



Школа глубокого обучения ФПМИ МФТИ

## Домашнее задание. Автоэнкодеры

### Часть 1. Vanilla Autoencoder (10 баллов)

#### 1.1. Подготовка данных (0.5 балла)

```
In [3]: import numpy as np
import pandas as pd

from torch.autograd import Variable
from torchvision import datasets
from torchvision import transforms

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data as data_utils

from sklearn.model_selection import train_test_split
import skimage
from skimage.transform import resize
import skimage.io

import os
import wget
import tarfile
from tqdm.notebook import tqdm
from copy import deepcopy

import matplotlib.pyplot as plt
from IPython.display import clear_output
%matplotlib inline
```

```
In [33]: device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
```

cuda

```
In [4]: def fetch_dataset(attrs_name = "lfw_attributes.txt",
                      images_name = "lfw-deepfneled",
                      dx=80,dy=80,
                      dimx=64,dimy=64
                      ):
    #download if not exists
    if not os.path.exists(images_name):
        print("images not found, downloading...")
        os.system("wget http://vis-www.cs.umass.edu/lfw/lfw-deepfneled.tgz -O tmp.tgz")
        print("extracting...")
        os.system("tar xvzf tmp.tgz && rm tmp.tgz")
        print("done")
        assert os.path.exists(images_name)

    if not os.path.exists(attrs_name):
        print("attributes not found, downloading...")
        os.system("wget http://www.cs.columbia.edu/CAVE/databases/pubfig/download/%s" % attrs_name)
        print("done")

    #read attrs
    df_attrs = pd.read_csv("lfw_attributes.txt",sep='\t',skiprows=1,)
    df_attrs = pd.DataFrame(df_attrs.iloc[:, :-1].values, columns = df_attrs.columns[1:])

    #read photos
    photo_ids = []
    for dirpath, dirnames, filenames in os.walk(images_name):
        for fname in filenames:
            if fname.endswith(".jpg"):
                fpath = os.path.join(dirpath,fname)
                photo_id = fname[:-4].replace('_', ' ').split()
                person_id = ' '.join(photo_id[:-1])
                photo_number = int(photo_id[-1])
                photo_ids.append({'person':person_id,'imagenum':photo_number,'photo_path':fpath})

    photo_ids = pd.DataFrame(photo_ids)
    #(photos now have same order as attributes)
    df = pd.merge(df_attrs,photo_ids,on=('person','imagenum'))

    assert len(df)==len(df_attrs),"lost some data when merging dataframes"

    #image preprocessing
    all_photos = df['photo_path'].apply(skimage.io.imread)\n        .apply(lambda img:img[dy:-dy,dx:-dx])\n        .apply(lambda img: resize(img,[dimx,dimy]))\n\n    all_photos = np.stack(all_photos.values).astype('uint8')
    all_attrs = df.drop(["photo_path","person","imagenum"],axis=1)

    return all_photos, all_attrs
```

```
In [5]: # The following Line fetches you two datasets: images, usable for autoencoder training and attributes.\n# Those attributes will be required for the final part of the assignment (applying smiles), so please keep them in mind.\nimgsize = 64\n\ndata, attrs = fetch_dataset(dimx=imgsize, dimy=imgsize)
```

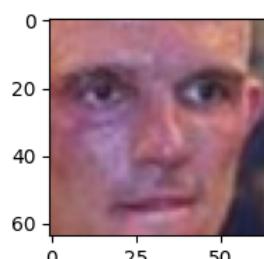
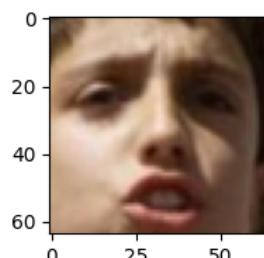
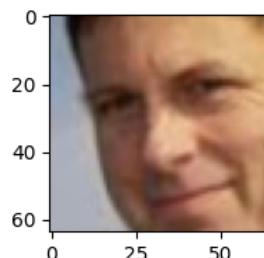
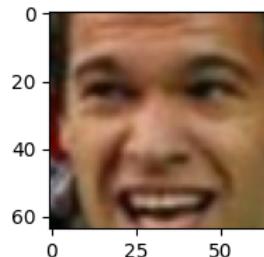
```
In [6]: data.shape, attrs.shape
```

```
Out[6]: ((13143, 64, 64, 3), (13143, 73))
```

Разбейте выборку картинок на train и val, выведите несколько картинок в output, чтобы посмотреть, как они выглядят, и приведите картинки к тензорам pytorch, чтобы можно было скормить их сети:

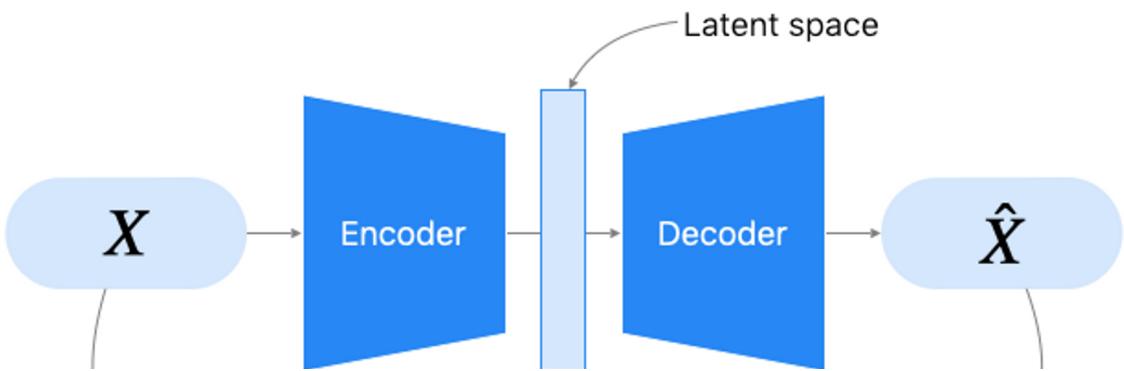
```
In [6]: batch_size = 10\ntrain_photos, val_photos, train_attrs, val_attrs = train_test_split(data, attrs, train_size=0.9, shuffle=True)\ntrain_loader = torch.utils.data.DataLoader(train_photos, batch_size=batch_size)\nval_loader = torch.utils.data.DataLoader(val_photos, batch_size=batch_size)
```

```
In [7]: for i in range(5):
    plt.figure(figsize=(2,2))
    plt.imshow(train_photos[i])
    plt.show()
```



## 1.2. Архитектура модели (1.5 балла)

В этом разделе мы напишем и обучем обычный автоэнкодер.



Реализуем autoencoder. Архитектуру (conv, fully-connected, ReLu, etc) можете выбирать сами. Экспериментируйте!

```
In [10]: class Autoencoder(torch.nn.Module):
    def __init__(self, channels_size=8):
        super().__init__()
        self.channels_size = channels_size

        self.conv_encoder = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=channels_size, kernel_size=3, padding=1),
#            nn.Conv2d(in_channels=3, out_channels=channels_size*2, kernel_size=3, padding=1),
#            nn.ReLU(),
#            nn.BatchNorm2d(channels_size*2),
#            nn.Conv2d(in_channels=channels_size*2, out_channels=channels_size, kernel_size=3, padding=1),
#            nn.ReLU(),
        )
        self.conv_to_linear_encoder = nn.Linear(imgsize*imgsize*channels_size, imgsize*channels_size)
        self.linear_to_conv_decoder = nn.Linear(imgsize*channels_size, imgsize*imgsize*channels_size)
        self.conv_decoder = nn.Sequential(
            nn.ConvTranspose2d(in_channels=channels_size, out_channels=3, kernel_size=3, padding=1),
#            nn.ConvTranspose2d(in_channels=channels_size, out_channels=channels_size*2, kernel_size=3, padding=1),
#            nn.ReLU(),
#            nn.ConvTranspose2d(in_channels=channels_size*2, out_channels=3, kernel_size=3, padding=1),
        )

    def encode(self, x):
        x = x.permute(0, 3, 1, 2)
        pre_latent = self.conv_encoder(x)
        latent = self.conv_to_linear_encoder(pre_latent.reshape((x.shape[0], -1)))
#        print(f"Latent: {latent.shape}")
        return latent

    def decode(self, latent):
#        print(f"Latent: {latent.shape}")
        reconstructed = self.linear_to_conv_decoder(latent)
#        print(f"reconstructed 1: {reconstructed.shape}")
        reconstructed = reconstructed.reshape((latent.shape[0], self.channels_size, imgsize, imgsize))
#        print(f"reconstructed 2: {reconstructed.shape}")
        reconstructed = self.conv_decoder(reconstructed)
        reconstructed = torch.sigmoid(reconstructed)
        reconstructed = reconstructed.permute(0, 2, 3, 1)
        return reconstructed

    def forward(self, x):
#        print(f"x: {x.shape}")
        latent = self.encode(x)
        reconstructed = self.decode(latent)
        return reconstructed
```

### 1.3 Обучение (2 балла)

Осталось написать код обучения автоэнкодера. При этом было бы неплохо в процессе иногда смотреть, как автоэнкодер реконструирует изображения на данном этапе обучения. Например, после каждой эпохи (прогоня train выборки через автоэнкодер) можно смотреть, какие реконструкции получились для каких-то изображений val выборки.

