

▼ Autoencoders

В этом ноутбуке мы будем тренировать автоэнкодеры кодировать лица людей. Для этого возьмем следующий датасет: "Labeled Faces in the Wild" (LFW) (<http://vis-www.cs.umass.edu/lfw/>). Код для скачивания и загрузки датасета написан за вас в файле `get_dataset.py`

▼ Vanilla Autoencoder (2 балла)

▼ Prepare the data

```
1 import numpy as np
2 from torch.autograd import Variable
3 import torch.nn as nn
4 import torch.nn.functional as F
5 import torch.optim as optim
6 import torch.utils.data as data_utils
7 import torch
8 import matplotlib.pyplot as plt
9 import PIL
10 from tqdm.notebook import tqdm
11 from IPython.display import clear_output
12 %matplotlib inline

1 from torchvision import transforms
2 from torchvision.datasets import MNIST
```

```
1 device = torch.device('cuda') if torch.cuda.is available() else torch.device('cpu')
```

```

1 # The following line fetches you two datasets: images, usable for autoencoder training and attributes.
2 # Those attributes will be required for the final part of the assignment (applying smiles), so please keep them in mind
3 from get_dataset import fetch_dataset
4 data, attrs = fetch_dataset()

```

```

[ ]> images not found, donwloading...
      extracting...
      done
      attributes not found, downloading...
      done

```

```

1 IMAGE_H = data.shape[1]
2 IMAGE_W = data.shape[2]
3 # у нас цветные изображения
4 N_CHANNELS = 3
5
6 TRAIN_SIZE = 10000
7 VAL_SIZE = data.shape[0] - TRAIN_SIZE
8 LOAD_WEIGHTS = False

```

Разбейте выборку картинок на train и val:

```

1 X_train = data[:TRAIN_SIZE]
2 X_val = data[TRAIN_SIZE:]

```

Напишем вспомогательную функцию, которая будет выводить $n_row \cdot n_col$ первых картинок в массиве images:

```

1 def plot_gallery(images, h, w, n_row=3, n_col=6, channels = 3):
2     """Helper function to plot a gallery of portraits"""
3     plt.figure(figsize=(1.5 * n_col, 1.7 * n_row))
4     plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
5     for i in range(n_row * n_col):

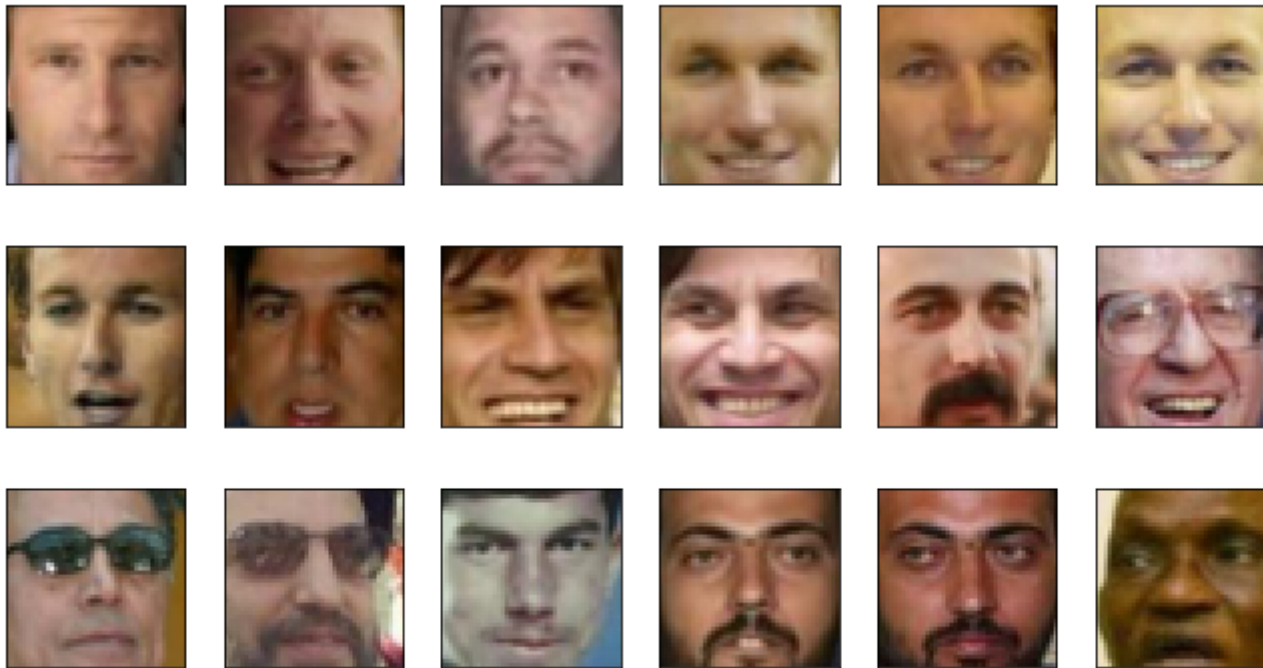
```

```

6     plt.subplot(n_row, n_col, 1 + 1)
7     #try:
8     plt.imshow(images[i].reshape((h, w, channels)), cmap=plt.cm.gray, vmin=-1, vmax=1, interpolation='nearest')
9     plt.xticks(())
10    plt.yticks(())
11    #except:
12    #    pass

