cost(self.MMDLayer, MMDtargetSample)
        ret = ret + 0 * K.sum(y_pred) + 0 * K.sum(y_true)
        return ret


source = genfromtxt(sourcePath, delimiter=',', skip_header=0)
target = genfromtxt(targetPath, delimiter=',', skip_header=0)

target = dh.preProcessCytofData(target)
source = dh.preProcessCytofData(source)

numZerosOK=1
```

```
toKeepS = np.sum((source==0), axis = 1) <=numZerosOK
print(np.sum(toKeepS))
toKeepT = np.sum((target==0), axis = 1) <=numZerosOK
print(np.sum(toKeepT))

inputDim = target.shape[1]

if denoise:
    trainTarget_ae = np.concatenate([source[toKeepS], target[toKeepT]],
axis=0)
    np.random.shuffle(trainTarget_ae)
    trainData_ae = trainTarget_ae * np.random.binomial(n=1, p=keepProb, size =
trainTarget_ae.shape)
    input_cell = Input(shape=(inputDim,))
    encoded = Dense(ae_encodingDim,
activation='relu',W_regularizer=l2(l2_penalty_ae))(input_cell)
    encoded1 = Dense(ae_encodingDim,
activation='relu',W_regularizer=l2(l2_penalty_ae))(encoded)
    decoded = Dense(inputDim,
activation='linear',W_regularizer=l2(l2_penalty_ae))(encoded1)
    autoencoder = Model(input=input_cell, output=decoded)
    autoencoder.compile(optimizer='rmsprop', loss='mse')
    autoencoder.fit(trainData_ae, trainTarget_ae, epochs=500, batch_size=128,
shuffle=True,  validation_split=0.1,
                    callbacks=[mn.monitor(),
cb.EarlyStopping(monitor='val_loss', patience=25,  mode='auto')])
    source = autoencoder.predict(source)
    target = autoencoder.predict(target)

preprocessor = prep.StandardScaler().fit(source)
source = preprocessor.transform(source)
target = preprocessor.transform(target)

calibInput = Input(shape=(inputDim,))
block1_bn1 = BatchNormalization()(calibInput)
block1_a1 = Activation('relu')(block1_bn1)
block1_w1 = Dense(mmdNetLayerSizes[0],
activation='linear',kernel_regularizer=l2(l2_penalty),
                  kernel_initializer=initializers.RandomNormal(stddev=1e-
4))(block1_a1)
block1_bn2 = BatchNormalization()(block1_w1)
block1_a2 = Activation('relu')(block1_bn2)
block1_w2 = Dense(inputDim,
activation='linear',kernel_regularizer=l2(l2_penalty),
                  kernel_initializer=initializers.RandomNormal(stddev=1e-
4))(block1_a2)
block1_output = add([block1_w2, calibInput])
block2_bn1 = BatchNormalization()(block1_output)
block2_a1 = Activation('relu')(block2_bn1)
block2_w1 = Dense(mmdNetLayerSizes[1],
activation='linear',kernel_regularizer=l2(l2_penalty),
                  kernel_initializer=initializers.RandomNormal(stddev=1e-
4))(block2_a1)
block2_bn2 = BatchNormalization()(block2_w1)
block2_a2 = Activation('relu')(block2_bn2)
block2_w2 = Dense(inputDim,
activation='linear',kernel_regularizer=l2(l2_penalty),
                  kernel_initializer=initializers.RandomNormal(stddev=1e-
```

```
4))(block2_a2)
block2_output = add([block2_w2, block1_output])
block3_bn1 = BatchNormalization()(block2_output)
block3_a1 = Activation('relu')(block3_bn1)
block3_w1 = Dense(mmdNetLayerSizes[1],
activation='linear',kernel_regularizer=l2(l2_penalty),
                  kernel_initializer=initializers.RandomNormal(stddev=1e-
4))(block3_a1)
block3_bn2 = BatchNormalization()(block3_w1)
block3_a2 = Activation('relu')(block3_bn2)
block3_w2 = Dense(inputDim,
activation='linear',kernel_regularizer=l2(l2_penalty),
                  kernel_initializer=initializers.RandomNormal(stddev=1e-
4))(block3_a2)
block3_output = add([block3_w2, block2_output])

calibMMDNet = Model(inputs=calibInput, outputs=block3_output)

def step_decay(epoch):
    initial_lrate = 0.001
    drop = 0.1
    epochs_drop = 150.0
    lrate = initial_lrate * math.pow(drop, math.floor((1+epoch)/epochs_drop))
    return lrate
lrate = LearningRateScheduler(step_decay)