А, ну еще было бы неплохо выводить графики train и val лоссов в процессе тренировки =)

```

In [11]: # Так как будем использовать свертки то зададим размер латентного пространства в виде размера фичимапы
dim_code = 8
max_epochs = 5
model = Autoencoder(channels_size=dim_code).to(device)
optimizer = optim.Adam(model.parameters())
loss_fn = F.mse_loss

history = []
for epoch in tqdm(range(max_epochs)):
    model.train()

    train_losses = 0
    mae_train_losses = 0
    reconstruction = []
    for batch in tqdm(train_loader):
        optimizer.zero_grad()

        reconstruction = model(batch.to(device).float())
        # Считаем Loss MSE для обучения модели
        loss = loss_fn(reconstruction.float(), batch.to(device).float())
        # Считаем Loss MAE для дополнительного сравнения
        mae_loss = nn.L1Loss()(reconstruction.float(), batch.to(device).float())
        loss.backward()
        optimizer.step()
        train_losses += loss.item()
        mae_train_losses += mae_loss
    train_losses = train_losses/len(train_loader)
    mae_train_losses = mae_train_losses/len(train_loader)

    model.eval()
    val_losses = 0
    mae_val_losses = 0
    with torch.no_grad():
        for batch in val_loader:
            reconstruction = model(batch.to(device).float())
            val_loss = loss_fn(reconstruction.float(), batch.to(device).float())
            val_mae_loss = nn.L1Loss()(reconstruction.float(), batch.to(device).float())
            val_losses += val_loss.item()
            mae_val_losses += val_mae_loss

            clear_output(wait=True)
            print(f'epoch: {epoch+1}')
            fig, axs = plt.subplots(nrows=3, ncols=2, figsize=(7,7))
            for i in range(3):
                axs[i, 0].set_title("Original")
                axs[i, 0].axis("off")
                axs[i, 0].imshow(batch[i])

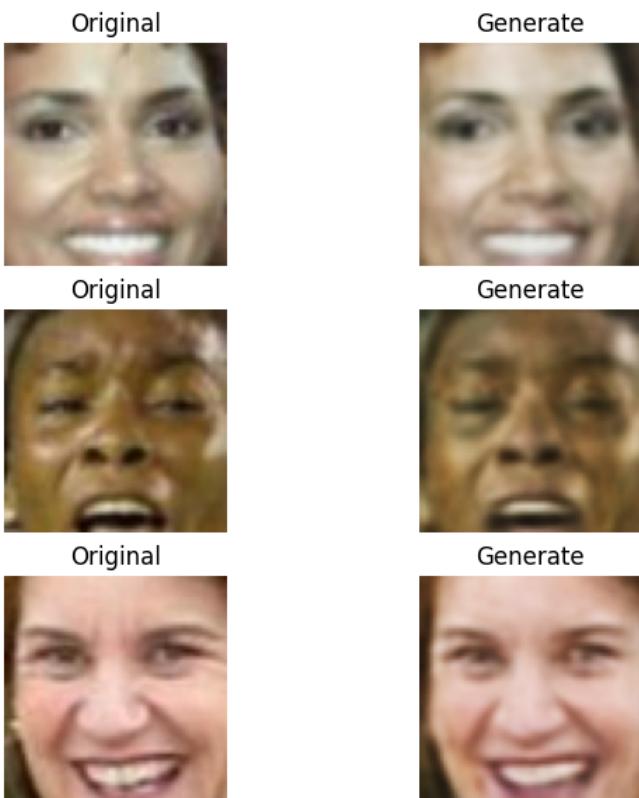
                axs[i, 1].set_title("Generate")
                axs[i, 1].axis("off")
                axs[i, 1].imshow(reconstruction[i].detach().cpu())
            plt.show()

    val_losses = val_losses / len(val_loader)
    mae_val_losses = mae_val_losses / len(val_loader)

#     history.append((train_losses, val_losses, mae_train_losses.item(), mae_val_losses.item()))
    history.append((train_losses, val_losses, mae_train_losses.item(), mae_val_losses.item()))

```

epoch: 5



```
In [13]: # Высвобождаем память на GPU
import gc
import time
# del model
def clear_memory():
    for _ in range(3):
        time.sleep(2)
        torch.cuda.empty_cache()
        gc.collect()
clear_memory()
torch.cuda.empty_cache()
gc.collect()
!nvidia-smi
```

```
Wed May 10 07:01:59 2023
+-----+
| NVIDIA-SMI 516.94      Driver Version: 516.94      CUDA Version: 11.7 |
+-----+-----+-----+
| GPU  Name     TCC/WDDM | Bus-Id     Disp.A  Volatile Uncorr. ECC | | |
| Fan  Temp   Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
|                               |             |           | MIG M. |
+-----+-----+-----+
| 0  NVIDIA GeForce ... WDDM | 00000000:01:00.0 On |          N/A |
| 30% 40C   P8    12W / 151W| 4010MiB / 4096MiB | 0%       Default |
|                           |                         |          N/A |
+-----+-----+-----+
+-----+
| Processes:
| GPU  GI  CI      PID  Type  Process name          GPU Memory
| ID   ID              ID               Usage
+-----+
| 0  N/A  N/A      860  C+G  ...wekyb3d8bbwe\Video.UI.exe  N/A
| 0  N/A  N/A     5192  C+G  ...y\ShellExperienceHost.exe  N/A
| 0  N/A  N/A     5424  C+G  ... 2020.1\bin\pycharm64.exe  N/A
| 0  N/A  N/A     5844  C+G  ...cw5n1h2txyewy\LockApp.exe  N/A
| 0  N/A  N/A     6452  C+G  ...5n1h2txyewy\SearchApp.exe  N/A
| 0  N/A  N/A     6960  C+G  C:\Windows\explorer.exe  N/A
| 0  N/A  N/A     8832  C+G  ...lPanel\SystemSettings.exe  N/A
| 0  N/A  N/A     9080  C+G  ...2txyewy\TextInputHost.exe  N/A
| 0  N/A  N/A     9136  C+G  ...gram Desktop\Telegram.exe  N/A
| 0  N/A  N/A    10040  C+G  ...bbwe\Microsoft.Photos.exe  N/A
| 0  N/A  N/A    11088  C+G  ...r\Application\browser.exe  N/A
| 0  N/A  N/A    11096  C+G  ...ge\Application\msedge.exe  N/A
| 0  N/A  N/A    11296  C+G  ...me\Application\chrome.exe  N/A
| 0  N/A  N/A    11824  C+G  ...root\Office16\WINWORD.EXE  N/A
| 0  N/A  N/A    13360  C+G  ...3d8bbwe\CalculatorApp.exe  N/A
| 0  N/A  N/A    16648  C  ...ts\_Conda\DLSS2\python.exe  N/A
+-----+
```

```

In [ ]: # Так как будем использовать свертки то зададим размер латентного пространства в виде размера фичимапы
dim_code = 8
max_epochs = 5
model = Autoencoder(channels_size=dim_code).to(device)
optimizer = optim.Adam(model.parameters())
loss_fn = F.mse_loss

history = []
for epoch in tqdm(range(max_epochs)):
    model.train()

    train_losses = 0
    mae_train_losses = 0
    reconstruction = []
    for batch in tqdm(train_loader):
        optimizer.zero_grad()

        reconstruction = model(batch.to(device).float())
        # Считаем Loss MSE для обучения модели
        loss = loss_fn(reconstruction.float(), batch.to(device).float())
        # Считаем Loss MAE для дополнительного сравнения
        mae_loss = nn.L1Loss()(reconstruction.float(), batch.to(device).float())
        loss.backward()
        optimizer.step()
        train_losses += loss.item()
        mae_train_losses += mae_loss
    train_losses = train_losses/len(train_loader)
    mae_train_losses = mae_train_losses/len(train_loader)

    model.eval()
    val_losses = 0
    mae_val_losses = 0
    with torch.no_grad():
        for batch in val_loader:
            reconstruction = model(batch.to(device).float())
            val_loss = loss_fn(reconstruction.float(), batch.to(device).float())
            val_mae_loss = nn.L1Loss()(reconstruction.float(), batch.to(device).float())
            val_losses += val_loss.item()
            mae_val_losses += val_mae_loss

            clear_output(wait=True)
            print(f'epoch: {epoch+1}')
            fig, axs = plt.subplots(nrows=3, ncols=2, figsize=(7,7))
            for i in range(3):
                axs[i, 0].set_title("Original")
                axs[i, 0].axis("off")
                axs[i, 0].imshow(batch[i])

                axs[i, 1].set_title("Generate")
                axs[i, 1].axis("off")
                axs[i, 1].imshow(reconstruction[i].detach().cpu())
            plt.show()

    val_losses = val_losses / len(val_loader)
    mae_val_losses = mae_val_losses / len(val_loader)

#     history.append((train_losses, val_losses, mae_train_losses.item(), mae_val_losses.item()))
history.append((train_losses, val_losses, mae_train_losses.item(), mae_val_losses.item()))

```

```

In [12]: # Посмотрим структуру нашей модели
from torchvision import models
from torchsummary import summary
summary(model, (64, 64, 3))