```

```
1 plot_gallery(X_train, IMAGE_H, IMAGE_W)
```



Осталось привести картинки к тензорам из PyTorch, чтобы можно было потом скармливать их автоэнкодеру:

```

1 class Data(torch.utils.data.Dataset):
2     def __init__(self, data):
3         super().__init__()
4         self.data = data
5         self.transforms = transforms.Compose([

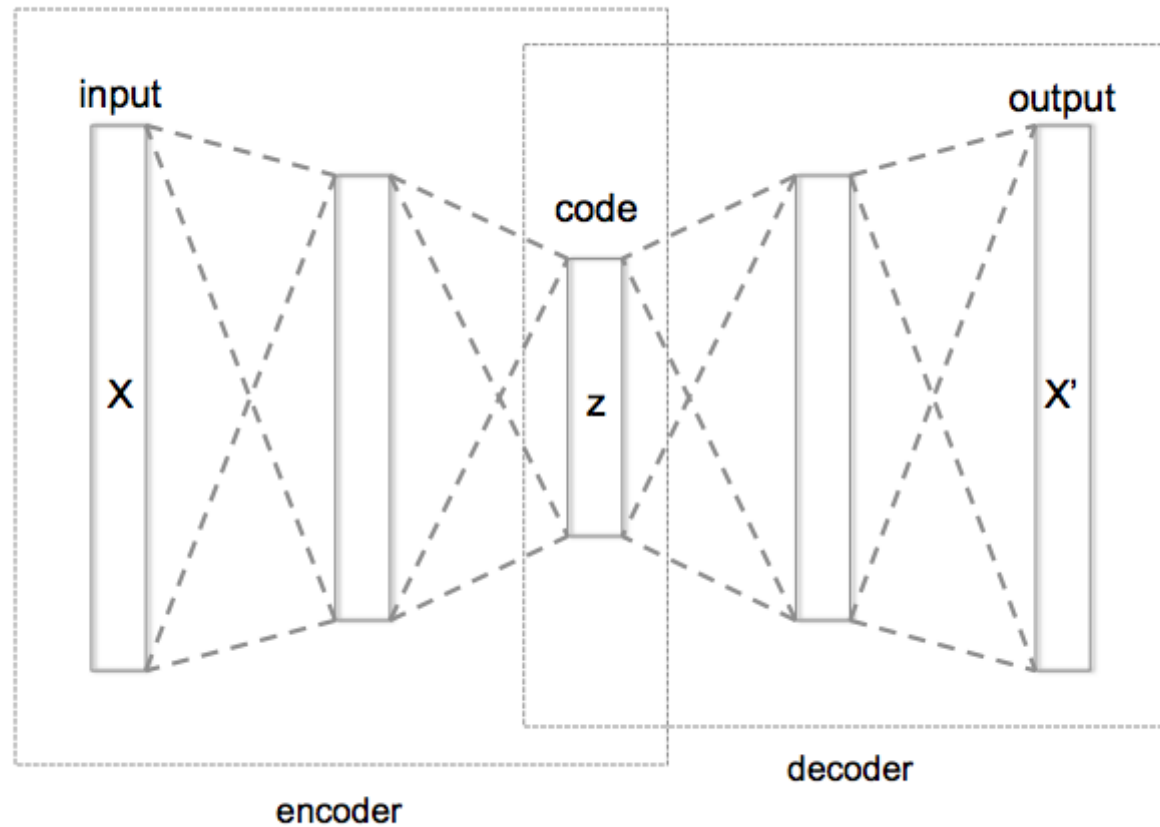
```

```
5     self.transforms = transforms.Compose([
6         transforms.ToTensor()
7     ])
8     def __getitem__(self, index):
9         if isinstance(index, slice):
10             return torch.stack([self.transforms(self.data[ii]) for ii in range(*index.indices(len(self)))])
11         image = self.data[index]
12         image = self.transforms(image)
13         #image = image.permute(1,2,0)
14         return image
15
16     def __len__(self):
17         return len(self.data)


1 Train_dataset = Data(X_train)
2 Val_dataset = Data(X_val)
3 train_loader = data_utils.DataLoader(Train_dataset, 64)
4 Val_loader = data_utils.DataLoader(Val_dataset, 64)
```

