

Школа глубокого обучения ФПМИ МФТИ

Домашнее задание. Generative adversarial networks

В этом домашнем задании вы обучите GAN генерировать лица людей и посмотрите на то, как можно оценивать качество генерации

```
In [170]: import time
import gc
import os

from torch.utils.data import DataLoader, Dataset, ConcatDataset, Subset
from torchvision.datasets import ImageFolder
import torchvision.transforms as tt
import torch
import torch.nn as nn
import cv2
from tqdm.notebook import tqdm
from torchvision.utils import save_image
from torchvision.utils import make_grid
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Часть 1. Подготовка данных (1 балл)

В качестве обучающей выборки возьмем часть датасета [Flickr Faces](https://github.com/NVlabs/fhq-dataset) (<https://github.com/NVlabs/fhq-dataset>), который содержит изображения лиц людей в высоком разрешении (1024x1024). Оригинальный датасет очень большой, поэтому мы возьмем его часть. Скачать датасет можно [здесь](https://drive.google.com/file/d/1KWPc4Pa7u2TWekUvNu9rTSO0U2eOlZA9/view?usp=sharing) (<https://drive.google.com/file/d/1KWPc4Pa7u2TWekUvNu9rTSO0U2eOlZA9/view?usp=sharing>).

Давайте загрузим наши изображения. Напишите функцию, которая строит DataLoader для изображений, при этом меняя их размер до нужного значения (размер 1024 слишком большой, поэтому мы рекомендуем взять размер 128 либо немного больше)

```
In [6]: PATH_DATASET = "./datasets/"
PATH_IMAGES = PATH_DATASET + "faces_small/"
```

```
In [26]: # Настройки датасета
image_size=64
stats = [[0.5, 0.5, 0.5], [0.5, 0.5, 0.5]]
```

```
In [27]: device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
device
```

```
Out[27]: device(type='cuda')
```

```
In [88]: def get_dataloader(dataset_path, image_size=128, batch_size=16):
    """
        Builds dataloader for training data.
        Use tt.Compose and tt.Resize for transformations
        :param image_size: height and width of the image
        :param batch_size: batch_size of the dataloader
        :returns: DataLoader object
    """
    transform = tt.Compose([
        tt.Resize(image_size),
        tt.CenterCrop(image_size),
        tt.ToTensor(),
        tt.Normalize(*stats),
    ])
    images_ds = ImageFolder(dataset_path, transform)
    images_loader = DataLoader(images_ds, batch_size, shuffle=True, pin_memory=True)
    return images_ds, images_loader

# Денормализация изображений. Используется для отображения нормализованных картинок
def denormalize(img_tensors):
    return img_tensors * stats[1][0] + stats[0][0]

batch_size = 8

train_images_ds, train_images_loader = get_dataloader(PATH_DATASET, image_size=image_size, batch_size=batch_size)
train_images_ds, train_images_loader
```

```
Out[88]: (Dataset ImageFolder
          Number of datapoints: 3143
          Root location: ./datasets/
          StandardTransform
          Transform: Compose(
              Resize(size=64, interpolation=bilinear, max_size=None, antialias=None)
              CenterCrop(size=(64, 64))
              ToTensor()
              Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
          ),
          <torch.utils.data.dataloader.DataLoader at 0x23a573ab7c8>)
```

```
In [91]: # Распределение нормализованных изображений находится между -1 и 1 поэтому в качестве выхода Генератора будет использо
first_batch_images, _ = next(iter(train_images_loader))
first_batch_images.min(), first_batch_images.max()
```

```
Out[91]: (tensor(-1.), tensor(1.))
```

```
In [29]: # Отобразить несколько картинок обучающей выборки (первый батч)
first_batch_images, _ = next(iter(train_images_loader))
fig, ax = plt.subplots(figsize=(8, 8))
ax.set_xticks([])
ax.set_yticks([])
ax.imshow(make_grid(denormalize(first_batch_images[0:10])), nrow=5).permute(1, 2, 0))
plt.show()
```



Часть 2. Построение и обучение модели (2 балла)

Сконструируйте генератор и дискриминатор. Помните, что:

- дискриминатор принимает на вход изображение (тензор размера $3 \times \text{image_size} \times \text{image_size}$) и выдает вероятность того, что изображение настоящее (тензор размера 1)
- генератор принимает на вход тензор шумов размера $\text{latent_size} \times 1 \times 1$ и генерирует изображение размера $3 \times \text{image_size} \times \text{image_size}$

```
In [30]: latent_size = 512
```

```
In [31]: class Disscriminator(nn.Module):
    def __init__(self, count_channels=64):
        super().__init__()
        self.count_channels = count_channels

        self.all = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=self.count_channels, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(self.count_channels),
            nn.LeakyReLU(),
            nn.Conv2d(in_channels=self.count_channels, out_channels=self.count_channels*2, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(self.count_channels*2),
            nn.LeakyReLU(),
            nn.Conv2d(in_channels=self.count_channels*2, out_channels=self.count_channels*4, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(self.count_channels*4),
            nn.LeakyReLU(),
            nn.Conv2d(in_channels=self.count_channels*4, out_channels=self.count_channels*8, kernel_size=4, stride=1, padding=0, bias=False),
            nn.BatchNorm2d(self.count_channels*8),
            nn.LeakyReLU(),
            nn.Conv2d(in_channels=self.count_channels*8, out_channels=1, kernel_size=5, stride=1, padding=0, bias=False),
            #         nn.BatchNorm2d(self.count_channels*16),
            #         nn.ReLU(),
            #         nn.Conv2d(in_channels=4, out_channels=1, kernel_size=4, stride=1, padding=0, bias=False),
            nn.Flatten(),
            #         nn.Linear(in_features=8*8, out_features=1),
            nn.Sigmoid(),
        )

    def forward(self, x):
        # print(f"x: {x.shape}")
        res = self.all(x)
        # print(f"res: {res.shape}")
        return res
```

Перейдем теперь к обучению нашего GAna. Алгоритм обучения следующий:

1. Учим дискриминатор:

- берем реальные изображения и присваиваем им метку 1
- генерируем изображения генератором и присваиваем им метку 0
- обучаем классификатор на два класса

2. Учим генератор:

- генерируем изображения генератором и присваиваем им метку 1
- предсказываем дискриминатором, реальное это изображение или нет

В качестве функции потерь берем бинарную кросс-энтропию

```
In [ ]:
```

```
In [105]: # V5 V1 + SiLU
# Удобный калькулятор слоев Conv/ConvTranspose2d
# https://thanos.charisoudis.gr/blog/a-simple-conv2d-dimensions-calculator-Logger
class Generator(nn.Module):
    def __init__(self, latent_size=128, count_chanel=64):
        super().__init__()
        self.count_chanel = count_chanel
        self.latent_size = latent_size

        self.all = nn.Sequential(
            nn.ConvTranspose2d(in_channels=latent_size, out_channels=count_chanel*8, kernel_size=4, stride=1, padding=1),
            nn.BatchNorm2d(count_chanel*8),
            nn.SiLU(True),

            nn.ConvTranspose2d(in_channels=count_chanel*8, out_channels=count_chanel*4, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(count_chanel*4),
            nn.SiLU(True),

            nn.ConvTranspose2d(in_channels=count_chanel*4, out_channels=count_chanel*2, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(count_chanel*2),
            nn.SiLU(True),

            nn.ConvTranspose2d(in_channels=count_chanel*2, out_channels=count_chanel, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(count_chanel),
            nn.SiLU(True),

            nn.ConvTranspose2d(in_channels=count_chanel, out_channels=3, kernel_size=4, stride=2, padding=1, bias=1),
            nn.Tanh()
        )
    # Неплохая альтернатива
    #     self.all = nn.Sequential(
    #         nn.ConvTranspose2d(in_channels=latent_size, out_channels=count_chanel*8, kernel_size=3, stride=2, padding=1),
    #         nn.BatchNorm2d(count_chanel*8),
    #         nn.SiLU(True),

    #         nn.ConvTranspose2d(in_channels=count_chanel*8, out_channels=count_chanel*6, kernel_size=3, stride=2, padding=1),
    #         nn.BatchNorm2d(count_chanel*6),
    #         nn.SiLU(True),

    #         nn.ConvTranspose2d(in_channels=count_chanel*6, out_channels=count_chanel*4, kernel_size=3, stride=2, padding=1),
    #         nn.BatchNorm2d(count_chanel*4),
    #         nn.SiLU(True),

    #         nn.ConvTranspose2d(in_channels=count_chanel*4, out_channels=count_chanel*2, kernel_size=3, stride=2, padding=1),
    #         nn.BatchNorm2d(count_chanel*2),
    #         nn.SiLU(True),

    #         nn.ConvTranspose2d(in_channels=count_chanel*2, out_channels=count_chanel, kernel_size=3, stride=2, padding=1),
    #         nn.BatchNorm2d(count_chanel),
    #         nn.SiLU(True),