```

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 8, 64, 64]	224
Linear-2	[-1, 512]	16,777,728
Linear-3	[-1, 32768]	16,809,984
ConvTranspose2d-4	[-1, 3, 64, 64]	219

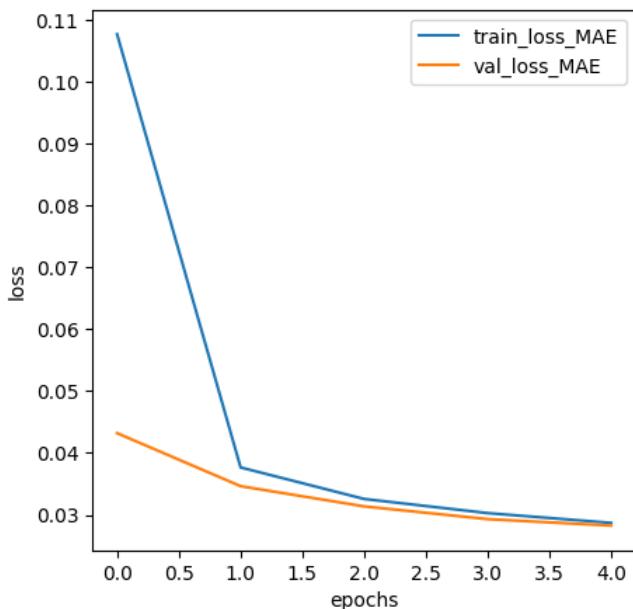
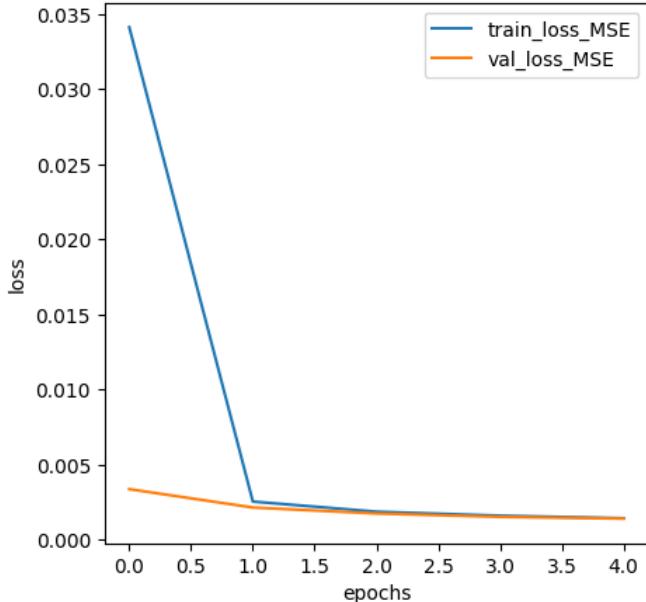
Total params: 33,588,155  
Trainable params: 33,588,155  
Non-trainable params: 0

Input size (MB): 0.05  
Forward/backward pass size (MB): 0.60  
Params size (MB): 128.13  
Estimated Total Size (MB): 128.77

```
In [13]: # Замерялись два лосса MSE и MAE. Обучение производилось на MSE, MAE отслеживался просто дополнительно для ого чтобы
```

```
mse_loss, val_mse_loss, mae_loss, val_mae_loss = zip(*history)
plt.figure(figsize=(5, 5))
plt.plot(mse_loss, label="train_loss_MSE")
plt.plot(val_mse_loss, label="val_loss_MSE")
plt.legend(loc='best')
plt.xlabel("epochs")
plt.ylabel("loss")
plt.show()

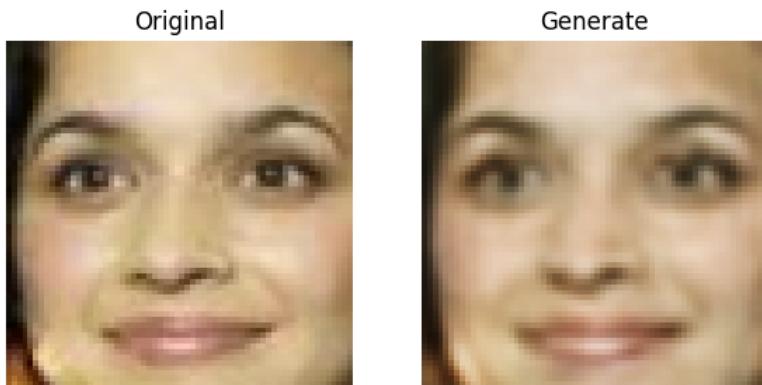
plt.figure(figsize=(5, 5))
plt.plot(mae_loss, label="train_loss_MAE")
plt.plot(val_mae_loss, label="val_loss_MAE")
plt.legend(loc='best')
plt.xlabel("epochs")
plt.ylabel("loss")
plt.show()
```



Давайте посмотрим, как наш тренированный автоэнкодер кодирует и восстанавливает картинки:

```
In [14]: # Выведем первый батч и восстановленные картинки
model.eval()
for batch in val_loader:
    fig, axs = plt.subplots(nrows=len(batch), ncols=2, figsize=(7,60))
    reconstruction = model(batch.to(device).float())
    for i in range(len(batch)):
        axs[i, 0].set_title("Original")
        axs[i, 0].axis("off")
        axs[i, 0].imshow(batch[i])

        axs[i, 1].set_title("Generate")
        axs[i, 1].axis("off")
        axs[i, 1].imshow(reconstruction[i].detach().cpu())
    break
```



Not bad, right?

## 1.4. Sampling (2 балла)

Давайте теперь будем не просто брать картинку, прогонять ее через автоэнкодер и получать реконструкцию, а попробуем создать что-то НОВОЕ

Давайте возьмем и подсунем декодеру какие-нибудь сгенерированные нами векторы (например, из нормального распределения) и посмотрим на результат реконструкции декодера:

Подсказка: Если вместо лиц у вас выводится непонятно что, попробуйте посмотреть, как выглядят латентные векторы картинок из датасета. Так как в обучении нейронных сетей есть определенная доля рандома, векторы латентного слоя могут быть распределены НЕ как `np.random.randn(25, <latent_space_dim>)`. А чтобы у нас получались лица при запихивании вектора декодеру, вектор должен быть распределен так же, как латентные векторы реальных фоток. Так что в таком случае придется рандом немного подогнать.

```
In [15]: # Считаем среднее значение и стандартное отклонение у латентного пространства тренировочной выборки

first_batch = next(iter(val_loader))
# x = temp_img[None].permute(0, 3, 1, 2)
# # x.float().shape, temp_img.shape
first_batch = first_batch.permute(0, 3, 1, 2)
latent_batch = model.conv_encoder(first_batch.to(device).float())
mu_latent = latent_batch.mean(dim=0)
std_latent = latent_batch.std(dim=0)
mu_latent.shape, std_latent.shape
```

Out[15]: (`torch.Size([8, 64, 64])`, `torch.Size([8, 64, 64])`)

```
In [67]: # Считаем среднее значение и стандартное отклонение у латентного пространства тренировочной выборки

latents = []
min_val = np.inf
max_val = 0
for batch in val_loader:
    batch = batch.permute(0, 3, 1, 2)
    latent_batch = model.conv_encoder(batch.to(device).float())
    latent = latent_batch.detach().cpu().numpy()
    latents.extend(latent)
    max_val = max(max_val, latent_batch.max().item())
    min_val = min(min_val, latent_batch.min().item())
latents = np.array(latents)
mu_latent = latents.mean(axis=0)
std_latent = latents.std(axis=0)
mu_latent.shape, std_latent.shape
```

Out[67]: ((`(8, 64, 64)`, `(8, 64, 64)`))

```
In [68]: print(f"min_val: {min_val}, max_val:{max_val}, mu_latent: {mu_latent.mean()}, std_latent: {std_latent.mean()}")
min_val: -0.669684886932373, max_val: 0.6948614716529846, mu_latent: 0.0025391338858753443, std_latent: 0.042544610
79835892
```

```
In [294]: mu_latent.min(), mu_latent.max(), std_latent.min(), std_latent.max()
```

```
Out[294]: (0.0, 3.352777, 0.0, 2.056413)
```

```
In [69]: # Вызов кодера и декодера модели по отдельности
temp_img = next(iter(val_loader))[0]
latent = model.encode(temp_img[None].to(device).float())
reconstr = model.decode(latent)

plt.axis("off")
plt.imshow(reconstr[0].detach().cpu())
plt.show()
```



```
In [70]: # Считаем среднее значение и стандартное отклонение у латентного пространства тренировочной выборки
latents = []
min_val = np.inf
max_val = -np.inf
for batch in val_loader:
    latent_batch = model.encode(batch.to(device).float())
    latent = latent_batch.detach().cpu().numpy()
    latents.extend(latent)
    max_val = max(max_val, latent_batch.max().item())
    min_val = min(min_val, latent_batch.min().item())

latents = np.array(latents)
mu_latent = latents.mean(axis=0)
std_latent = latents.std(axis=0)
print(f"min_val: {min_val}, max_val:{max_val}, mu_latent: {mu_latent.mean()}, std_latent: {std_latent.mean()}")
mu_latent.shape, std_latent.shape
```

```
min_val: -55.01955795288086, max_val: 45.60897445678711, mu_latent: -0.07344985753297806, std_latent: 2.03571462631
2256
```

```
Out[70]: ((512,), (512,))
```

```
In [85]: # сгенерируем 25 случайных векторов размера Latent_space
z = np.random.randn(25, 512)
# Прибавляем к случайному вектору сдвиг (среднее значение по всем векторам) и умножаем на разброс (стандартное отклонение)
z = (z + mu_latent[None]) * std_latent[None]
z = torch.tensor(z).to(device).float()
reconstructed = model.decode(z)

for recons in reconstructed:
    plt.axis("off")
    plt.imshow(recons.detach().cpu())
    plt.show()
```



### 1.5 Time to make fun! (4 балла)

Давайте научимся пририсовывать людям улыбки =)

## so linear

this is you when looking at the HW for the first time



План такой:

1. Нужно выделить "вектор улыбки": для этого нужно из выборки изображений найти несколько (~15) людей с улыбками и столько же без.