optimizer = keras.optimizers.rmsprop(lr=0.0)

calibMMDNet.compile(optimizer=optimizer, loss=lambda y_true,y_pred:

cf.MMD(block3_output,target,MMDTargetValidation_split=0.1).KerasCost(y_true,y_
pred))
K.get_session().run(tf.global_variables_initializer())

sourceLabels = np.zeros(source.shape[0])
calibMMDNet.fit(source,sourceLabels,nb_epoch=500,batch_size=1000,validation_sp
lit=0.1,verbose=1,
           callbacks=[lrate, mn.monitorMMD(source, target,
calibMMDNet.predict),

cb.EarlyStopping(monitor='val_loss',patience=50,mode='auto')])

calibratedSource = calibMMDNet.predict(source)

pca = decomposition.PCA()
pca.fit(target)

target_sample_pca = pca.transform(target)
projection_before = pca.transform(source)
projection_after = pca.transform(calibratedSource)
```

Файл scvis.py:

```
class GaussianVAE(MLP):
    def __init__(self, input_data, input_size,
                 layer_size=LAYER_SIZE,
                 output_dim=OUTPUT_DIM,
```

```
                 decoder_layer_size=LAYER_SIZE[::-1]):
        super(self.__class__, self).__init__(input_data, input_size,
                                             layer_size, output_dim)

        self.num_encoder_layer = len(self.layer_size)

        with tf.name_scope('encoder-mu'):
            self.bias_mu = self.init_b([self.output_dim])
            self.weights_mu = self.init_w([self.layer_size[-1],
self.output_dim])

        with tf.name_scope('encoder-sigma'):
            self.bias_sigma_square = self.init_b([self.output_dim])
            self.weights_sigma_square = self.init_w([self.layer_size[-1],
self.output_dim])

        with tf.name_scope('encoder-parameter'):
            self.encoder_parameter = self.encoder()

        with tf.name_scope('sample'):
            self.ep = tf.random_normal(
                [self.input_size, self.output_dim],
                mean=0, stddev=1, name='epsilon_univariate_norm')

            self.z = tf.add(self.encoder_parameter.mu,
                            tf.sqrt(self.encoder_parameter.sigma_square) *
self.ep,
                            name='latent_z')

        self.decoder_layer_size = decoder_layer_size
        self.num_decoder_layer = len(self.decoder_layer_size)

        with tf.name_scope('decoder'):
            self.weights.append(self.init_w([self.output_dim,
self.decoder_layer_size[0]]))
            self.biases.append(self.init_b([self.decoder_layer_size[0]]))

            self.decoder_hidden_layer_out = self.activate(
                tf.matmul(self.z, self.weights[-1]) +
                self.biases[-1])

            for in_dim, out_dim in \
                    zip(self.decoder_layer_size, self.decoder_layer_size[1:]):
                self.weights.append(self.init_w([in_dim, out_dim]))
                self.biases.append(self.init_b([out_dim]))

                self.decoder_hidden_layer_out = self.activate(
                    tf.matmul(self.decoder_hidden_layer_out, self.weights[-1])
+
                    self.biases[-1])

            self.decoder_bias_mu = self.init_b([self.input_dim])
            self.decoder_weights_mu = \
                self.init_w([self.decoder_layer_size[-1],
                             self.input_dim])

            self.decoder_bias_sigma_square = self.init_b([self.input_dim])
            self.decoder_weights_sigma_square = \
```

```
                self.init_w([self.decoder_layer_size[-1],
                             self.input_dim])

            mu = tf.add(tf.matmul(self.decoder_hidden_layer_out,
                                  self.decoder_weights_mu),
                        self.decoder_bias_mu)
            sigma_square = tf.add(tf.matmul(self.decoder_hidden_layer_out,

self.decoder_weights_sigma_square),
                                  self.decoder_bias_sigma_square)

            self.decoder_parameter = \
                LocationScale(mu,
tf.clip_by_value(tf.nn.softplus(sigma_square),
                                                      EPS, MAX_SIGMA_SQUARE))

    def decoder(self, z):
        hidden_layer_out = self.activate(
            tf.matmul(z, self.weights[self.num_encoder_layer]) +
            self.biases[self.num_encoder_layer]
        )

        for layer in range(self.num_encoder_layer+1,
                           self.num_encoder_layer + self.num_decoder_layer):
            hidden_layer_out = self.activate(
                tf.matmul(hidden_layer_out, self.weights[layer]) +
                self.biases[layer])