    #         nn.ConvTranspose2d(in_channels=count_chanel, out_channels=3, kernel_size=2, stride=1, padding=0, bias=1),
    #         nn.Tanh()
    #     )

    def forward(self, x):
        res = self.all(x)
        return res
```

```
In [109]: lr = 0.0001

model = {
    "discriminator": Discriminator().to(device),
    "generator": Generator(latent_size).to(device),
}

optimizer = {
    "discriminator": torch.optim.Adam(model["discriminator"].parameters(), lr=lr),
    "generator": torch.optim.Adam(model["generator"].parameters(), lr=lr)
}

criterion = {
    "discriminator": nn.BCELoss(),
    "generator": nn.BCELoss()
}
```



```
In [107]: %%time
# VS
def fit(model, criterion, optimizer, epochs, train_images_loader):
    torch.cuda.empty_cache()
    gc.collect()
    model["discriminator"].train()
    model["generator"].train()
    history = []
    i = 0
    for epoch in tqdm(range(epochs)):
        print(f"epoch: {epoch}")
        mean_real_discriminator_loss = 0
        mean_fake_discriminator_loss = 0
        mean_discriminator_loss = 0
        mean_generator_loss = 0
        for img_batch, classes in tqdm(train_images_loader):
            img_batch = img_batch.to(device)
            print(img_batch.shape)
            # Обучаем дискриминатор на хороших и плохих картинках
            # Очищаем градиенты
            optimizer["discriminator"].zero_grad()

            # Прогнозируем класс для заведомо хороших картинок (класс 1)
            real_targets = torch.ones(img_batch.shape[0])
            real_pred = model["discriminator"](img_batch)[:, 0].cpu()
            real_discriminator_loss = criterion["discriminator"](real_pred, real_targets)

            # Генерируем примеры фейковых картинок (класс 0)
            fake_targets = torch.zeros(img_batch.shape[0])
            fake_latent = torch.randn(img_batch.shape[0], latent_size)
            #print(f"fake_Latent.shape: {fake_Latent.shape}")
            #print(f"Latent.shape: {Latent.shape}")

            fake_gen_image = model["generator"]((fake_latent.to(device)[:, :, None, None]))
            #print(f"fake_Latent.shape: {fake_Latent.to(device)[:, :, None, None].shape}")
            #print(f"fake_gen_image.shape: {fake_gen_image.shape}")

            # Прогнозируем класс для заведомо плохих картинок
            fake_pred = model["discriminator"]((fake_gen_image)[:, 0].cpu())
            fake_discriminator_loss = criterion["discriminator"](fake_pred, fake_targets)
            # Общий loss дискриминатора
            discriminator_loss = real_discriminator_loss + fake_discriminator_loss
            mean_real_discriminator_loss += real_discriminator_loss
            mean_fake_discriminator_loss += fake_discriminator_loss
            mean_discriminator_loss += discriminator_loss

            # Делаем бэкпропагейшн и шаг оптимизатора
            discriminator_loss.backward()
            optimizer["discriminator"].step()

            # Обучаем Генератор
            # Очищаем градиенты
            optimizer["generator"].zero_grad()

            # Генерируем изображение из случайной выборке
            latent = torch.randn(img_batch.shape[0], latent_size)
            gen_image = model["generator"]((latent.to(device)[:, :, None, None]))

            # Смотрим как дискриминатор определит сгенерированную картинку
            # Но теперь мы говорим дискриминатору что это нужные нам картинки (класс 1)
            gen_targets = torch.ones(img_batch.shape[0])
            gen_pred = model["discriminator"]((gen_image)[:, 0].cpu())
            generator_loss = criterion["generator"](gen_pred, gen_targets)
            mean_generator_loss += generator_loss
            #print(f"generator_Loss: {generator_Loss}")

            generator_loss.backward()
            optimizer["generator"].step()

            fig, ax = plt.subplots(figsize=(8, 8))
            ax.set_xticks([]); ax.set_yticks([])
            ax.imshow(make_grid(denormalize(gen_image.cpu().detach()), nrow=4).permute(1, 2, 0))
            plt.show()

            mean_real_discriminator_loss = mean_real_discriminator_loss / len(train_images_loader)
            mean_fake_discriminator_loss = mean_fake_discriminator_loss / len(train_images_loader)
            mean_discriminator_loss = mean_discriminator_loss / len(train_images_loader)
            mean_generator_loss = mean_generator_loss / len(train_images_loader)

            history.append((mean_real_discriminator_loss.detach().cpu(), mean_fake_discriminator_loss.detach().cpu(), mean_discriminator_loss, mean_generator_loss))

    return history
```

```
history = fit(model, criterion, optimizer, epochs=60, train_images_loader=train_images_loader)
    ...
59     else:
60         data = self.dataset[possibly_batched_index]

D:\_Work\_Projects\_Conda\DLSS2\lib\site-packages\torch\utils\data\_utils\fetch.py in <listcomp>(.0)
56         data = self.dataset.__getitem__(possibly_batched_index)
57     else:
--> 58         data = [self.dataset[idx] for idx in possibly_batched_index]
59     else:
60         data = self.dataset[possibly_batched_index]

D:\_Work\_Projects\_Conda\DLSS2\lib\site-packages\torchvision\datasets\folder.py in __getitem__(self, index)
229     sample = self.loader(path)
230     if self.transform is not None:
--> 231         sample = self.transform(sample)
232     if self.target_transform is not None:
233         target = self.target_transform(target)

D:\_Work\_Projects\_Conda\DLSS2\lib\site-packages\torchvision\transforms\transforms.py in __call__(self, img)
93     def __call__(self, img):
94         ...

In [ ]:
```

```
In [110]: # V6 V1 + SiLU + none inplace
# Удобный калькулятор слоев Conv/ConvTranspose2d
# https://thanos.charisoudis.gr/blog/a-simple-conv2d-dimensions-calculator-Logger
class Generator(nn.Module):
    def __init__(self, latent_size=128, count_chanel=64):
        super().__init__()
        self.count_chanel = count_chanel
        self.latent_size = latent_size

        self.all = nn.Sequential(
            nn.ConvTranspose2d(in_channels=latent_size, out_channels=count_chanel*8, kernel_size=4, stride=1, padding=1),
            nn.BatchNorm2d(count_chanel*8),
            nn.SiLU(),

            nn.ConvTranspose2d(in_channels=count_chanel*8, out_channels=count_chanel*4, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(count_chanel*4),
            nn.SiLU(),

            nn.ConvTranspose2d(in_channels=count_chanel*4, out_channels=count_chanel*2, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(count_chanel*2),
            nn.SiLU(),

            nn.ConvTranspose2d(in_channels=count_chanel*2, out_channels=count_chanel, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(count_chanel),
            nn.SiLU(),

            nn.ConvTranspose2d(in_channels=count_chanel, out_channels=3, kernel_size=4, stride=2, padding=1, bias=False),
            nn.Tanh()
        )
    # Неплохая альтернатива
    #     self.all = nn.Sequential(
    #         nn.ConvTranspose2d(in_channels=latent_size, out_channels=count_chanel*8, kernel_size=3, stride=2, padding=1),
    #         nn.BatchNorm2d(count_chanel*8),
    #         nn.SiLU(True),

    #         nn.ConvTranspose2d(in_channels=count_chanel*8, out_channels=count_chanel*6, kernel_size=3, stride=2, padding=1),
    #         nn.BatchNorm2d(count_chanel*6),
    #         nn.SiLU(True),

    #         nn.ConvTranspose2d(in_channels=count_chanel*6, out_channels=count_chanel*4, kernel_size=3, stride=2, padding=1),
    #         nn.BatchNorm2d(count_chanel*4),
    #         nn.SiLU(True),

    #         nn.ConvTranspose2d(in_channels=count_chanel*4, out_channels=count_chanel*2, kernel_size=3, stride=2, padding=1),
    #         nn.BatchNorm2d(count_chanel*2),
    #         nn.SiLU(True),

    #         nn.ConvTranspose2d(in_channels=count_chanel*2, out_channels=count_chanel, kernel_size=3, stride=2, padding=1),
    #         nn.BatchNorm2d(count_chanel),
    #         nn.SiLU(True),

    #         nn.ConvTranspose2d(in_channels=count_chanel, out_channels=3, kernel_size=2, stride=1, padding=0, bias=False),
    #         nn.Tanh()
    #     )

    def forward(self, x):
        res = self.all(x)
        return res
```

```
In [111]: lr = 0.0001

model = {
    "discriminator": Discriminator().to(device),
    "generator": Generator(latent_size).to(device),
}

optimizer = {
    "discriminator": torch.optim.Adam(model["discriminator"].parameters(), lr=lr),
    "generator": torch.optim.Adam(model["generator"].parameters(), lr=lr)
}

criterion = {
    "discriminator": nn.BCELoss(),
    "generator": nn.BCELoss()
}
```



```

In [ ]: %%time
# VS
def fit(model, criterion, optimizer, epochs, train_images_loader):
    torch.cuda.empty_cache()
    gc.collect()
    model["discriminator"].train()
    model["generator"].train()
    history = []
    i = 0
    for epoch in tqdm(range(epochs)):
        print(f"epoch: {epoch}")
        mean_real_discriminator_loss = 0
        mean_fake_discriminator_loss = 0
        mean_discriminator_loss = 0
        mean_generator_loss = 0
        for img_batch, classes in tqdm(train_images_loader):
            img_batch = img_batch.to(device)
            print(img_batch.shape)
            # Обучаем дискриминатор на хороших и плохих картинках
            # Очищаем градиенты
            optimizer["discriminator"].zero_grad()