Найти людей с улыбками вам поможет файл с описанием датасета, скачанный вместе с датасетом. В нем указаны имена картинок и присутствующие атрибуты (улыбки, очки...)

2. Вычислить латентный вектор для всех улыбающихся людей (прогнать их через encoder) и то же для всех грустненьких
3. Вычислить, собственно, вектор улыбки -- посчитать разность между средним латентным вектором улыбающихся людей и средним латентным вектором грустных людей
4. А теперь приделаем улыбку грустному человеку: добавим полученный в пункте 3 вектор к латентному вектору грустного человека и прогоним полученный вектор через decoder. Получим того же человека, но уже не грустненького!

```
In [162]: # Ищем картинки с улыбками
```

```
# Выберем фото с улыбками
smile_photos = data[attr[attr["Smiling"] > 2.5].index]
for smile in smile_photos[:3]:
    plt.imshow(smile)
    plt.show()

# Выберем фото без улыбками
no_smile_photos = data[attr[attr["Smiling"] <= 2.35].index]
for no_smile in no_smile_photos[:3]:
    plt.imshow(no_smile)
    plt.show()

smile_photos.shape, no_smile_photos.shape
```



```
In [163]: # Вычислить латентный вектор для всех улыбающихся людей (прогнать их через encoder) и то же для всех грустненьких
smile_latents = []
for smile in smile_photos:
    latent_smile = model.encode(torch.tensor(smile[None]).to(device).float())
    smile_latents.extend(latent_smile.detach().cpu().numpy())
smile_latents = np.array(smile_latents)
mean_smile_latents = smile_latents.mean(axis=0)

no_smile_latents = []
for no_smile in no_smile_photos:
    latent_no_smile = model.encode(torch.tensor(no_smile[None]).to(device).float())
    no_smile_latents.extend(latent_no_smile.detach().cpu().numpy())
no_smile_latents = np.array(no_smile_latents)
mean_no_smile_latents = no_smile_latents.mean(axis=0)

# Выводим фото по среднему латентному вектору с улыбкой
mean_smile_reconstr = model.decode(torch.tensor(mean_smile_latents[None]).to(device).float())
plt.axis("off")
plt.imshow(mean_smile_reconstr[0].detach().cpu())
plt.show()

# Выводим фото по среднему латентному вектору без улыбки
mean_no_smile_reconstr = model.decode(torch.tensor(mean_no_smile_latents[None]).to(device).float())
plt.axis("off")
plt.imshow(mean_no_smile_reconstr[0].detach().cpu())
plt.show()
```



```
In [164]: # Вычислим, собственно, вектор улыбки -- посчитать разность между средним латентным вектором улыбающихся людей и ср
smile_vector = mean_smile_latents - mean_no_smile_latents
smile_vector = torch.tensor(smile_vector).to(device).float()

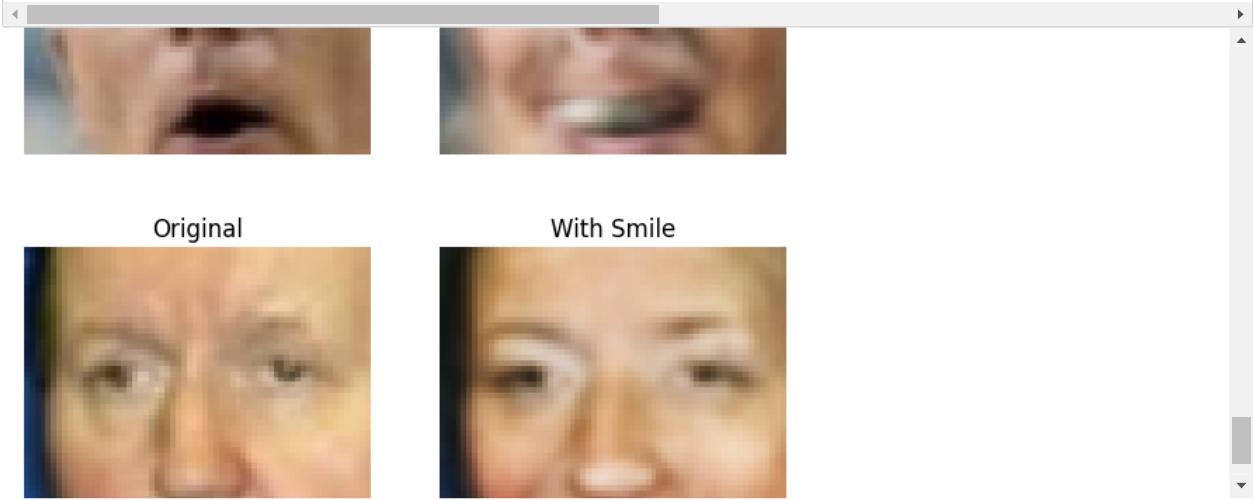
# А теперь приделаем улыбку грустному человеку: добавим полученный в пункте 3 вектор к латентному вектору грустного
fig, axs = plt.subplots(nrows=len(no_smile_photos[:10]), ncols=2, figsize=(7,40))
for i, no_smile in enumerate(no_smile_photos[:10]):
    axs[i, 0].axis("off")
    axs[i, 0].set_title("Original")
    axs[i, 0].imshow(no_smile)

    latent_no_smile = model.encode(torch.tensor(no_smile[None]).to(device).float())

    new_latent = latent_no_smile + smile_vector
    reconstr = model.decode(new_latent)

    axs[i, 1].axis("off")
    axs[i, 1].set_title("With Smile")
    axs[i, 1].imshow(reconstr[0].detach().cpu())

plt.show()
```



Вуаля! Вы восхитительны!

Теперь вы можете пририсовывать людям не только улыбки, но и много чего другого -- закрывать/открывать глаза, пририсовывать очки... в общем, все, на что хватит фантазии и на что есть атрибуты в `all_attrs` :)

## Часть 2: Variational Autoencoder (10 баллов)

Займемся обучением вариационных автоэнкодеров — проапгрейженной версии AE. Обучать будем на датасете MNIST, содержащем написанные от руки цифры от 0 до 9

```
In [8]: batch_size = 32
# MNIST Dataset
train_dataset = datasets.MNIST(root='./mnist_data/', train=True, transform=transforms.ToTensor(), download=True)
test_dataset = datasets.MNIST(root='./mnist_data/', train=False, transform=transforms.ToTensor(), download=False)

# Data Loader (Input Pipeline)
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)
```

### 2.1 Архитектура модели и обучение (2 балла)

Реализуем VAE. Архитектуру (conv, fully-connected, ReLu, etc) можете выбирать сами. Рекомендуем пользоваться более сложными моделями, чем та, что была на семинаре:) Экспериментируйте!

```
In [56]: class VAE(nn.Module):
    def __init__(self, latent_size=32, min_channels_size=8, imgsize=28):
        super().__init__()
        self.latent_size = latent_size
        self.imgsize = imgsize
        self.min_channels_size = min_channels_size

        self.conv_encoder = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=self.min_channels_size*2, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(self.min_channels_size*2),

            nn.Flatten(),
            nn.Linear(in_features=self.imgsize*self.imgsize*self.min_channels_size*2, out_features=self.latent_size)
        )

        self.conv_decoder = nn.Sequential(
            nn.Linear(in_features=latent_size, out_features=self.imgsize*self.imgsize*self.min_channels_size*2),
            nn.Unflatten(1,(self.min_channels_size*2, self.imgsize, self.imgsize)),
            nn.ReLU(),
            nn.BatchNorm2d(self.min_channels_size*2),

            nn.ConvTranspose2d(in_channels=self.min_channels_size*2, out_channels=1, kernel_size=3),
        )

    def encode(self, x):
        """
        <реализуйте forward проход энкодера
        в качестве возвращаемых переменных -- mu и logsigma>
        """
        latent = self.conv_encoder(x)
        latent = latent.view((x.shape[0], 2, -1))
        mu, logsigma = latent[:,0,:], latent[:,1,:]
        return mu, logsigma

    def gaussian_sampler(self, mu, logsigma):
        if self.training:
            # <засемплируйте латентный вектор из нормального распределения с параметрами mu и sigma>
            std = torch.exp(0.5 * logsigma) # standard deviation
            eps = torch.randn_like(std) # `randn_like` as we need the same size
            sample = mu + (eps * std) # sampling as if coming from the input space
            return sample
        else:
            # на инференсе возвращаем не случайный вектор из нормального распределения, а центральный -- mu.
            # на инференсе выход автоэнкодера должен быть детерминирован.
            return mu

    def decode(self, z):
        """
        <реализуйте forward проход декодера
        в качестве возвращаемой переменной -- reconstruction>
        """
        reconstruction = self.conv_decoder(z)
        reconstruction = torch.sigmoid(reconstruction)
        return reconstruction

    def forward(self, x):
        """
        <используя encode и decode, реализуйте forward проход автоэнкодера
        в качестве возвращаемых переменных -- mu, logsigma и reconstruction>
        """
        mu, logsigma = self.encode(x)
        sample = self.gaussian_sampler(mu, logsigma)
        reconstruction = self.decode(sample)
        return mu, logsigma, reconstruction
```

Определим лосс и его компоненты для VAE:

Надеюсь, вы уже прочитали материал в towardsdatascience (или еще где-то) про VAE и знаете, что лосс у VAE состоит из двух частей: KL и log-likelihood.