        mu = tf.add(tf.matmul(hidden_layer_out, self.decoder_weights_mu),
                    self.decoder_bias_mu)
        sigma_square = tf.add(tf.matmul(hidden_layer_out,
                                        self.decoder_weights_sigma_square),
                              self.decoder_bias_sigma_square)

        return LocationScale(mu,
tf.clip_by_value(tf.nn.softplus(sigma_square),
                                                    EPS, MAX_SIGMA_SQUARE))

    def encoder(self, prob=0.9):
        weights_mu = tf.nn.dropout(self.weights_mu, prob)
        mu = tf.add(tf.matmul(self.hidden_layer_out, weights_mu),
                    self.bias_mu)
        sigma_square = tf.add(tf.matmul(self.hidden_layer_out,
                                        self.weights_sigma_square),
                              self.bias_sigma_square)

        return LocationScale(mu,
                             tf.clip_by_value(tf.nn.softplus(sigma_square),
                                              EPS, MAX_SIGMA_SQUARE))
class SCVIS(object):
    def __init__(self, architecture, hyperparameter):
        self.eps = 1e-20

        tf.reset_default_graph()
        self.sess = tf.InteractiveSession()
        self.normalizer = tf.Variable(1.0, name='normalizer', trainable=False)

        self.architecture, self.hyperparameter = architecture, hyperparameter
```

```
        self.regularizer_l2 = self.hyperparameter['regularizer_l2']
        self.n = self.hyperparameter['batch_size']
        self.perplexity = self.hyperparameter['perplexity']

        tf.set_random_seed(self.hyperparameter['seed'])

        # Place_holders
        self.batch_size = tf.placeholder(dtype=tf.int32)
        self.x = tf.placeholder(tf.float32, shape=[None,
self.architecture['input_dimension']])
        self.z = tf.placeholder(tf.float32, shape=[None,
self.architecture['latent_dimension']])

        self.p = tf.placeholder(tf.float32, shape=[None, None])
        self.iter = tf.placeholder(dtype=tf.float32)

        self.vae = GaussianVAE(self.x,
                               self.batch_size,
                               self.architecture['inference']['layer_size'],
                               self.architecture['latent_dimension'],

decoder_layer_size=self.architecture['model']['layer_size'])

        self.encoder_parameter = self.vae.encoder_parameter
        self.latent = dict()
        self.latent['mu'] = self.encoder_parameter.mu
        self.latent['sigma_square'] = self.encoder_parameter.sigma_square
        self.latent['sigma'] = tf.sqrt(self.latent['sigma_square'])

        self.decoder_parameter = self.vae.decoder_parameter
        self.dof = tf.Variable(tf.constant(1.0,
shape=[self.architecture['input_dimension']]),
                               trainable=True, name='dof')
        self.dof = tf.clip_by_value(self.dof, 0.1, 10, name='dof')

        with tf.name_scope('ELBO'):
            self.weight = tf.clip_by_value(tf.reduce_sum(self.p, 0), 0.01,
2.0)

            self.log_likelihood = tf.reduce_mean(tf.multiply(
                log_likelihood_student(self.x,
                                       self.decoder_parameter.mu,
                                       self.decoder_parameter.sigma_square,
                                       self.dof),
                self.weight), name="log_likelihood")

            self.kl_divergence = \
                tf.reduce_mean(0.5 * tf.reduce_sum(self.latent['mu'] ** 2 +
                                                   self.latent['sigma_square']
-

tf.log(self.latent['sigma_square']) - 1,
                                                   reduction_indices=1))
            self.kl_divergence *= tf.maximum(0.1,
self.architecture['input_dimension']/self.iter)
            self.elbo = self.log_likelihood - self.kl_divergence

        self.z_batch = self.vae.z
```

```
        with tf.name_scope('tsne'):
            self.kl_pq = self.tsne_repel() * tf.minimum(self.iter,
self.architecture['input_dimension'])

        with tf.name_scope('objective'):
            self.obj = self.kl_pq + self.regularizer() - self.elbo