            # Прогнозируем класс для заведомо хороших картинок (класс 1)
            real_targets = torch.ones(img_batch.shape[0])
            real_pred = model["discriminator"](img_batch)[:, 0].cpu()
            real_discriminator_loss = criterion["discriminator"](real_pred, real_targets)

            # Генерируем примеры фейковых картинок (класс 0)
            fake_targets = torch.zeros(img_batch.shape[0])
            fake_latent = torch.randn(img_batch.shape[0], latent_size)
            #print(f"fake_Latent.shape: {fake_Latent.shape}")
            #print(f"Latent.shape: {Latent.shape}")

            fake_gen_image = model["generator"]((fake_latent.to(device)[:, :, None, None]))
            #print(f"fake_Latent.shape: {fake_Latent.to(device)[:, :, None, None].shape}")
            #print(f"fake_gen_image.shape: {fake_gen_image.shape}")

            # Прогнозируем класс для заведомо плохих картинок
            fake_pred = model["discriminator"]((fake_gen_image)[:, 0].cpu())
            fake_discriminator_loss = criterion["discriminator"](fake_pred, fake_targets)
            # Общий loss дискриминатора
            discriminator_loss = real_discriminator_loss + fake_discriminator_loss
            mean_real_discriminator_loss += real_discriminator_loss
            mean_fake_discriminator_loss += fake_discriminator_loss
            mean_discriminator_loss += discriminator_loss

            # Делаем бэкпропагейшн и шаг оптимизатора
            discriminator_loss.backward()
            optimizer["discriminator"].step()

            # Обучаем Генератор
            # Очищаем градиенты
            optimizer["generator"].zero_grad()

            # Генерируем изображение из случайной выборке
            latent = torch.randn(img_batch.shape[0], latent_size)
            gen_image = model["generator"]((latent.to(device)[:, :, None, None]))

            # Смотрим как дискриминатор определит сгенерированную картинку
            # Но теперь мы говорим дискриминатору что это нужные нам картинки (класс 1)
            gen_targets = torch.ones(img_batch.shape[0])
            gen_pred = model["discriminator"]((gen_image)[:, 0].cpu())
            generator_loss = criterion["generator"](gen_pred, gen_targets)
            mean_generator_loss += generator_loss
            #print(f"generator_Loss: {generator_Loss}")

            generator_loss.backward()
            optimizer["generator"].step()

            fig, ax = plt.subplots(figsize=(8, 8))
            ax.set_xticks([]); ax.set_yticks([])
            ax.imshow(make_grid(denormalize(gen_image.cpu().detach()), nrow=4).permute(1, 2, 0))
            plt.show()

            mean_real_discriminator_loss = mean_real_discriminator_loss / len(train_images_loader)
            mean_fake_discriminator_loss = mean_fake_discriminator_loss / len(train_images_loader)
            mean_discriminator_loss = mean_discriminator_loss / len(train_images_loader)
            mean_generator_loss = mean_generator_loss / len(train_images_loader)

            history.append((mean_real_discriminator_loss.detach().cpu(), mean_fake_discriminator_loss.detach().cpu(), mean_discriminator_loss, mean_generator_loss))

    return history

```



Обучение первые 60 эпох

```
In [112]: history = fit(model, criterion, optimizer, epochs=60, train_images_loader=train_images_loader)
```



epoch: 56

100%

393/393 [02:37<00:00, 2.36it/s]

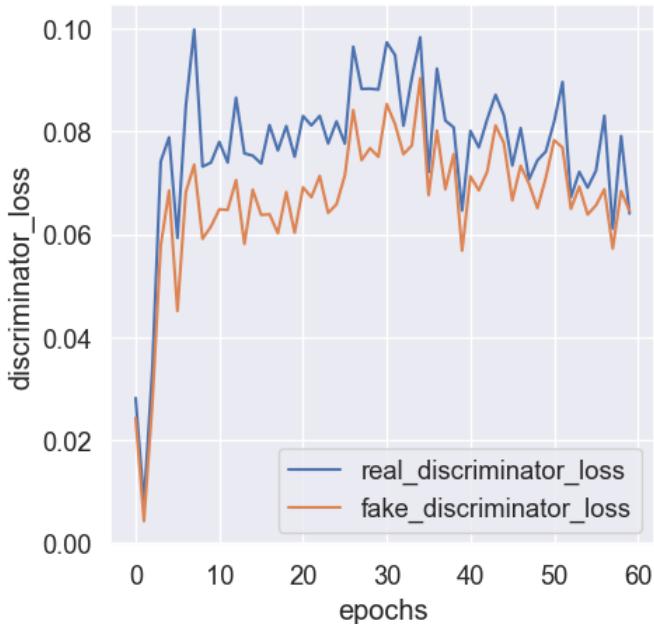
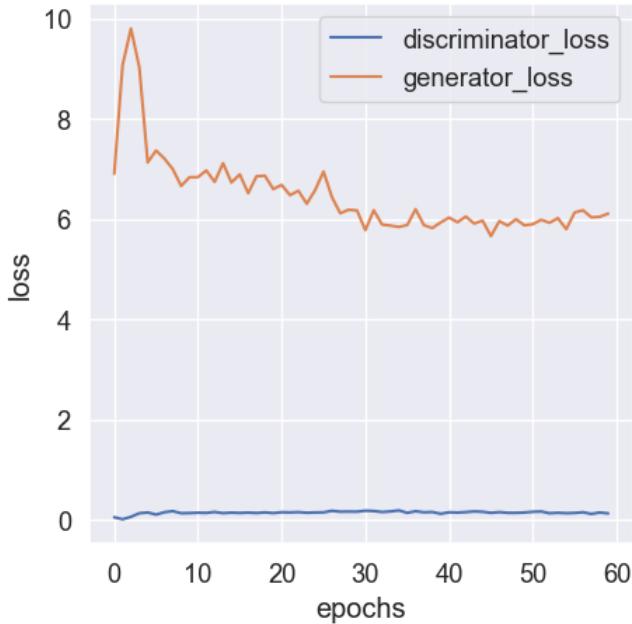


```
In [ ]:
```

```
In [113]: real_discriminator_loss, fake_discriminator_loss, discriminator_loss, generator_loss = zip(*history)

plt.figure(figsize=(5, 5))
plt.plot(discriminator_loss, label="discriminator_loss")
plt.plot(generator_loss, label="generator_loss")
plt.legend(loc='best')
plt.xlabel("epochs")
plt.ylabel("loss")
plt.show()

plt.figure(figsize=(5, 5))
plt.plot(real_discriminator_loss, label="real_discriminator_loss")
plt.plot(fake_discriminator_loss, label="fake_discriminator_loss")
plt.legend(loc='best')
plt.xlabel("epochs")
plt.ylabel("discriminator_loss")
plt.show()
```



Обучение вторые 60 эпох

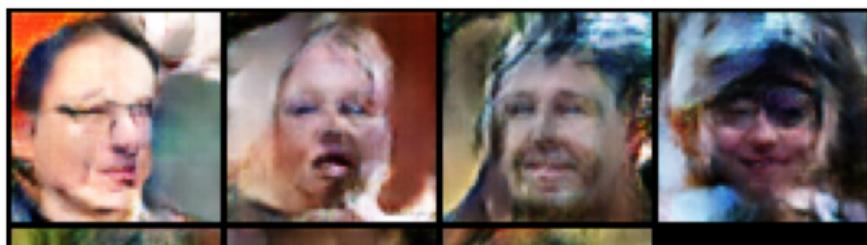
```
In [114]: history2 = fit(model, criterion, optimizer, epochs=60, train_images_loader=train_images_loader)
```



epoch: 56

100%

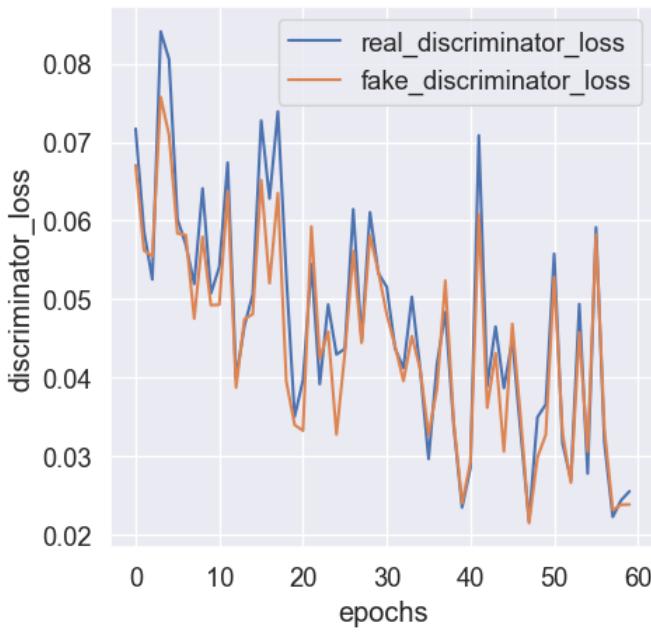
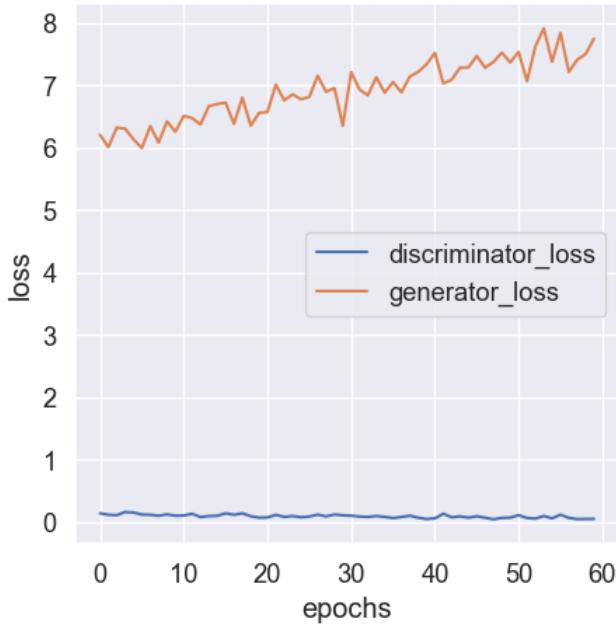
393/393 [02:57<00:00, 2.43it/s]