Общий лосс будет выглядеть так:

$$\mathcal{L} = -D_{KL}(q_\phi(z|x)||p(z)) + \log p_\theta(x|z)$$

Формула для KL-дивергенции:

$$D_{KL} = -\frac{1}{2} \sum_{i=1}^{\dim Z} (1 + \log(\sigma_i^2) - \mu_i^2 - \sigma_i^2)$$

В качестве loss-likelihood возьмем привычную нам кросс-энтропию.

```
In [57]: def KL_divergence(mu, logsigma):
    """
    часть функции потерь, которая отвечает за "близость" латентных представлений разных людей
    """
    #     print(f"Logsigtma: {logsigma}")
    loss = -0.5 * torch.sum(1 + logsigma - mu.pow(2) - logsigma.exp())
    return loss

def log_likelihood(x, reconstruction):
    """
    часть функции потерь, которая отвечает за качество реконструкции (как mse в обычном autoencoder)
    """
    #     Loss = nn.BCELoss()
    loss = nn.BCELoss(reduction="sum")
    return loss(reconstruction, x)

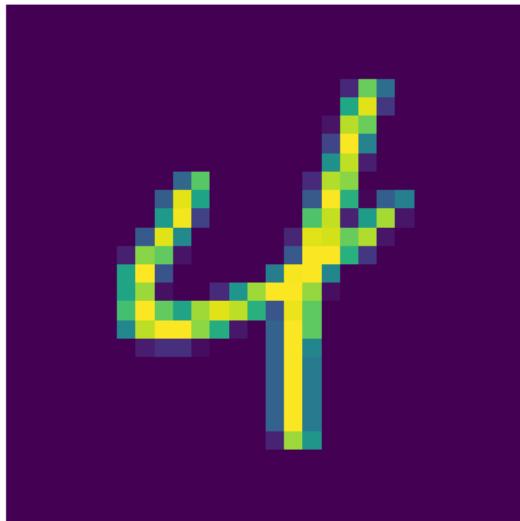
def loss_vae(x, mu, logsigma, reconstruction):
    # <соедините тут две компоненты lossa. Mind the sign!>
    loss_vae = KL_divergence(mu, logsigma) + log_likelihood(x, reconstruction)
    return loss_vae
```

И обучим модель:

```
In [58]: # Выведем изображение из MNIST
temp_img = next(iter(train_loader))
temp_img[0].shape
temp_img = torch.tensor(temp_img[0][0])
temp_img = temp_img.permute(1, 2, 0)
plt.axis("off")
plt.imshow(temp_img)
plt.show()
```

D:\\_Work\\_Projects\\_Conda\DLs2\lib\site-packages\ipykernel\_launcher.py:4: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().requires\_grad\_(True), rather than torch.tensor(sourceTensor).

after removing the cwd from sys.path.



```
In [59]: imgsize=28
latent_size=32
criterion = loss_vae
autoencoder = VAE(imgsize=imgsize, latent_size=latent_size)
optimizer = optim.Adam(autoencoder.parameters())

history = []
max_epochs = 20
for epoch in tqdm(range(max_epochs)):
    print(f"epoch: {epoch}")

    autoencoder.train()
    train_losses = 0
    for x_batch, y_batch in tqdm(train_loader):
        optimizer.zero_grad()
        mu, logsigma, reconstruction = autoencoder(x_batch)
        loss = loss_vae(x_batch, mu, logsigma, reconstruction)
        loss.backward()
        optimizer.step()
        train_losses += loss.item()
    train_losses = train_losses / len(train_loader)

    autoencoder.eval()
    val_losses = 0
    with torch.no_grad():
        for x_batch, y_batch in tqdm(test_loader):
            mu, logsigma, reconstruction = autoencoder(x_batch)
            loss = loss_vae(x_batch, mu, logsigma, reconstruction)
            val_losses += loss.item()

    val_losses = val_losses / len(test_loader)

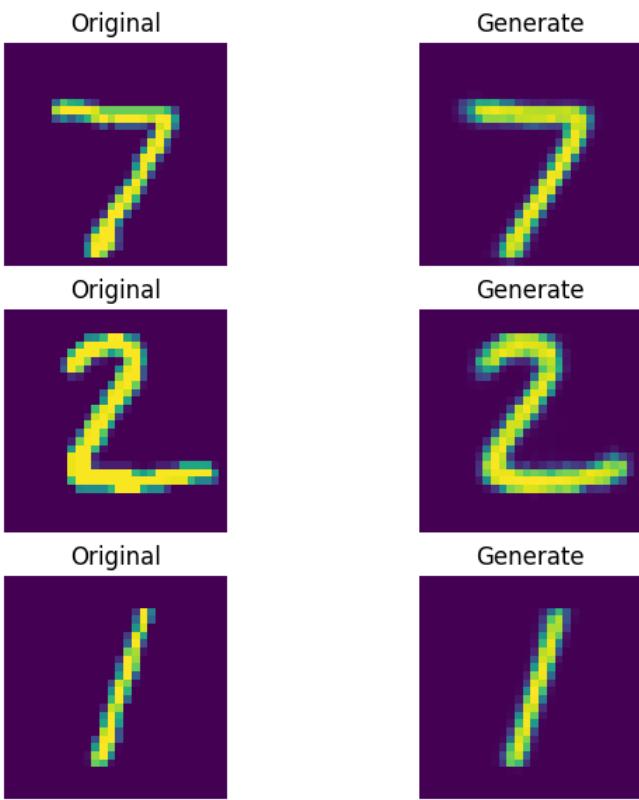
    # Show example
    clear_output(wait=True)
    x_batch, _ = next(iter(test_loader))
    mu, logsigma, reconstruction = autoencoder(x_batch)

    print(f"epoch: {epoch+1}")
    fig, axs = plt.subplots(nrows=3, ncols=2, figsize=(7,7))
    for i in range(3):
        axs[i, 0].set_title("Original")
        axs[i, 0].axis("off")
        axs[i, 0].imshow(x_batch[i].permute(1, 2, 0))

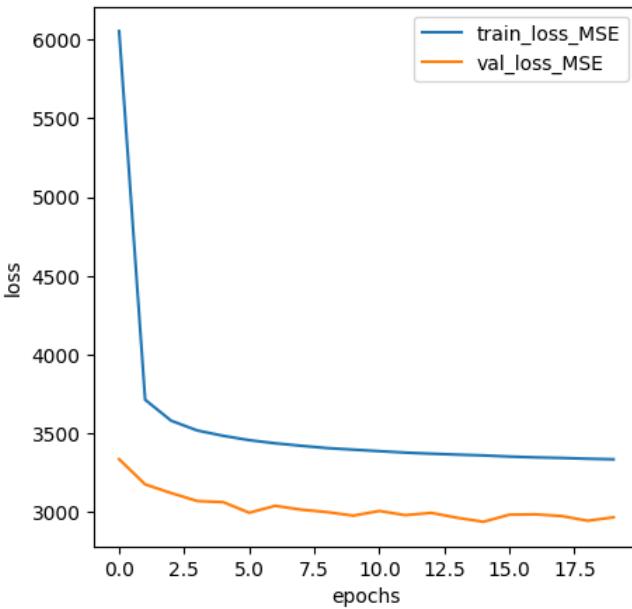
        axs[i, 1].set_title("Generate")
        axs[i, 1].axis("off")
        axs[i, 1].imshow(reconstruction[i].permute(1, 2, 0))
    plt.show()

    history.append((train_losses, val_losses))
```

epoch: 20



```
In [60]: # MSE loss
mse_loss, val_mse_loss = zip(*history)
plt.figure(figsize=(5, 5))
plt.plot(mse_loss, label="train_loss_MSE")
plt.plot(val_mse_loss, label="val_loss_MSE")
plt.legend(loc='best')
plt.xlabel("epochs")
plt.ylabel("loss")
plt.show()
```



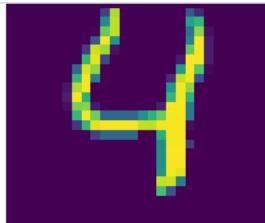
Давайте посмотрим, как наш тренированный VAE кодирует и восстанавливает картинки:

```
In [61]: # < тут Ваш код: выведите первые X картинок и их реконструкций из val выборки на экран>
```