        # Optimization
        with tf.name_scope('optimizer'):
            learning_rate =
self.hyperparameter['optimization']['learning_rate']

            if self.hyperparameter['optimization']['method'].lower() ==
'adagrad':
                self.optimizer = tf.train.AdagradOptimizer(learning_rate)
            elif self.hyperparameter['optimization']['method'].lower() ==
'adam':
                self.optimizer = tf.train.AdamOptimizer(learning_rate,
                                                        beta1=0.9,
                                                        beta2=0.999,
                                                        epsilon=0.001)

            gradient_clipped = self.clip_gradient()

            self.train_op = self.optimizer.apply_gradients(gradient_clipped,
name='minimize_cost')

        self.saver = tf.train.Saver()

    def clip_gradient(self, clip_value=3.0, clip_norm=10.0):
        trainable_variable =
self.sess.graph.get_collection('trainable_variables')
        grad_and_var = self.optimizer.compute_gradients(self.obj,
trainable_variable)

        grad_and_var = [(grad, var) for grad, var in grad_and_var if grad is
not None]
        grad, var = zip(*grad_and_var)
        grad, global_grad_norm = tf.clip_by_global_norm(grad,
clip_norm=clip_norm)

        grad_clipped_and_var = [(tf.clip_by_value(grad[i], -clip_value*0.1,
clip_value*0.1), var[i])
                                if 'encoder-sigma' in var[i].name
                                else (tf.clip_by_value(grad[i], -clip_value,
clip_value), var[i])
                                for i in range(len(grad_and_var))]

        return grad_clipped_and_var

    def regularizer(self):
        penalty = [tf.nn.l2_loss(var) for var in
                   self.sess.graph.get_collection('trainable_variables')
                   if 'weight' in var.name]

        l2_regularizer = self.regularizer_l2 * tf.add_n(penalty)
```

```
        return l2_regularizer

    def tsne_repel(self):
        nu = tf.constant(self.architecture['latent_dimension'] - 1,
dtype=tf.float32)

        sum_y = tf.reduce_sum(tf.square(self.z_batch), reduction_indices=1)
        num = -2.0 * tf.matmul(self.z_batch,
                               self.z_batch,
                               transpose_b=True) + tf.reshape(sum_y, [-1, 1])
+ sum_y
        num = num / nu

        p = self.p + 0.1 / self.n
        p = p / tf.expand_dims(tf.reduce_sum(p, reduction_indices=1), 1)

        num = tf.pow(1.0 + num, -(nu + 1.0) / 2.0)
        attraction = tf.multiply(p, tf.log(num))
        attraction = -tf.reduce_sum(attraction)

        den = tf.reduce_sum(num, reduction_indices=1) - 1
        repellant = tf.reduce_sum(tf.log(den))

        return (repellant + attraction) / self.n

    def _train_batch(self, x, t):
        p = compute_transition_probability(x, perplexity=self.perplexity)

        feed_dict = {self.x: x,
                     self.p: p,
                     self.batch_size: x.shape[0],
                     self.iter: t}

        _, elbo, tsne_cost = self.sess.run([
            self.train_op,
            self.elbo,
            self.kl_pq],
            feed_dict=feed_dict)

        return elbo, tsne_cost

    def train(self, data, max_iter=1000, batch_size=None,
              pretrained_model=None, verbose=True, verbose_interval=50,
              show_plot=True, plot_dir='./img/'):

        max_iter = max_iter
        batch_size = batch_size or self.hyperparameter['batch_size']

        status = dict()
        status['elbo'] = np.zeros(max_iter)
        status['tsne_cost'] = np.zeros(max_iter)

        if pretrained_model is None:
            self.sess.run(tf.global_variables_initializer())
        else:
            self.load_sess(pretrained_model)

        start = datetime.now()
```

```
            for iter_i in range(max_iter):
                x, y = data.next_batch(batch_size)

                status_batch = self._train_batch(x, iter_i+1)
                status['elbo'][iter_i] = status_batch[0]
                status['tsne_cost'][iter_i] = status_batch[1]

                if verbose and iter_i % verbose_interval == 0:
                    print('Batch {}'.format(iter_i))
                    print((
                        'elbo: {}\n'
                        'scaled_tsne_cost: {}\n').format(
                        status['elbo'][iter_i],
                        status['tsne_cost'][iter_i]))

                    if show_plot:
                        z_mu, _ = self.encode(x)
                        plt.figure(figsize=(10, 7))

                        plt.scatter(z_mu[:, 0], z_mu[:, 1], c=y, s=5)

                        if not os.path.isdir(plot_dir):
                            os.mkdir(plot_dir)