```
In [115]: real_discriminator_loss, fake_discriminator_loss, discriminator_loss, generator_loss = zip(*history2)

plt.figure(figsize=(5, 5))
plt.plot(discriminator_loss, label="discriminator_loss")
plt.plot(generator_loss, label="generator_loss")
plt.legend(loc='best')
plt.xlabel("epochs")
plt.ylabel("loss")
plt.show()

plt.figure(figsize=(5, 5))
plt.plot(real_discriminator_loss, label="real_discriminator_loss")
plt.plot(fake_discriminator_loss, label="fake_discriminator_loss")
plt.legend(loc='best')
plt.xlabel("epochs")
plt.ylabel("discriminator_loss")
plt.show()
```



Обучение третьи 60 эпох

```
In [116]: history3 = fit(model, criterion, optimizer, epochs=60, train_images_loader=train_images_loader)
```



epoch: 56

100%

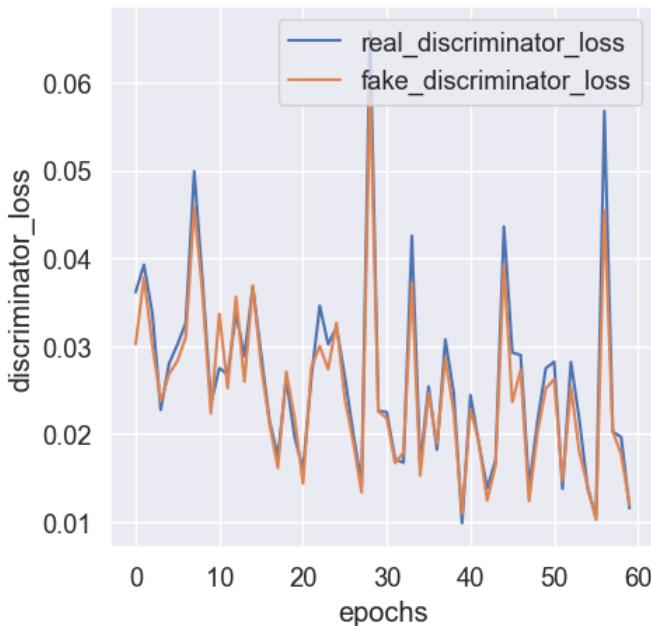
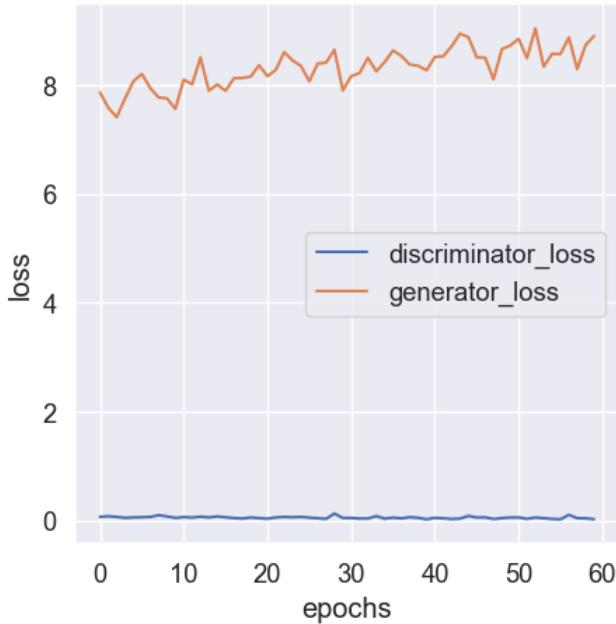
393/393 [03:25<00:00, 1.64it/s]



```
In [126]: real_discriminator_loss, fake_discriminator_loss, discriminator_loss, generator_loss = zip(*history3)

plt.figure(figsize=(5, 5))
plt.plot(discriminator_loss, label="discriminator_loss")
plt.plot(generator_loss, label="generator_loss")
plt.legend(loc='best')
plt.xlabel("epochs")
plt.ylabel("loss")
plt.show()

plt.figure(figsize=(5, 5))
plt.plot(real_discriminator_loss, label="real_discriminator_loss")
plt.plot(fake_discriminator_loss, label="fake_discriminator_loss")
plt.legend(loc='best')
plt.xlabel("epochs")
plt.ylabel("discriminator_loss")
plt.show()
```



In []:

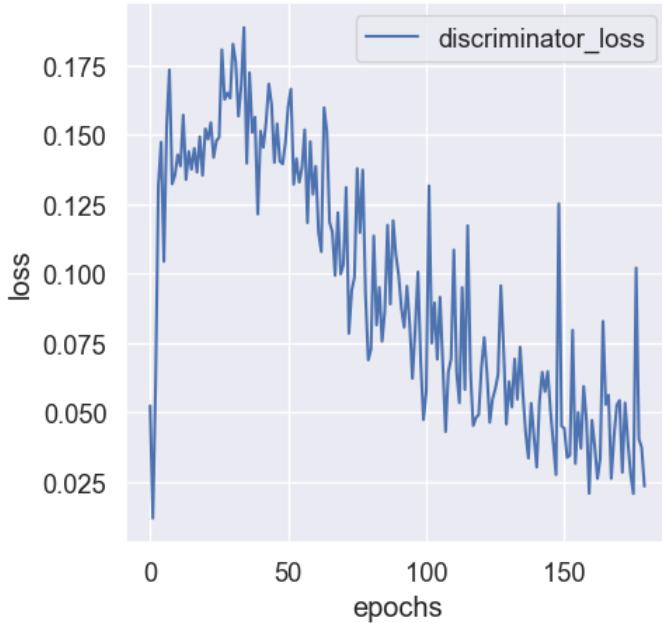
Постройте графики лосса для генератора и дискриминатора. Что вы можете сказать про эти графики?

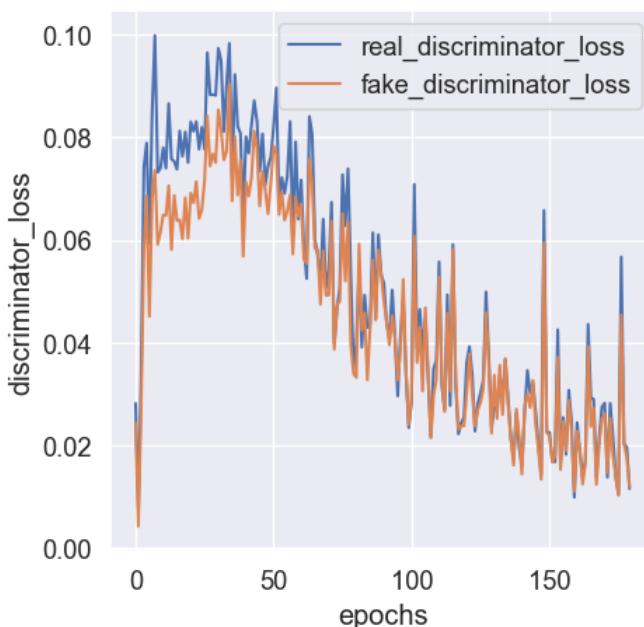
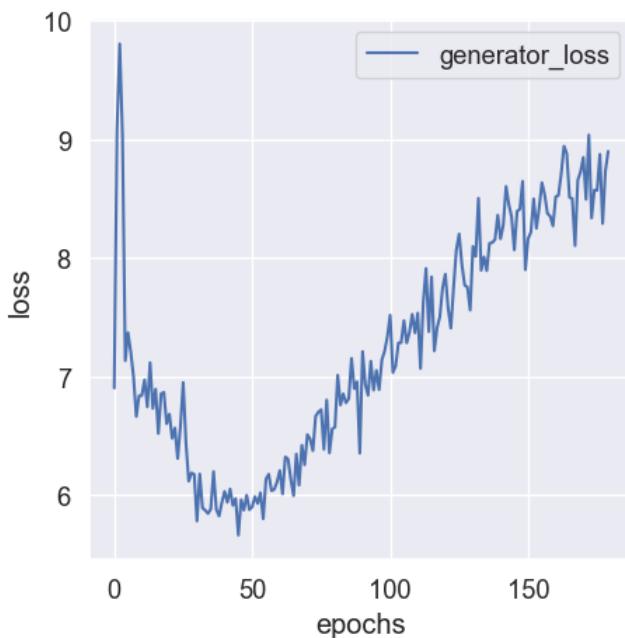
```
In [127]: # Объединяем данные по loss для всех трех заходов по обучению (3 раза по 60 эпох)
real_discriminator_loss, fake_discriminator_loss, discriminator_loss, generator_loss = zip(*history)
real_discriminator_loss2, fake_discriminator_loss2, discriminator_loss2, generator_loss2 = zip(*history2)
real_discriminator_loss3, fake_discriminator_loss3, discriminator_loss3, generator_loss3 = zip(*history3)
discriminator_loss = (discriminator_loss + discriminator_loss2 + discriminator_loss3)
generator_loss = (generator_loss + generator_loss2 + generator_loss3)
real_discriminator_loss = (real_discriminator_loss + real_discriminator_loss2 + real_discriminator_loss3)
fake_discriminator_loss = (fake_discriminator_loss + fake_discriminator_loss2 + fake_discriminator_loss3)