▼ Autoencoder

Why to use all this complicated formulaes and regularizations, what is the need for variational inference? To analyze the difference, let's first



train just an autoencoder on the data:

```
1 inp_size=X_train.shape[1]
2 dim_code = 100 # размер латентного вектора, т.е. code
```

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

```
1 from copy import deepcopy
2
3 class Autoencoder(nn.Module):
4     def __init__(self):
5         super(Autoencoder, self).__init__()
6         #определите архитектуры encoder и decoder>
7         self.encoder = nn.Sequential(
8             nn.Conv2d(3,32,3, padding = 1), nn.ReLU(), nn.MaxPool2d(2), nn.BatchNorm2d(32),
9             nn.Conv2d(32,64,3, padding=1), nn.ReLU(), nn.MaxPool2d(2), nn.BatchNorm2d(64),
10            nn.Conv2d(64,100,3, padding=1), nn.ReLU(), nn.MaxPool2d(2), nn.BatchNorm2d(100),
11        )
12
13        self.decoder = nn.Sequential(
14            nn.ConvTranspose2d(100,64,3, stride=2), nn.Conv2d(64,64,3, padding=1), nn.ReLU(), nn.BatchNorm2d(64),
15            nn.ConvTranspose2d(64,32,3, stride=2), nn.Conv2d(32,32,3, padding=1), nn.ReLU(), nn.BatchNorm2d(32),
16            nn.ConvTranspose2d(32,3,3, stride=2), nn.Conv2d(3,3,3), nn.ReLU()
17        )
18
19    def forward(self, x):
20        latent_code = self.encoder(x)
21        reconstruction = self.decoder(latent_code)
22
23        return reconstruction, latent_code


1 criterion = nn.MSELoss()
2
3 autoencoder = Autoencoder()
4 autoencoder.to(device)
5
6 optimizer = torch.optim.Adam(autoencoder.parameters())


1 if LOAD_WEIGHTS:
2     autoencoder.load_state_dict(torch.load('autoencoder'))
```

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 (прогона train выборки через автоэнкодер) можно смотреть, какие реконструкции получились для каких-то изображений val выборки.