                        name = os.path.join(plot_dir, 'embedding_iter_%05d.png' %
iter_i)
                        plt.savefig(name)
                        plt.close()

            print('Time used for training: {}\n'.format(datetime.now() - start))

            return status

        def encode(self, x):
            var = self.vae.encoder(prob=1.0)
            feed_dict = {self.x: x}

            return self.sess.run(var, feed_dict=feed_dict)

        def decode(self, z):
            var = self.vae.decoder(tf.cast(z, tf.float32))
            feed_dict = {self.z: z, self.batch_size: z.shape[0]}

            return self.sess.run(var, feed_dict=feed_dict)

        def encode_decode(self, x):
            var = [self.latent['mu'],
                   self.latent['sigma_square'],
                   self.decoder_parameter.mu,
                   self.decoder_parameter.sigma_square]

            feed_dict = {self.x: x, self.batch_size: x.shape[0]}

            return self.sess.run(var, feed_dict=feed_dict)

        def save_sess(self, model_name):
            self.saver.save(self.sess, model_name)
```

```
    def load_sess(self, model_name):
        self.saver.restore(self.sess, model_name)

    def get_log_likelihood(self, x, dof=None):

        dof = dof or self.dof
        log_likelihood = log_likelihood_student(
            self.x,
            self.decoder_parameter.mu,
            self.decoder_parameter.sigma_square,
            dof
        )
        num_samples = 5

        feed_dict = {self.x: x, self.batch_size: x.shape[0]}
        log_likelihood_value = 0

        for i in range(num_samples):
            log_likelihood_value += self.sess.run(log_likelihood,
feed_dict=feed_dict)

        log_likelihood_value /= np.float32(num_samples)

        return log_likelihood_value

    def get_elbo(self, x):
        log_likelihood = log_likelihood_student(
            self.x,
            self.decoder_parameter.mu,
            self.decoder_parameter.sigma_square,
            self.dof
        )
        kl_divergence = tf.reduce_mean(0.5 * tf.reduce_sum(self.latent['mu']
** 2 +

self.latent['sigma_square'] -

tf.log(self.latent['sigma_square']) - 1,
                                       reduction_indices=1))

        feed_dict = {self.x: x, self.batch_size: x.shape[0]}

        return self.sess.run(log_likelihood - kl_divergence,
feed_dict=feed_dict)

    def set_normalizer(self, normalizer=1.0):
        normalizer_op = self.normalizer.assign(normalizer)
        self.sess.run(normalizer_op)

    def get_normalizer(self):
        return self.sess.run(self.normalizer)
```

Файл DCA.py:

```
import os
import pickle
from abc import ABCMeta, abstractmethod
```

```
import numpy as np
import scanpy as sc

import keras
from keras.layers import Input, Dense, Dropout, Activation,
BatchNormalization, Lambda
from keras.models import Model
from keras.regularizers import l1_l2
from keras.objectives import mean_squared_error
from keras.initializers import Constant
from keras import backend as K

import tensorflow as tf
class Autoencoder():
    def __init__(self,
                 input_size,
                 output_size=None,
                 hidden_size=(64, 32, 64),
                 l2_coef=0.,
                 l1_coef=0.,
                 l2_enc_coef=0.,
                 l1_enc_coef=0.,
                 ridge=0.,
                 hidden_dropout=0.,
                 input_dropout=0.,
                 batchnorm=True,
                 activation='relu',
                 init='glorot_uniform',
                 file_path=None,
                 debug=False):

        self.input_size = input_size
        self.output_size = output_size
        self.hidden_size = hidden_size
        self.l2_coef = l2_coef
        self.l1_coef = l1_coef
        self.l2_enc_coef = l2_enc_coef
        self.l1_enc_coef = l1_enc_coef
        self.ridge = ridge
        self.hidden_dropout = hidden_dropout
        self.input_dropout = input_dropout
        self.batchnorm = batchnorm
        self.activation = activation
        self.init = init
        self.loss = None
        self.file_path = file_path
        self.extra_models = {}
        self.model = None
        self.encoder = None
        self.decoder = None
        self.input_layer = None
        self.sf_layer = None
        self.debug = debug

        if self.output_size is None:
            self.output_size = input_size
```

```
        if isinstance(self.hidden_dropout, list):
            assert len(self.hidden_dropout) == len(self.hidden_size)
        else:
            self.hidden_dropout = [self.hidden_dropout]*len(self.hidden_size)

    def build(self):

        self.input_layer = Input(shape=(self.input_size,), name='count')
        self.sf_layer = Input(shape=(1,), name='size_factors')
        last_hidden = self.input_layer

        if self.input_dropout > 0.0:
            last_hidden = Dropout(self.input_dropout,
name='input_dropout')(last_hidden)

        for i, (hid_size, hid_drop) in enumerate(zip(self.hidden_size,
self.hidden_dropout)):
            center_idx = int(np.floor(len(self.hidden_size) / 2.0))
            if i == center_idx:
                layer_name = 'center'
                stage = 'center'              elif i < center_idx:
                layer_name = 'enc%s' % i
                stage = 'encoder'
            else:
                layer_name = 'dec%s' % (i-center_idx)
                stage = 'decoder'