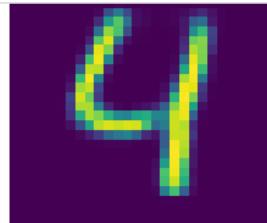
```
x_batch, _ = next(iter(test_loader))
mu, logsigma, reconstruction = autoencoder(x_batch)

print(f"epoch: {epoch+1}")
fig, axs = plt.subplots(nrows=7, ncols=2, figsize=(7,20))
for i in range(7):
    axs[i, 0].set_title("Original")
    axs[i, 0].axis("off")
    axs[i, 0].imshow(x_batch[i].permute(1, 2, 0))

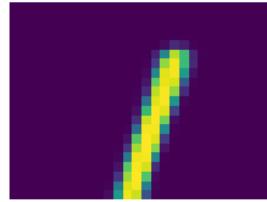
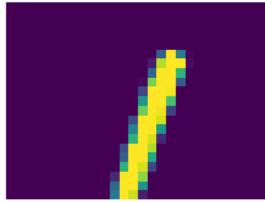
    axs[i, 1].set_title("Generate")
    axs[i, 1].axis("off")
    axs[i, 1].imshow(reconstruction[i].detach().cpu().permute(1, 2, 0))
plt.show()
```



Original



Generate



Давайте попробуем проделать для VAE то же, что и с обычным автоэнкодером -- подсунуть decoder'у из VAE случайные векторы из нормального распределения и посмотреть, какие картинки получаются:

```
# Считаем среднее значение и стандартное отклонение у латентного пространства тренировочной выборки
means = []
stds = []
autoencoder.eval()
for x_batch, y_batch in test_loader:
    mu, logsigma = autoencoder.encode(x_batch)
    means.extend(mu.detach().cpu().numpy())
    stds.extend(logsigma.detach().cpu().numpy())

means = np.array(means)
mean_mu = means.mean(axis=0)
stds = np.array(stds)
mean_std = stds.mean(axis=0)

print(f"mean_mu.shape: {mean_mu.shape}, mean_std.shape: {mean_std.shape}")
mean_mu.mean(), stds.mean()

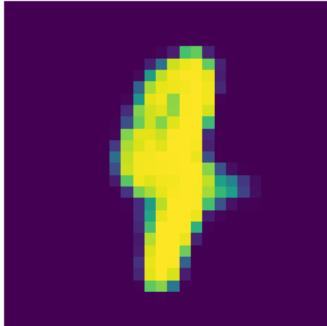
mean_mu.shape: (32,), mean_std.shape: (32,)
```

```
Out[62]: (0.000557221, -1.7663014)
```

```
In [67]: # Вспомните про замечание из этого же пункта обычного AE про распределение латентных переменных
batch_z = np.array([(np.random.rand(latent_size) * mean_std + mean_mu) for i in range(10)])

reconstruction = autoencoder.decode(torch.tensor(batch_z).float())

for recon in reconstruction:
    plt.figure(figsize=(3,7))
    plt.axis("off")
    plt.imshow(recon.detach().cpu().permute(1, 2, 0))
    plt.show()
```



## 2.2. Latent Representation (2 балла)

Давайте посмотрим, как латентные векторы картинок лиц выглядят в пространстве. Ваша задача -- изобразить латентные векторы картинок точками в двумерном пространстве.

Это позволит оценить, насколько плотно распределены латентные векторы изображений цифр в пространстве.

Плюс давайте сделаем такую вещь: покрасим точки, которые соответствуют картинкам каждой цифры, в свой отдельный цвет

Подсказка: красить -- это просто =) У plt.scatter есть параметр c (color), см. в документации.

Итак, план:

1. Получить латентные представления картинок тестового датасета
2. С помощью TSNE (есть в sklearn) сжать эти представления до размерности 2 (чтобы можно было их визуализировать точками в пространстве)
3. Визуализировать полученные двумерные представления с помощью matplotlib.scatter, покрасить разными цветами точки, соответствующие картинкам разных цифр.

```
In [125]: from sklearn.manifold import TSNE
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf']
digital_classes = {i: [] for i in range(10)}

latents = []
targets = []

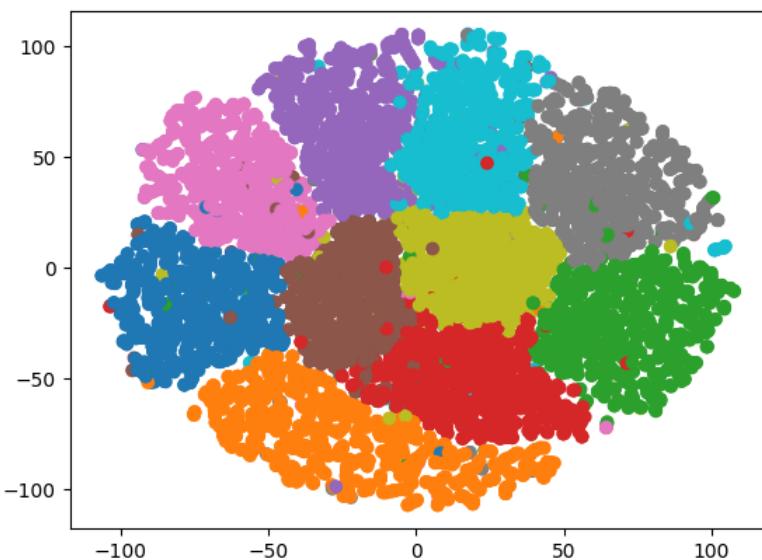
for x_batch, y_batch in test_loader:
    mu, logsigma = autoencoder.encode(x_batch)
    sample = autoencoder.gaussian_sampler(mu, logsigma)
    for i, y_tensor in enumerate(y_batch):
        y = y_tensor.item()
        digital_classes[y].append(sample[i].detach().numpy().flatten())
        latents.append(sample[i].detach().numpy().flatten())
        targets.append(y)

latents = np.array(latents)

X_embedded = TSNE(n_components=2, learning_rate='auto', init='random', perplexity=3).fit_transform(latents)

plt.scatter(X_embedded[:, 0], X_embedded[:, 1], c=[colors[t] for t in targets])
plt.show()
```

Out[125]: <matplotlib.collections.PathCollection at 0x1f8512c3748>



Что вы думаете о виде латентного представления?

Латентные представления довольно четко разделены, при чем обеспечивается довольно хорошее покрытие, т.е. нет пустых пятен, это значит, что будет довольно хорошо определен класс

Congrats v2.0!

### 2.3. Conditional VAE (6 баллов)

Мы уже научились обучать обычный АЕ на датасете картинок и получать новые картинки, используя генерацию шума и декодер. Давайте теперь допустим, что мы обучили АЕ на датасете MNIST и теперь хотим генерировать новые картинки с числами с помощью декодера (как выше мы генерили рандомные лица). И вот нам понадобилось сгенерировать цифру 8, и мы подставляем разные варианты шума, но восьмерка никак не генерится:(

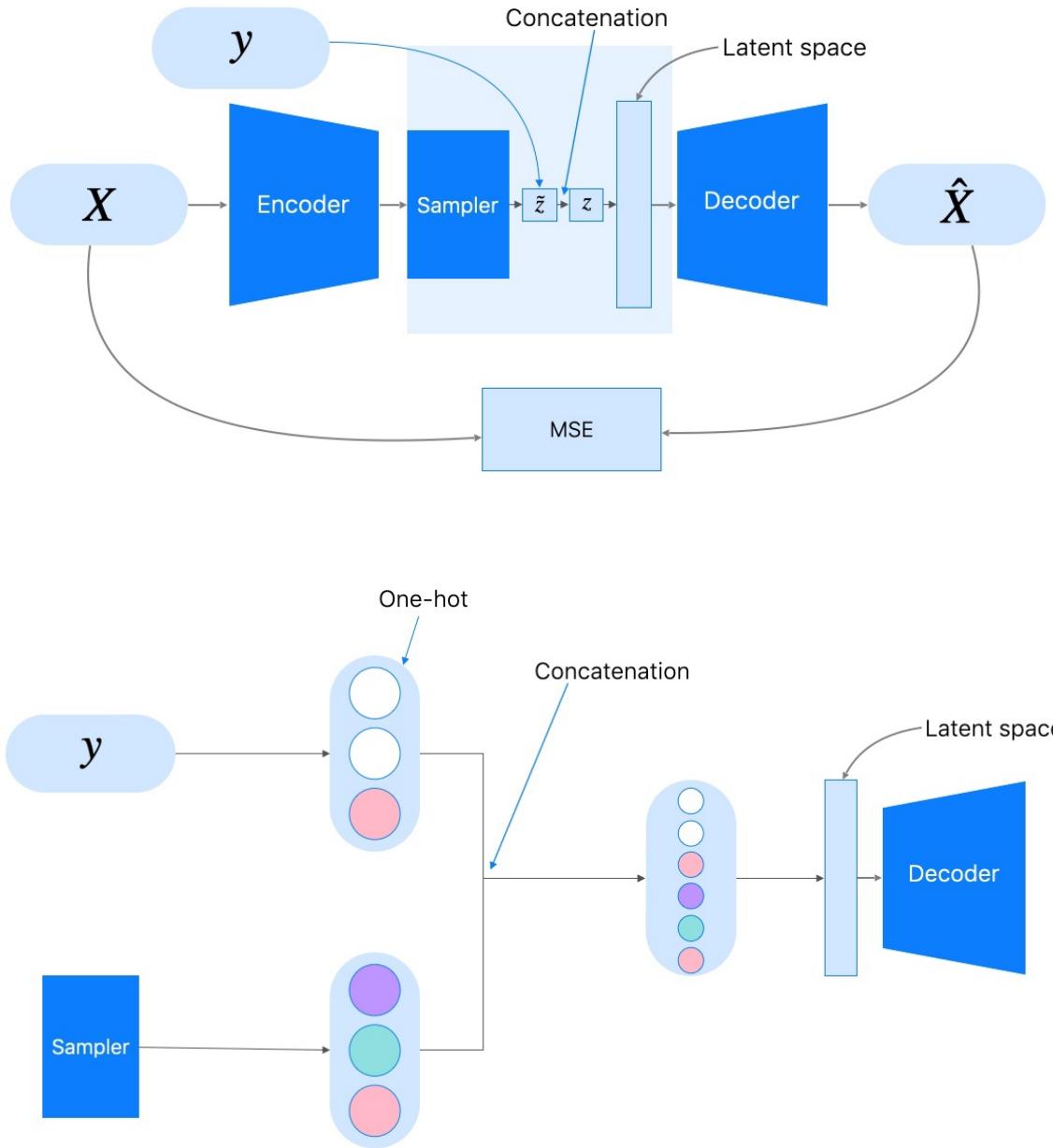
Хотелось бы добавить к нашему АЕ функцию "выдай мне рандомное число из вот этого вот класса", где классов десять (цифры от 0 до 9 образуют десять классов). Conditional AE — так называется вид автоэнкодера, который предоставляет такую возможность. Ну, название "conditional" уже говорит само за себя.