plt.figure(figsize=(5, 5))
plt.plot(discriminator_loss, label="discriminator_loss")
plt.legend(loc='best')
plt.xlabel("epochs")
plt.ylabel("loss")
plt.show()

plt.figure(figsize=(5, 5))
plt.plot(generator_loss, label="generator_loss")
plt.legend(loc='best')
plt.xlabel("epochs")
plt.ylabel("loss")
plt.show()

plt.figure(figsize=(5, 5))
plt.plot(real_discriminator_loss, label="real_discriminator_loss")
plt.plot(fake_discriminator_loss, label="fake_discriminator_loss")
plt.legend(loc='best')
plt.xlabel("epochs")
plt.ylabel("discriminator_loss")
plt.show()
```





По графикам видно что:

1. График лосса генератора как и говорилось на лекции сначала падает затем начинает расти. Аналогично график лосса Дискриминатора, только наоборот сначала возрастает, затем начинает падать. Это говорит о том, что по сути задумка GAN реализована
2. Графики лосса при определении фейковых и реальных изображений очень похожи и уменьшаются, это значит что дискриминатор смог научиться довольно точно определять фейк от реальной фотографии. Это также значит, что генератор довольно плохо работает и не смог обмануть классификатор.

Итого в этой битве выиграл Дискриминатор (

```
In [119]: # Сохраним обе модели
def save_models(model):
    torch.save(model["discriminator"], "discriminator_model_2023.pth")
    torch.save(model["generator"], "generator_model_2023.pth")

save_models(model)
```

In []:

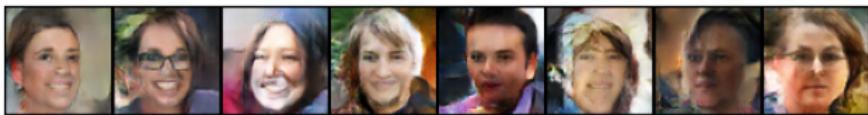
```
In [128]: # Выведем реальные foto
for img_batch, classes in train_images_loader:
    real_img_batch = img_batch

    fig, ax = plt.subplots(figsize=(8, 8))
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(make_grid(denormalize(img_batch.cpu().detach()), nrow=8).permute(1, 2, 0))
    plt.show()
    break
```



```
In [121]: # Выведем фейковые изображения
fixed_latent = torch.randn(batch_size, latent_size, 1, 1, device=device)
fake_images = model["generator"](fixed_latent)

fig, ax = plt.subplots(figsize=(8, 8))
ax.set_xticks([]); ax.set_yticks([])
ax.imshow(make_grid(denormalize(fake_images.cpu().detach()), nrow=8).permute(1, 2, 0))
plt.show()
```



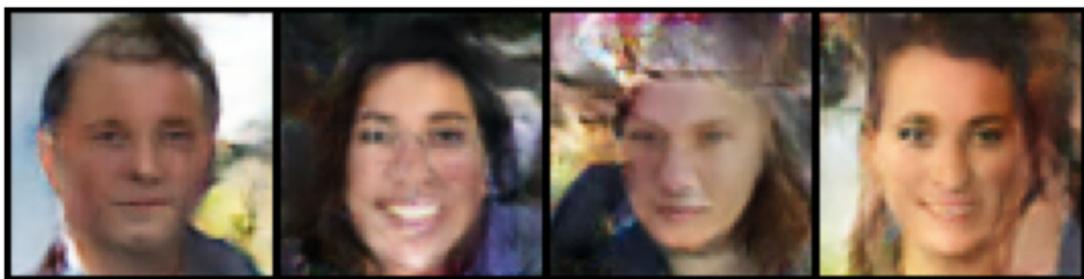
Часть 3. Генерация изображений (1 балл)

Теперь давайте оценим качество получившихся изображений. Напишите функцию, которая выводит изображения, сгенерированные нашим генератором

```
In [129]: n_images = 4

fixed_latent = torch.randn(n_images, latent_size, 1, 1, device=device)
fake_images = model["generator"](fixed_latent)
```

```
In [130]: # Сделаем метод для вывода изображений
def show_images(generated, nrow=8):
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(make_grid(denormalize(generated.cpu().detach()), nrow=nrow).permute(1, 2, 0))
    plt.show()
show_images(fake_images, nrow=8)
```



Как вам качество получившихся изображений?

Ответ

Качество конечно ужасное, но сравнивая последние и первые эпохи можно точно сказать, что это работает! =) Для получения более лучшего качества надо побольше разных трюков попробовать поприменять

Часть 4. Leave-one-out-1-NN classifier accuracy (6 баллов)

4.1. Подсчет accuracy (4 балла)

Не всегда бывает удобно оценивать качество сгенерированных картинок глазами. В качестве альтернативы вам предлагается реализовать следующий подход:

- Сгенерировать столько же фейковых изображений, сколько есть настоящих в обучающей выборке. Присвоить фейковым метку класса 0, настоящим – 1.
- Построить leave-one-out оценку: обучить 1NN Classifier (`sklearn.neighbors.KNeighborsClassifier(n_neighbors=1)`) предсказывать класс на всех объектах, кроме одного, проверить качество (accuracy) на оставшемся объекте. В этом вам поможет `sklearn.model_selection.LeaveOneOut`

Пакетная проверка

Не понятно зачем предлагается использовать LeaveOneOut - это кросс валидация по 1му объекту, т.е. каждый раз обучать модель на всех объектах кроме одного, чтобы сделать предсказания на этом одном, оправдано только в случаях ну очень малого кол-ва данных. Проще сделать деление K-Fold, а в нашем случае так вообще можно нагенерировать фейковых данных. Будем использовать обычный `train_test_split`

```
In [232]: # Сделаем датасет с фейковыми картинками
class FakeDataSetImages(Dataset):
    def __init__(self, gen_model, device, latent_size, total_fakeimg=1000):
        self.total_fakeimg = total_fakeimg
        self.gen_model = gen_model
        self.device = device
        self.latent_size = latent_size

    def __len__(self):
        return self.total_fakeimg

    def __getitem__(self, idx):
        fixed_latent = torch.randn(1, self.latent_size, 1, 1, device=self.device)
        fake_images = self.gen_model(fixed_latent)
        tensor_image = fake_images.cpu().detach()[0]
        # tensor_image = tensor_image.permute(1, 2, 0)
        # по умолчанию при загрузке оригинальных изображений из файлов у них таргет = 0,
        # а тогда фейковым мы указываем таргет = 1, потом развернем
        label = 1
        return tensor_image, label

fake_images_ds = FakeDataSetImages(gen_model=model["generator"], device=device, latent_size=latent_size, total_fakeimg=1000)
test_only_fake_dataset = FakeDataSetImages(gen_model=model["generator"], device=device, latent_size=latent_size, total_fakeimg=1000)
```

```
In [222]: # for tmp in train_images_ds:
#     fig, ax = plt.subplots(figsize=(3, 3))
#     ax.set_xticks([]); ax.set_yticks([])
#     ax.imshow(denormalize(tmp[0]).permute(1, 2, 0))

#     plt.show()
#     break
```



```
In [240]: %%time
# Формируем обучающую и тестовую выборки, а также большую тестовую выборку только с фейками
from sklearn.model_selection import train_test_split