            if self.l1_enc_coef != 0. and stage in ('center', 'encoder'):
                l1 = self.l1_enc_coef
            else:
                l1 = self.l1_coef

            if self.l2_enc_coef != 0. and stage in ('center', 'encoder'):
                l2 = self.l2_enc_coef
            else:
                l2 = self.l2_coef

            last_hidden = Dense(hid_size, activation=None,
kernel_initializer=self.init,
                                kernel_regularizer=l1_l2(l1, l2),
                                name=layer_name)(last_hidden)
            if self.batchnorm:
                last_hidden = BatchNormalization(center=True,
scale=False)(last_hidden)

                        if self.activation in advanced_activations:
                last_hidden =
keras.layers.__dict__[self.activation](name='%s_act'%layer_name)(last_hidden)
            else:
                last_hidden = Activation(self.activation,
name='%s_act'%layer_name)(last_hidden)

            if hid_drop > 0.0:
                last_hidden = Dropout(hid_drop,
name='%s_drop'%layer_name)(last_hidden)

        self.decoder_output = last_hidden
        self.build_output()
```

```
    def build_output(self):

        self.loss = mean_squared_error
        mean = Dense(self.output_size, kernel_initializer=self.init,
                     kernel_regularizer=l1_l2(self.l1_coef, self.l2_coef),
                     name='mean')(self.decoder_output)
        output = ColwiseMultLayer([mean, self.sf_layer])

        # keep unscaled output as an extra model
        self.extra_models['mean_norm'] = Model(inputs=self.input_layer,
outputs=mean)
        self.extra_models['decoded'] = Model(inputs=self.input_layer,
outputs=self.decoder_output)
        self.model = Model(inputs=[self.input_layer, self.sf_layer],
outputs=output)

        self.encoder = self.get_encoder()

    def save(self):
        if self.file_path:
            os.makedirs(self.file_path, exist_ok=True)
            with open(os.path.join(self.file_path, 'model.pickle'), 'wb') as
f:
                pickle.dump(self, f)

    def load_weights(self, filename):
        self.model.load_weights(filename)
        self.encoder = self.get_encoder()
        self.decoder = None  # get_decoder()

    def get_decoder(self):
        i = 0
        for l in self.model.layers:
            if l.name == 'center_drop':
                break
            i += 1

        return Model(inputs=self.model.get_layer(index=i+1).input,
                     outputs=self.model.output)

    def get_encoder(self, activation=False):
        if activation:
            ret = Model(inputs=self.model.input,
                        outputs=self.model.get_layer('center_act').output)
        else:
            ret = Model(inputs=self.model.input,
                        outputs=self.model.get_layer('center').output)
        return ret

    def predict(self, adata, mode='denoise', return_info=False, copy=False):

        assert mode in ('denoise', 'latent', 'full'), 'Unknown mode'

        adata = adata.copy() if copy else adata

        if mode in ('denoise', 'full'):
            print('dca: Calculating reconstructions...')
```

```
            adata.X = self.model.predict({'count': adata.X,
                                          'size_factors':
adata.obs.size_factors})
        if mode in ('latent', 'full'):
            print('dca: Calculating low dimensional representations...')

            adata.obsm['X_dca'] = self.encoder.predict({'count': adata.X,
                                                        'size_factors':
adata.obs.size_factors})
        if mode == 'latent':
            adata.X = adata.raw.X.copy() #recover normalized expression values

        return adata if copy else None

    def write(self, adata, file_path, mode='denoise', colnames=None):

        colnames = adata.var_names.values if colnames is None else colnames
        rownames = adata.obs_names.values

        print('dca: Saving output(s)...')
        os.makedirs(file_path, exist_ok=True)

        if mode in ('denoise', 'full'):
            print('dca: Saving denoised expression...')
            write_text_matrix(adata.X,
                              os.path.join(file_path, 'mean.tsv'),
                              rownames=rownames, colnames=colnames,
transpose=True)

        if mode in ('latent', 'full'):
            print('dca: Saving latent representations...')
            write_text_matrix(adata.obsm['X_dca'],
                              os.path.join(file_path, 'latent.tsv'),
                              rownames=rownames, transpose=False)

        class ZINBAutoencoder(Autoencoder):

            def build_output(self):
                pi = Dense(self.output_size, activation='sigmoid',
kernel_initializer=self.init,
                           kernel_regularizer=l1_l2(self.l1_coef,
self.l2_coef),
                           name='pi')(self.decoder_output)

                disp = Dense(self.output_size, activation=DispAct,
                             kernel_initializer=self.init,
                             kernel_regularizer=l1_l2(self.l1_coef,
self.l2_coef),
                             name='dispersion')(self.decoder_output)

                mean = Dense(self.output_size, activation=MeanAct,
kernel_initializer=self.init,
                             kernel_regularizer=l1_l2(self.l1_coef,
self.l2_coef),
                             name='mean')(self.decoder_output)
                output = ColwiseMultLayer([mean, self.sf_layer])
                output = SliceLayer(0, name='slice')([output, disp, pi])
```