И в этой части задания мы научимся такие обучать.

#### Архитектура

На картинке ниже представлена архитектура простого Conditional VAE.

По сути, единственное отличие от обычного -- это то, что мы вместе с картинкой в первом слое энкодера и декодера передаем еще информацию о классе картинки.



На всякий случай: это VAE, то есть, latent у него все еще состоит из mu и sigma

Таким образом, при генерации новой рандомной картинки мы должны будем передать декодеру сконкатенированные латентный вектор и класс картинки.

P.S. Также можно передавать класс картинки не только в первый слой, но и в каждый слой сети. То есть на каждом слое сконкатенировать выход из предыдущего слоя и информацию о классе.

```

In [191]: class CVAE(nn.Module):
    def __init__(self, latent_size=32, min_channels_size=8, imgsize=28):
        super().__init__()
        self.latent_size = latent_size
        self.imgsize = imgsize
        self.min_channels_size = min_channels_size

        self.conv_encoder = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=self.min_channels_size*2, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(self.min_channels_size*2),

            nn.Flatten(),
            nn.Linear(in_features=self.imgsize*self.imgsize*self.min_channels_size*2, out_features=self.latent_size)
        )

        self.conv_decoder = nn.Sequential(
            nn.Linear(in_features=(latent_size+10), out_features=self.imgsize*self.imgsize*self.min_channels_size*2

            nn.Unflatten(1,(self.min_channels_size*2, self.imgsize, self.imgsize)),
            nn.ReLU(),
            nn.BatchNorm2d(self.min_channels_size*2),

            nn.ConvTranspose2d(in_channels=self.min_channels_size*2, out_channels=1, kernel_size=1),
        )

    def encode(self, x):
        """
            <реализуйте forward проход энкодера
            в качестве возвращаемых переменных -- mu и logsigma>
        """
        latent = self.conv_encoder(x)
        latent = latent.view((x.shape[0], 2, -1))
        mu, logsigma = latent[:,0,:], latent[:,1,:]
        return mu, logsigma

    def gaussian_sampler(self, mu, logsigma):
        if self.training:
            # <засемплируйте латентный вектор из нормального распределения с параметрами mu и sigma>
            std = torch.exp(0.5 * logsigma) # standard deviation
            eps = torch.randn_like(std) # `randn_like` as we need the same size
            sample = mu + (eps * std) # sampling as if coming from the input space
            return sample
        else:
            # на инференсе возвращаем не случайный вектор из нормального распределения, а центральный -- mu.
            # на инференсе выход автоэнкодера должен быть детерминирован.
            return mu

    def taget_to_ohe(self, y_targets):
        ohe_y_batch = []
        for y in y_targets:
            zeros = np.zeros(10)
            zeros[y] = 1
            ohe_y_batch.append(zeros)

        return torch.tensor(ohe_y_batch)

    def decode(self, z, ohe_class):
        """
            <реализуйте forward проход декодера
            в качестве возвращаемой переменной -- reconstruction>
        """
        # print(f"ohe_class.shape: {ohe_class.shape}")
        # print(f"z.shape: {z.shape}")
        z = torch.cat((ohe_class, z), dim=1).float()
        reconstruction = self.conv_decoder(z)
        reconstruction = torch.sigmoid(reconstruction)
        return reconstruction

    def forward(self, x, y_targets):
        """
            используя encode и decode, реализуйте forward проход автоэнкодера
            в качестве возвращаемых переменных -- mu, logsigma и reconstruction
        """
        ohe_classes = self.taget_to_ohe(y_targets)
        mu, logsigma = self.encode(x)
        sample = self.gaussian_sampler(mu, logsigma)
        reconstruction = self.decode(sample, ohe_classes)
        return mu, logsigma, reconstruction

```

```
In [194]: imgsize=28
latent_size=32
criterion = loss_vae
class_autoencoder = CVAE(imgsize=imgsize, latent_size=latent_size)
optimizer = optim.Adam(class_autoencoder.parameters())

history = []
max_epochs = 5
for epoch in tqdm(range(max_epochs)):
    print(f"epoch: {epoch}")

    class_autoencoder.train()
    train_losses = 0
    for x_batch, y_batch in tqdm(train_loader):
        optimizer.zero_grad()
        mu, logsigma, reconstruction = class_autoencoder(x_batch, y_batch)
        loss = loss_vae(x_batch, mu, logsigma, reconstruction)
        loss.backward()
        optimizer.step()
        train_losses += loss.item()
    train_losses = train_losses / len(train_loader)

    class_autoencoder.eval()
    val_losses = 0
    with torch.no_grad():
        for x_batch, y_batch in tqdm(test_loader):
            mu, logsigma, reconstruction = class_autoencoder(x_batch, y_batch)
            loss = loss_vae(x_batch, mu, logsigma, reconstruction)
            val_losses += loss.item()

    val_losses = val_losses / len(test_loader)

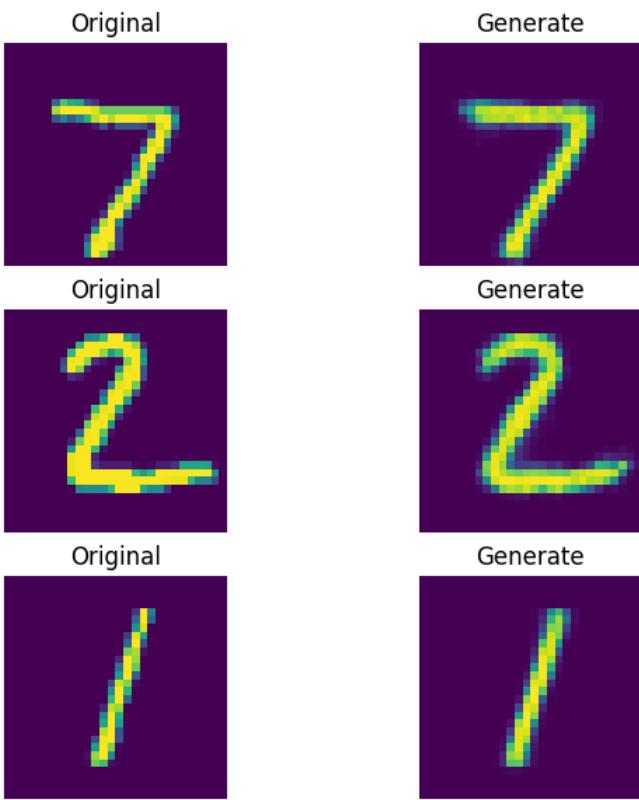
    # Show example
    clear_output(wait=True)
    x_batch, y_batch = next(iter(test_loader))
    mu, logsigma, reconstruction = class_autoencoder(x_batch, y_batch)

    print(f"epoch: {epoch+1}")
    fig, axs = plt.subplots(nrows=3, ncols=2, figsize=(7,7))
    for i in range(3):
        axs[i, 0].set_title("Original")
        axs[i, 0].axis("off")
        axs[i, 0].imshow(x_batch[i].permute(1, 2, 0))

        axs[i, 1].set_title("Generate")
        axs[i, 1].axis("off")
        axs[i, 1].imshow(reconstruction[i].permute(1, 2, 0))
    plt.show()

    history.append((train_losses, val_losses))
```