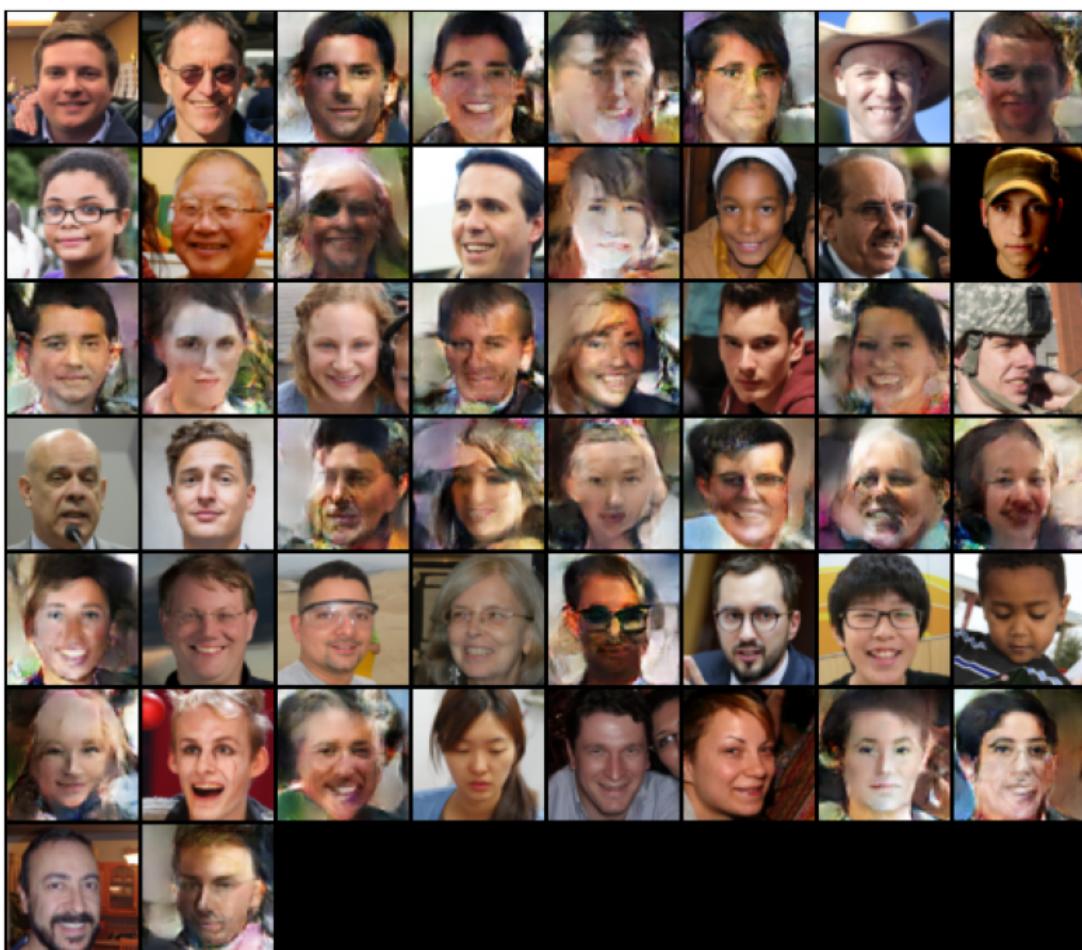
# Объединяем датасет реальных фото с датасетом сгенерированных фото и делаем лоадеры
union_dataset = ConcatDataset([train_images_ds, fake_images_ds])
train_union_dataset, test_union_dataset = train_test_split(union_dataset, test_size=0.2, random_state=53)

# Объединенный трейн (фейк+реал)
train_union_loader = DataLoader(train_union_dataset, batch_size=50, shuffle=True)
# Объединенный тест (фейк+реал)
test_union_loader = DataLoader(test_union_dataset, batch_size=50)
# Только фейк тест
test_only_fake_loader = DataLoader(test_only_fake_dataset, batch_size=50)
```

Wall time: 3min 12s

```
In [241]: # Выведем один пакет изображений из лоадера
# По умолчанию: Label: 0 - реальное фото, 1 - сгенерированное изображение
# Для более понятной интерпретации меняем 1 и 0
for batch_images, label in train_union_loader:
    # по умолчанию при загрузке оригинальных изображений из файлов у них таргет = 0,
    # а фейковым мы указали таргет = 1, теперь разворачиваем
    label = 1-label
    print(label)
    show_images(batch_images)
    break
```

```
tensor([1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0])
```



```
In [242]: # Оценка качества сгенерированных картинок через KNN
from sklearn.neighbors import KNeighborsClassifier

# Формируем общий массив всех данных и подаем на KNN
knn_model = KNeighborsClassifier(n_neighbors=1)
x, y = None, np.array([])
for batch_images, label in tqdm(train_union_loader):
    # по умолчанию при загрузке оригинальных изображений из файлов у них таргет = 0,
    # а фейковым мы указали таргет = 1, теперь разворачиваем
    label = 1-label
    flatten_batch_images = torch.flatten(batch_images, start_dim=1)
    if x is None:
        x = flatten_batch_images.numpy()
        y = label.numpy()
    else:
        x = np.vstack((x, flatten_batch_images.numpy()))
        y = np.append(y, label.numpy())
print(f"x.shape: {x.shape}")
print(f"y.shape: {y.shape}")
# Обучаем KNN
knn_model.fit(x, y)
```

100% 101/101 [00:04<00:00, 11.67it/s]

x.shape: (5028, 12288)
y.shape: (5028,)

Out[242]: KNeighborsClassifier(n_neighbors=1)

Считаем метрику для общего тестового набора (в teste фейк+реал)

```
In [243]: from sklearn.metrics import f1_score, accuracy_score

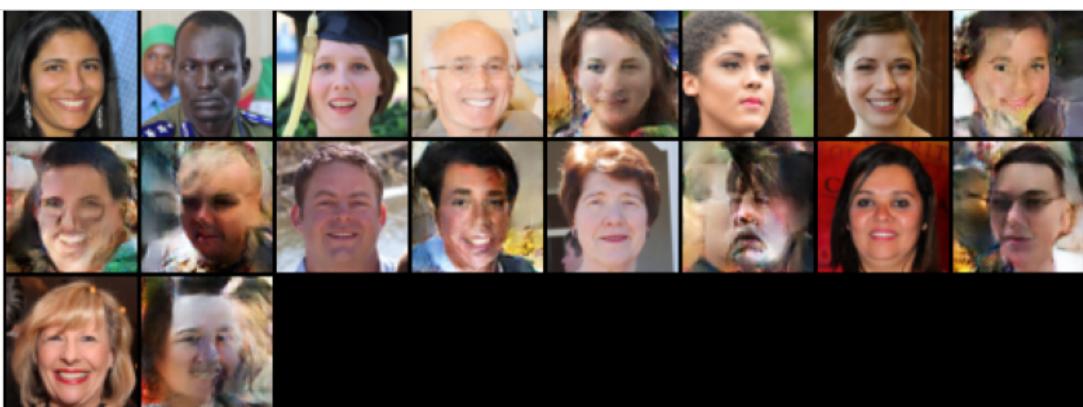
# Вместе фейк + реал
pred_labels = np.array([])
real_labels = np.array([])
for batch_images, real_label in tqdm(test_union_loader):
    # по умолчанию при загрузке оригинальных изображений из файлов у них таргет = 0,
    # а фейковым мы указали таргет = 1, теперь разворачиваем
    real_label = 1-real_label
    flatten_batch_images = torch.flatten(batch_images, start_dim=1)
    pred_label = knn_model.predict(flatten_batch_images)

    pred_labels = np.append(pred_labels, pred_label)
    real_labels = np.append(real_labels, real_label.numpy())

    print(f"pred_label: {pred_label}")
    print(f"real_label: {real_label.numpy()}")
    print(f"f1_score: {f1_score(y_true=real_label.numpy(), y_pred=pred_label)}")
    print(f"accuracy_score: {accuracy_score(y_true=real_label.numpy(), y_pred=pred_label)}")
    show_images(batch_images)

print(pred_labels.shape)
print(real_labels.shape)

print(f"all f1_score: {f1_score(y_true=real_labels, y_pred=pred_labels)}")
print(f"all accuracy_score: {accuracy_score(y_true=real_labels, y_pred=pred_labels)})")
```



pred_label: [1 1 1 0 1 0 0 1 1 0 1 1 0 0 1 0 1 0 1 1 0 0 0 1 1 0 1 0 0 0
0 1 0 1 1 0 0 1 1 0 0 1

Считаем метрику для тестового набора в котором только фейковые данные

На с же интересует как хорошо KNN определяет именно фейковые изображения, поэтому дополнительно тестируем только на фейках которых больше в тесте. Это позволит:

1. Оценивать только фейки

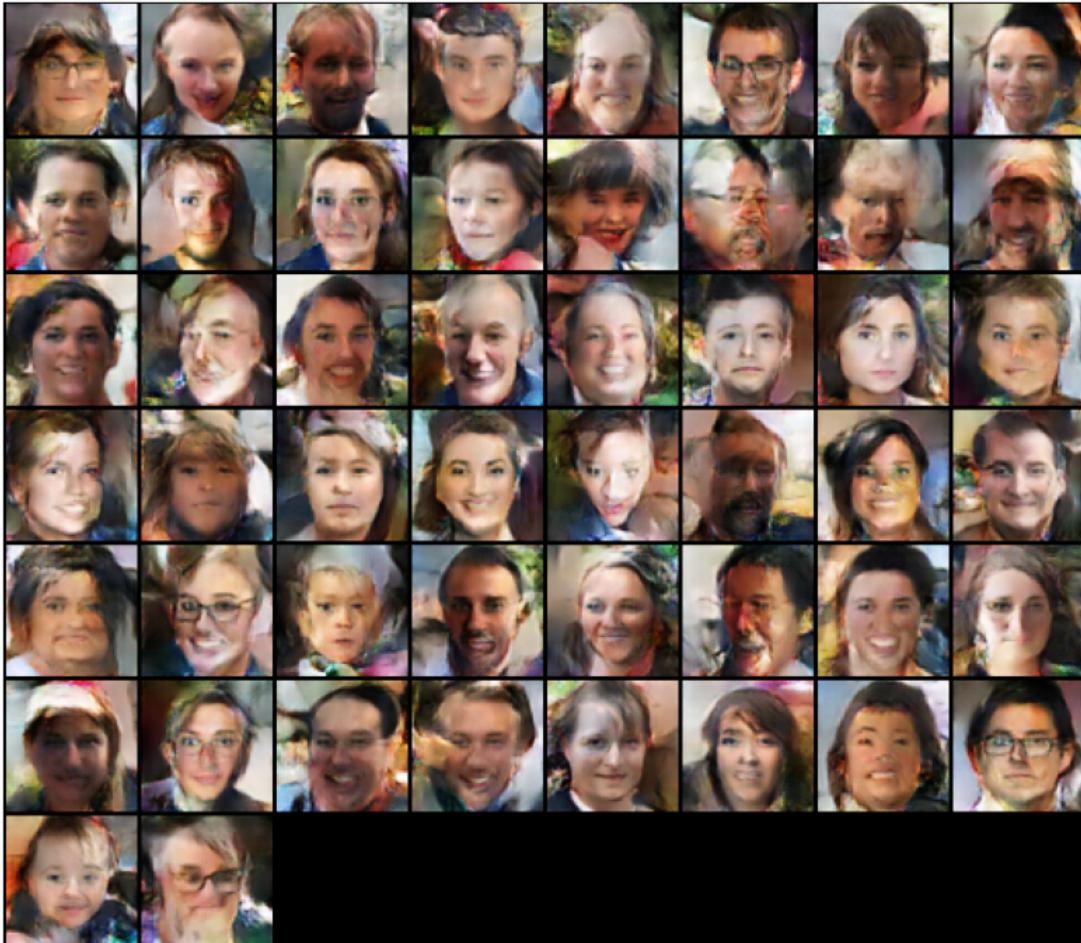
2. Сделать более правильную оценку из-за объема тестируемых фейков