```
                zinb = ZINB(pi, theta=disp, ridge_lambda=self.ridge,
debug=self.debug)
                self.loss = zinb.loss
                self.extra_models['pi'] = Model(inputs=self.input_layer,
outputs=pi)
                self.extra_models['dispersion'] =
Model(inputs=self.input_layer, outputs=disp)
                self.extra_models['mean_norm'] =
Model(inputs=self.input_layer, outputs=mean)
                self.extra_models['decoded'] = Model(inputs=self.input_layer,
outputs=self.decoder_output)

                self.model = Model(inputs=[self.input_layer, self.sf_layer],
outputs=output)

                self.encoder = self.get_encoder()

            def predict(self, adata, mode='denoise', return_info=False,
copy=False, colnames=None):

                adata = adata.copy() if copy else adata

                if return_info:
                    adata.obsm['X_dca_dispersion'] =
self.extra_models['dispersion'].predict(adata.X)
                    adata.obsm['X_dca_dropout'] =
self.extra_models['pi'].predict(adata.X)

                # warning! this may overwrite adata.X
                super().predict(adata, mode, return_info, copy=False)
                return adata if copy else None

            def write(self, adata, file_path, mode='denoise', colnames=None):
                colnames = adata.var_names.values if colnames is None else
colnames
                rownames = adata.obs_names.values

                super().write(adata, file_path, mode, colnames=colnames)

                if 'X_dca_dispersion' in adata.obsm_keys():
                    write_text_matrix(adata.obsm['X_dca_dispersion'],
                                      os.path.join(file_path,
'dispersion.tsv'),
                                      colnames=colnames, transpose=True)

                if 'X_dca_dropout' in adata.obsm_keys():
                    write_text_matrix(adata.obsm['X_dca_dropout'],
                                      os.path.join(file_path, 'dropout.tsv'),
                                      colnames=colnames, transpose=True)


class ZINBAutoencoder(Autoencoder):

    def build_output(self):
        pi = Dense(self.output_size, activation='sigmoid',
kernel_initializer=self.init,
                   kernel_regularizer=l1_l2(self.l1_coef, self.l2_coef),
```

```
                        name='pi')(self.decoder_output)

        disp = Dense(self.output_size, activation=DispAct,
                             kernel_initializer=self.init,
                             kernel_regularizer=l1_l2(self.l1_coef,
self.l2_coef),
                             name='dispersion')(self.decoder_output)

        mean = Dense(self.output_size, activation=MeanAct,
kernel_initializer=self.init,
                        kernel_regularizer=l1_l2(self.l1_coef, self.l2_coef),
                        name='mean')(self.decoder_output)
        output = ColwiseMultLayer([mean, self.sf_layer])
        output = SliceLayer(0, name='slice')([output, disp, pi])

        zinb = ZINB(pi, theta=disp, ridge_lambda=self.ridge, debug=self.debug)
        self.loss = zinb.loss
        self.extra_models['pi'] = Model(inputs=self.input_layer, outputs=pi)
        self.extra_models['dispersion'] = Model(inputs=self.input_layer,
outputs=disp)
        self.extra_models['mean_norm'] = Model(inputs=self.input_layer,
outputs=mean)
        self.extra_models['decoded'] = Model(inputs=self.input_layer,
outputs=self.decoder_output)

        self.model = Model(inputs=[self.input_layer, self.sf_layer],
outputs=output)

        self.encoder = self.get_encoder()

    def predict(self, adata, mode='denoise', return_info=False, copy=False,
colnames=None):

        adata = adata.copy() if copy else adata

        if return_info:
            adata.obsm['X_dca_dispersion'] =
self.extra_models['dispersion'].predict(adata.X)
            adata.obsm['X_dca_dropout']    =
self.extra_models['pi'].predict(adata.X)


        super().predict(adata, mode, return_info, copy=False)
        return adata if copy else None

    def write(self, adata, file_path, mode='denoise', colnames=None):
        colnames = adata.var_names.values if colnames is None else colnames
        rownames = adata.obs_names.values

        super().write(adata, file_path, mode, colnames=colnames)

        if 'X_dca_dispersion' in adata.obsm_keys():
            write_text_matrix(adata.obsm['X_dca_dispersion'],
                              os.path.join(file_path, 'dispersion.tsv'),
                              colnames=colnames, transpose=True)

        if 'X_dca_dropout' in adata.obsm_keys():
            write_text_matrix(adata.obsm['X_dca_dropout'],
```