epoch: 5



## Sampling

Тут мы будем сэмплировать из CVAE. Это прикольнее, чем сэмплировать из простого AE/VAE: тут можно взять один и тот же латентный вектор и попросить CVAE восстановить из него картинки разных классов! Для MNIST вы можете попросить CVAE восстановить из одного латентного вектора, например, картинки цифры 5 и 7.

```
In [214]: # Считаем среднее значение и стандартное отклонение у латентного пространства тренировочной выборки
means = []
stds = []
class_autoencoder.eval()
for x_batch, y_batch in test_loader:
    mu, logsigma = class_autoencoder.encode(x_batch)
    means.extend(mu.detach().cpu().numpy())
    stds.extend(logsigma.detach().cpu().numpy())

means = np.array(means)
mean_mu = means.mean(axis=0)
stds = np.array(stds)
mean_std = stds.mean(axis=0)

print(f"mean_mu.shape: {mean_mu.shape}, mean_std.shape: {mean_std.shape}")
mean_mu.mean(), stds.mean()

mean_mu.shape: (32,), mean_std.shape: (32,)

Out[214]: (-0.013245222, -1.7426767)
```

```
In [248]: # Вспомните про замечание из этого же пункта обычного AE про распределение латентных переменных
N = 10
# batch_z = np.array([(np.random.rand(latent_size) * mean_std + mean_mu) for i in range(10)])
# batch_z = np.array([(np.random.uniform(0,1, size=latent_size) * mean_std + mean_mu) for i in range(10)])
batch_z = np.array([(np.random.uniform(0.01,1, size=latent_size)) for i in range(10)])

batch_yz = np.random.randint(10, size=N)
ohe_classes = class_autoencoder.taget_to_ohe(torch.tensor(batch_yz))

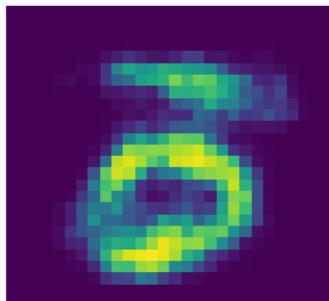
print(f"Случайные классы со случайным распределением: {batch_yz}")

reconstruction = class_autoencoder.decode(torch.tensor(batch_z).float(), ohe_classes)

for i, recon in enumerate(reconstruction):
    plt.figure(figsize=(3,7))
    plt.axis("off")
    plt.title(str(batch_yz[i].item()))
    plt.imshow(recon.detach().cpu().permute(1, 2, 0))
    plt.show()
```



5



Splendid! Вы великолепны!

## Latent Representations

Давайте посмотрим, как выглядит латентное пространство картинок в CVAE и сравним с картинкой для VAE =)

Опять же, нужно покрасить точки в разные цвета в зависимости от класса.

```
In [249]: # <ваш код получения латентных представлений, применения TSNE и визуализации>
from sklearn.manifold import TSNE
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf']
digital_classes = {i: [] for i in range(10)}

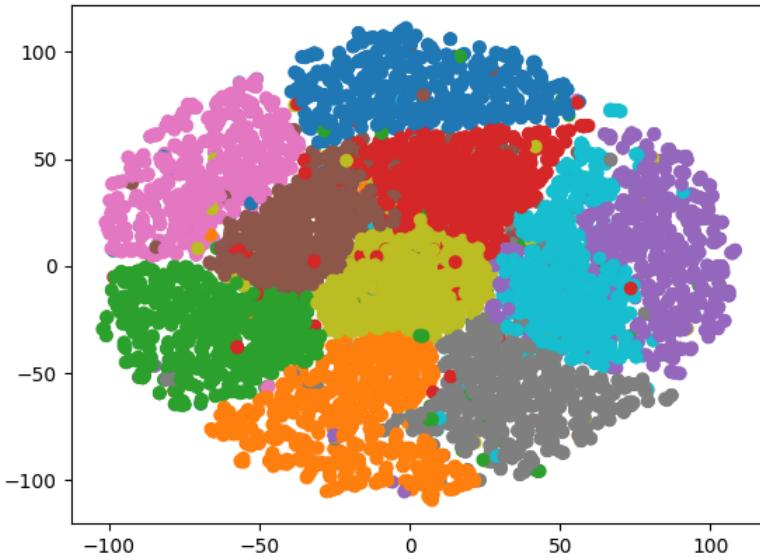
latents = []
targets = []

for x_batch, y_batch in test_loader:
    mu, logsigma = class_autoencoder.encode(x_batch)
    sample = class_autoencoder.gaussian_sampler(mu, logsigma)
    for i, y_tensor in enumerate(y_batch):
        y = y_tensor.item()
        digital_classes[y].append(sample[i].detach().numpy().flatten())
        latents.append(sample[i].detach().numpy().flatten())
        targets.append(y)

latents = np.array(latents)

X_embedded = TSNE(n_components=2, learning_rate='auto', init='random', perplexity=3).fit_transform(latents)

plt.scatter(X_embedded[:, 0], X_embedded[:, 1], c=[colors[t] for t in targets])
plt.show()
```



Что вы думаете насчет этой картинки? Отличается от картинки для VAE?

Здесь также как и в VAE классы довольно хорошо распределяются и четко видно различие между разными классами

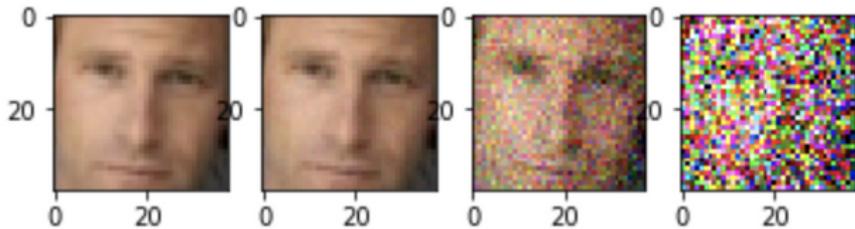
Type *Markdown* and *LaTeX*:  $\alpha^2$

## BONUS 1: Denoising

**Внимание! За бонусы доп. баллы не ставятся, но вы можете сделать их для себя.**

У автоэнкодеров, кроме сжатия и генерации изображений, есть другие практические применения. Про одно из них эта бонусная часть задания.

Автоэнкодеры могут быть использованы для избавления от шума на фотографиях (denoising). Для этого их нужно обучить специальным образом: input картинка будет зашумленной, а выдавать автоэнкодер должен будет картинку без шума. То есть, loss-функция АЕ останется той же (MSE между реальной картинкой и выданной), а на вход автоэнкодеру будет подаваться зашумленная картинка.



(<https://ibb.co/YbRJ1nZ>).

Для этого нужно взять ваш любимый датасет (датасет лиц из первой части этого задания или любой другой) и сделать копию этого датасета с шумом.

В питоне шум можно добавить так:

```
In [ ]: noise_factor = 0.5
X_noisy = X + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=X.shape)
```

```
In [ ]: <тут ваш код обучения автоэнкодера на зашумленных картинках. Не забудьте разбить на train/test!>
```

```
In [ ]: <тут проверка, как АЕ убирает шум с тестовых картинок. Надеюсь, все получилось =>>
```

## BONUS 2: Image Retrieval

**Внимание! За бонусы доп. баллы не ставятся, но вы можете сделать их для себя.**

Давайте представим, что весь наш тренировочный датасет -- это большая база данных людей. И вот мы получили картинку лица какого-то человека с уличной камеры наблюдения (у нас это картинка из тестового датасета) и хотим понять, что это за человек. Что нам делать? Правильно -- берем наш VAE, кодируем картинку в латентное представление и ищем среди латентных представлений лиц нашей базы самые близайшие!

План:

1. Получаем латентные представления всех лиц тренировочного датасета
2. Обучаем на них LSHForest (`sklearn.neighbors.LSHForest`) , например, с `n_estimators=50`
3. Берем картинку из тестового датасета, с помощью VAE получаем ее латентный вектор
4. Ищем с помощью обученного LSHForest близайшие из латентных представлений тренировочной базы
5. Находим лица тренировочного датасета, которым соответствуют близайшие латентные представления, визуализируем!

Немного кода вам в помощь: (feel free to delete everything and write your own)

```
In [ ]: codes = <получите латентные представления картинок из трейна>
```

```
In [ ]: # обучаем LSHForest
from sklearn.neighbors import LSHForest
lshf = LSHForest(n_estimators=50).fit(codes)
```

```
In [ ]: def get_similar(image, n_neighbours=5):
    # функция, которая берет тестовый image и с помощью метода kneighbours у LSHForest ищет близайшие векторы
    # прогоняет векторы через декодер и получает картинки ближайших людей

    code = <получение латентного представления image>
    (distances,), (idx,) = lshf.kneighbors(code, n_neighbours=n_neighbours)

    return distances, X_train[idx]
```

```
In [ ]: def show_similar(image):  
    # функция, которая принимает тестовый image, ищет ближайшие к нему и визуализирует результат  
  
    distances,neighbors = get_similar(image,n_neighbors=11)  
  
    plt.figure(figsize=[8,6])  
    plt.subplot(3,4,1)  
    plt.imshow(image.cpu().numpy().transpose([1,2,0]))  
    plt.title("Original image")  
  
    for i in range(11):  
        plt.subplot(3,4,i+2)  
        plt.imshow(neighbors[i].cpu().numpy().transpose([1,2,0]))  
        plt.title("Dist=%3f"%distances[i])  
    plt.show()
```

```
In [ ]: <тут выведите самые похожие лица к какому-нибудь лицу из тестовой части датасета>
```