```
In [244]: # Выведем один пакет изображений из лоадера
# По умолчанию: label: 0 - реальное фото, 1 - сгенерированное изображение
# Для более понятной интерпретации меняем 1 и 0
for batch_images, label in test_only_fake_loader:
    # по умолчанию при загрузке оригинальных изображений из файлов у них таргет = 0,
    # а фейковым мы указали таргет = 1, теперь разворачиваем
    label = 1-label
    print(label)
    show_images(batch_images)
    break

tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
    0, 0])
```



```
In [247]: # В тесте только фейк и их много
pred_labels = np.array([])
real_labels = np.array([])
for batch_images, real_label in tqdm(test_only_fake_loader):
    # по умолчанию при загрузке оригинальных изображений из файлов у них метрет = 0,
    # а фейковым мы указали метрет = 1, теперь разворачиваем
    real_label = 1-real_label
    flatten_batch_images = torch.flatten(batch_images, start_dim=1)
    pred_label = knn_model.predict(flatten_batch_images)

    pred_labels = np.append(pred_labels,pred_label)
    real_labels = np.append(real_labels,real_label.numpy())

print(f"pred_label: {pred_label}")
print(f"real_label: {real_label.numpy()}")
print(f"accuracy_score: {accuracy_score(y_true=real_label.numpy(), y_pred=pred_label)}")
show_images(batch_images)

print(f"all accuracy_score: {accuracy_score(y_true=real_labels, y_pred=pred_labels)}")
```



Что вы можете сказать о получившемся результате? Какой accuracy мы хотели бы получить и почему?

Ответ:

Модель классификации не всегда правильно определяет реальное изображение или сгенерированное. Для тестовых данных в которых и реальные и фейковые данные:

- F1_score ~ 86,8%, Accuracy ~ 86,2%
Для тестовых данных в которых только фейковые данные:
- Accuracy: 0.84%

Такая точность обуславливается несколькими факторами:

- knn-модель с n_neighbors=1 сама по себе является довольно легкой моделью для классификации и понятное дело может ошибаться
- размер изображений небольшой всего 64x64 - поэтому незначительные искажения при генерации изображений не сильно влияют/выделяются
- Иногда knn-модель неверно определяет те картинки которые редкие для тренировочного датасета, например у человека на реальном фото сильно повернута голова, такие изображения могут быть засчитаны как фейк
- При генерации попадаются довольно хорошие примеры которые визуально действительно похоже на нормальное фото, но ряд неточностей выдают фейк

Обученный дескrimинатор (классификатор на базе нейронной сети) конечно, справляется в разы лучше.

4.2. Визуализация распределений (2 балла)

Давайте посмотрим на то, насколько похожи распределения настоящих и фейковых изображений. Для этого воспользуйтесь методом, снижающим размерность (к примеру, TSNE) и изобразите на графике разным цветом точки, соответствующие реальным и сгенерированным изображениями

```
In [308]: import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
import seaborn as sns
sns.set(style='darkgrid', font_scale=1.2)
```

```
In [309]: # В качестве распределение картинки следует использовать среднее значение стандартное отклонение  
# Попробуем также использовать анализ главных компоненты (PCA)
```

```
union_test_labels = np.array([])  
union_test_values = None  
for batch_images, label in tqdm(test_union_loader):  
    flatten_batch_images = torch.flatten(batch_images, start_dim=1)  
    if union_test_values is None:  
        union_test_values = flatten_batch_images.numpy()  
    else:  
        union_test_values = np.vstack((union_test_values, flatten_batch_images.numpy()))  
    union_test_labels = np.append(union_test_labels, label.numpy())  
  
print(f"union_test_values.shape : {union_test_values.shape}")  
print(f"union_test_labels.shape : {union_test_labels.shape}")
```

```
100% 26/26 [00:00<00:00, 62.30it/s]
```

```
union_test_values.shape : (1258, 12288)  
union_test_labels.shape : (1258,)
```

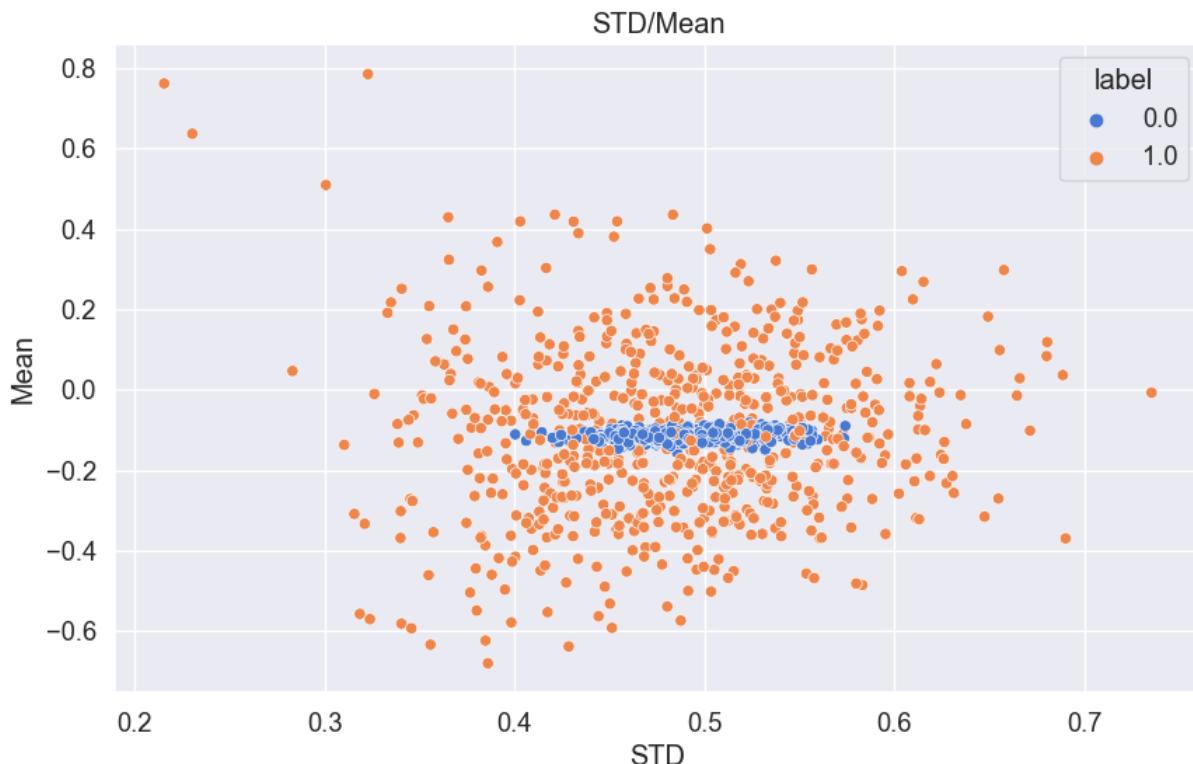
Сравнение изображений через анализ среднего значения и стандартного отклоения

```
In [310]: df_std = pd.DataFrame()  
df_std['Mean'] = union_test_values.mean(axis=1)  
df_std['STD'] = union_test_values.std(axis=1)  
df_std['label'] = 1 - union_test_labels  
df_std.head(3)
```

Out[310]:

	Mean	STD	label
0	-0.130659	0.496415	0.0
1	-0.125747	0.528247	0.0
2	-0.105460	0.506995	0.0