```
                                os.path.join(file_path, 'dropout.tsv'),
                                colnames=colnames, transpose=True)


def train(adata, network, output_dir=None, optimizer='RMSprop',
learning_rate=None,
          epochs=300, reduce_lr=10, output_subset=None,
use_raw_as_output=True,
          early_stop=15, batch_size=32, clip_grad=5., save_weights=False,
          validation_split=0.1, tensorboard=False, verbose=True, threads=None,
          **kwds):

    tf.compat.v1.keras.backend.set_session(
        tf.compat.v1.Session(
            config=tf.compat.v1.ConfigProto(
                intra_op_parallelism_threads=threads,
                inter_op_parallelism_threads=threads,
            )
        )
    )
    model = network.model
    loss = network.loss
    if output_dir is not None:
        os.makedirs(output_dir, exist_ok=True)

    if learning_rate is None:
        optimizer = opt.__dict__[optimizer](clipvalue=clip_grad)
    else:
        optimizer = opt.__dict__[optimizer](lr=learning_rate,
clipvalue=clip_grad)

    model.compile(loss=loss, optimizer=optimizer)

    callbacks = []

    if save_weights and output_dir is not None:
        checkpointer = ModelCheckpoint(filepath="%s/weights.hdf5" %
output_dir,
                                       verbose=verbose,
                                       save_weights_only=True,
                                       save_best_only=True)
        callbacks.append(checkpointer)
    if reduce_lr:
        lr_cb = ReduceLROnPlateau(monitor='val_loss', patience=reduce_lr,
verbose=verbose)
        callbacks.append(lr_cb)
    if early_stop:
        es_cb = EarlyStopping(monitor='val_loss', patience=early_stop,
verbose=verbose)
        callbacks.append(es_cb)
    if tensorboard:
        tb_log_dir = os.path.join(output_dir, 'tb')
        tb_cb = TensorBoard(log_dir=tb_log_dir, histogram_freq=1,
write_grads=True)
        callbacks.append(tb_cb)

    if verbose: model.summary()
```

```
    inputs = {'count': adata.X, 'size_factors': adata.obs.size_factors}

    if output_subset:
        gene_idx = [np.where(adata.raw.var_names == x)[0][0] for x in
output_subset]
        output = adata.raw.X[:, gene_idx] if use_raw_as_output else adata.X[:,
gene_idx]
    else:
        output = adata.raw.X if use_raw_as_output else adata.X

    loss = model.fit(inputs, output,
                     epochs=epochs,
                     batch_size=batch_size,
                     shuffle=True,
                     callbacks=callbacks,
                     validation_split=validation_split,
                     verbose=verbose,
                     **kwds)

    return loss


def train():

    tf.compat.v1.keras.backend.set_session(tf.compat.v1.Session())
    net = ZINBAutoencoder()
    net.save()
    net.build()

    losses = train(adata[adata.obs.dca_split == 'train'], net,
                   output_dir=args.outputdir,
                   learning_rate=args.learningrate,
                   epochs=args.epochs, batch_size=args.batchsize,
                   early_stop=args.earlystop,
                   reduce_lr=args.reducelr,
                   output_subset=genelist,
                   optimizer=args.optimizer,
                   clip_grad=args.gradclip,
                   save_weights=args.saveweights,
                   tensorboard=args.tensorboard)

    if genelist:
        predict_columns = adata.var_names[[np.where(adata.var_names==x)[0][0]
for x in genelist]]
    else:
        predict_columns = adata.var_names

    net.predict(adata, mode='full', return_info=True)
    net.write(adata, args.outputdir, mode='full', colnames=predict_columns)
```

ДОДАТОК Б

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	94 с.
	Інші документи	
2.	Презентаційні матеріали	21 плакат

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Баранов Є. О.			Дослідження методів аналізу даних scRNA-seq за допомогою глибинних нейронних мереж	Шифр групи	Код напр./спец.
Перевірив		Шергін В. Л.				СШІм-19-2	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Філатов В.О.					