```
In [311]: plt.figure(figsize=(10,6))  
sns.scatterplot(data=df_std, hue='label', palette='muted', x="STD", y="Mean").set_title('STD/Mean');  
plt.show()
```



Сравнение изображений через анализ главных компонентов (PCA)

```
In [312]: pca = PCA(n_components=2)
union_test_pca = pca.fit_transform(union_test_values)
union_test_pca.shape, x_std_pca.shape
```

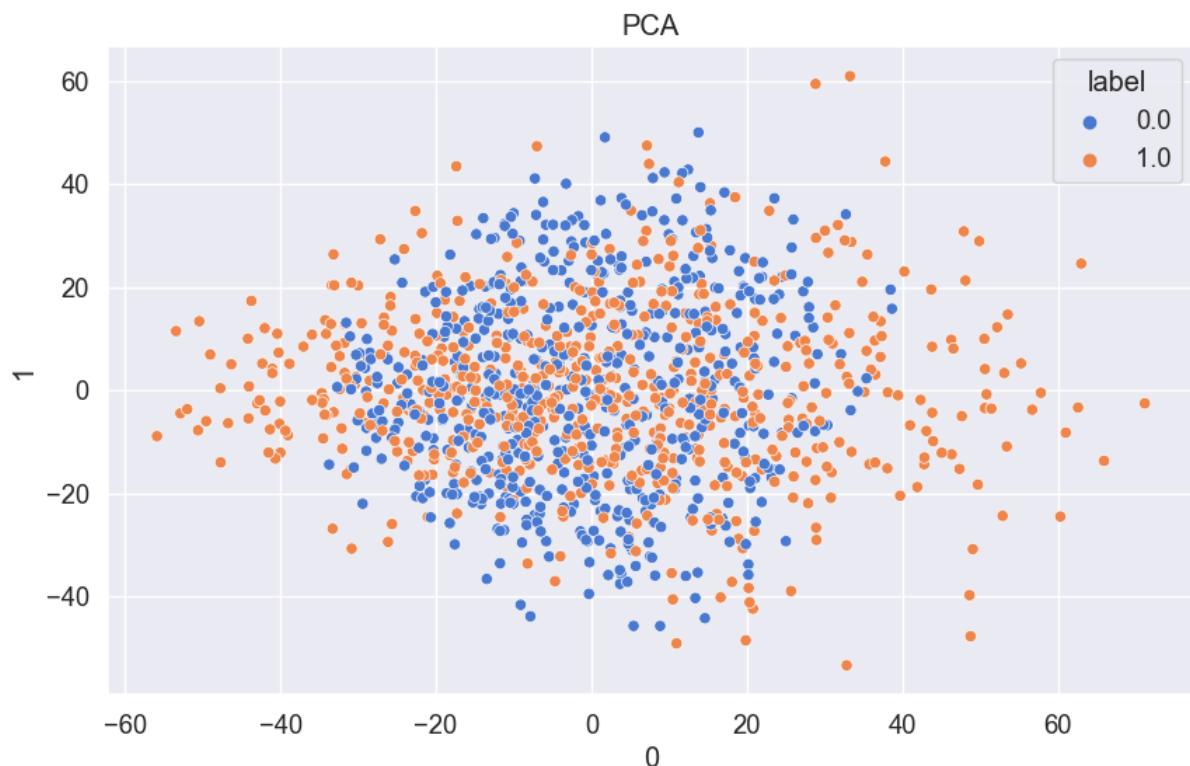
```
Out[312]: ((1258, 2), (1258, 2))
```

```
In [313]: df_pca = pd.DataFrame(union_test_pca)
df_pca["label"] = 1 - union_test_labels
df_pca.head(3)
```

```
Out[313]:
```

	0	1	label
0	-9.067583	-0.783372	0.0
1	-24.326569	20.873564	0.0
2	-9.499015	-1.093014	0.0

```
In [314]: plt.figure(figsize=(10,6))
sns.scatterplot(data=df_pca, hue='label', palette='muted', x=0, y=1).set_title('PCA');
plt.show()
```



Сравнение изображений через TSNE

```
In [315]: tsne = TSNE(n_components=2, learning_rate=150, init='random')
union_test_tsne = tsne.fit_transform(x_test)
union_test_values.shape, union_test_tsne.shape
```

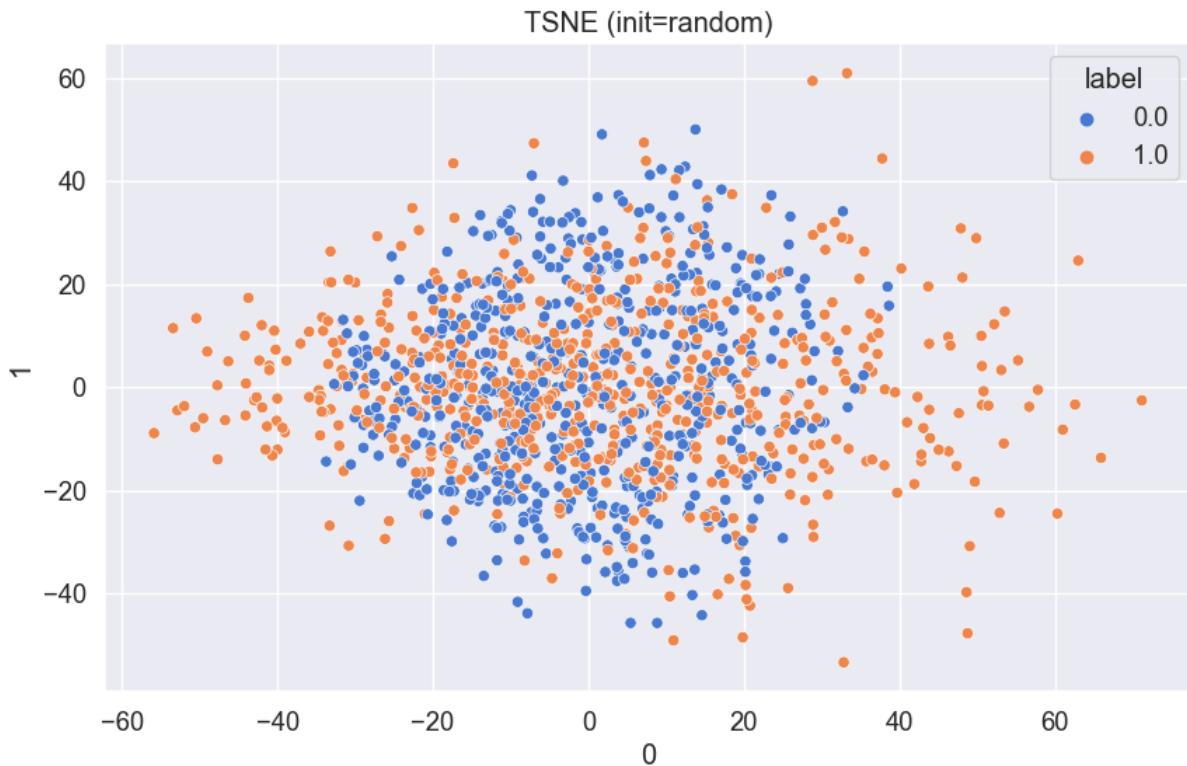
```
Out[315]: ((1258, 12288), (1258, 2))
```

```
In [316]: df_tsne = pd.DataFrame(union_test_tsne)
df_tsne["label"] = 1 - union_test_labels
df_tsne.head(3)
```

```
Out[316]:
```

	0	1	label
0	22.763845	-17.679823	0.0
1	-2.571473	-17.520859	0.0
2	6.701507	9.808678	0.0

```
In [317]: plt.figure(figsize=(10,6))
sns.scatterplot(data=df_pca, hue='label', palette='muted', x=0, y=1).set_title('TSNE (init=random)');
plt.show()
```



Прокомментируйте получившийся результат:

Выход

По итогам визуального анализа можно выделить:

- у реальных изображений стандартные отклонения и средние значения имеют равномерное распределение, а у фейковых сгруппированы довольно плотно в одном месте
- Анализ через выделение главных компонент PCA и через TSNE показывает, что сами "вектора" изображений распределяются довольно равномерно, точнее при уменьшении размерности до двух компонент распределения фейков схожи с реальными. Однако заметно, что есть области, где реальные фото присутствуют но фейковых нет.

В целях развития модели при генерации изображений стоит сделать акцент на распределении средних значений и стандартного отклонения. Это можно сделать например при генерации латентного пространства. Также можно попробовать внедрить дополнительную модель оценки генерированного распределения и распределения реальных фото и результат дополнительной модели можно использовать в лосссе генератора

Общий вывод:

Дополнительно стоит выделить, что один из важнейших гиперпараметров это размер латентного пространства. Слишком маленький размер латентного пространства дает низкое качество, слишком большой размер латентного пространства также не очень хорошо использовать, т.к. при его увеличении также увеличивается время на обучения, при этом в какой-то момент прирост качества уже незначительный. Размер латентного пространства выбирается опытном путем.

При первичном построении GAN очень легко ошибиться, и не всегда по результатам понятно в какой именно модели проблема. Лучшим вариантом решения такого рода задач, это сначала получить минимально работающую модель и только затем накручивать её улучшениями и утежелять.

А так ДЗ очень классное, прям немного магии видится в этом во всем =)