Подсказка: если `x_val` -- картинка, а `reconstruction` -- ее реконструкция автоэнкодером, то красиво вывести эту картинку и ее реконструкцию можно с помощью функции `plot_gallery` вот так:

```
plot_gallery([x_val, reconstruction], image_h, image_w, n_row=1, n_col=2)
```

А ну еще было бы неплохо вывести графики train и val losses в процессе тренировки -)

```
1 def show_results(count):
2     X = Val_dataset[:count].to(device)
3     reconstruction = autoencoder(X)[0].detach().permute(0,2,3,1).cpu()
4     X = X.permute(0,2,3,1).cpu()
5     images = list(zip(X,reconstruction))
6     for i in range(count):
7         plot_gallery(images[i],IMAGE_H,IMAGE_W,n_row=1, n_col=2)
```

```
1 def train(model, crit, opt, num_epochs):
2     try:
3         train_losses = []
4         val_losses = []
5         for epoch in tqdm(range(num_epochs)):
6             for stage in ['train', 'val']:
7                 if stage == 'train':
8                     batches_losses = []
9                     for X in train_loader:
10                        X = X.to(device)
11                        opt.zero_grad()
12                        reconstruction = model(X)[0]
13                        loss = crit(reconstruction, X)
14                        loss.backward()
15                        opt.step()
16                        batches_losses.append(loss.item())
17                    train_losses.append(np.mean(batches_losses))
18
19                 elif stage == 'val':
```

```

19     elif stage == 'val':
20         batches_losses = []
21         for y in Val_loader:
22             y = y.to(device)
23             with torch.no_grad():
24                 reconstruction = model(y)[0]
25                 loss = crit(reconstruction,y)
26                 batches_losses.append(loss.item())
27         val_losses.append(np.mean(batches_losses))
28     else:
29         raise KeyError('Wrong stage parameter')
30     return train_losses, val_losses
31 except KeyboardInterrupt:
32     return train_losses, val_losses

```

Увеличить batch size/ Взять первую пикчу в отрисовке

```
1 train_losses, val_losses = train(autoencoder,criterion,optimizer,20)
```

☞ 100% 20/20 [00:59<00:00, 2.97s/it]

```

1 plt.plot(train_losses,label='train losses')
2 plt.plot(val_losses, label = 'val losses')
3 plt.legend()
4 plt.show()

```

☞

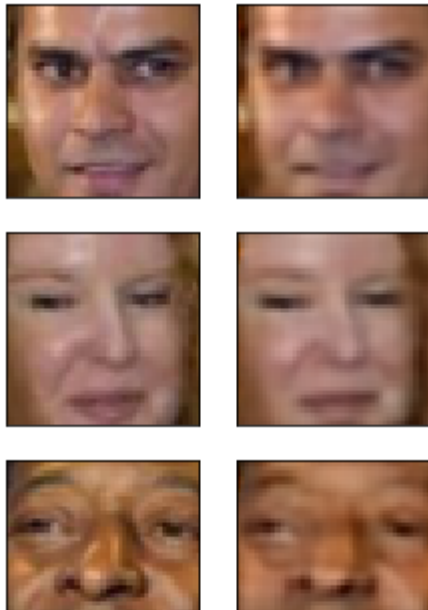


Давайте посмотрим, как наш тренированный автоэкодер кодирует и восстанавливает картинки:

```
1 show_results(7)
```



Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
 Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).



Not bad, right?



▼ Sampling



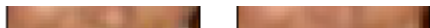
Давайте теперь будем не просто брать картинку, прогонять ее через автоэнкодер и получать реконструкцию, а попробуем создать что-то НОВОЕ

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

▼ If that doesn't work

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

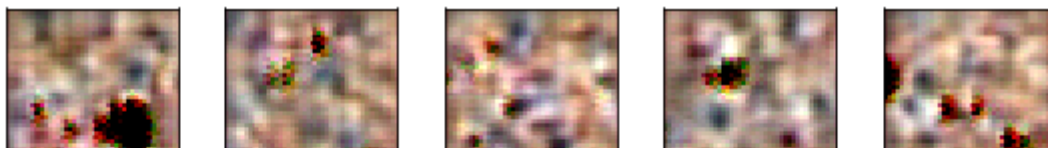
`np.random.randn(25,)`. А чтобы у нас получались лица при записывании вектора декодеру, вектор должен быть распределен так же, как латентные векторы реальных фоток. Так что придется рандом подогнать.



```
1 # сгенерируем 25 случайных векторов размера latent_space
2 z = torch.Tensor(np.random.normal(-0.382, 0.9952, size=(25, dim_code, 5, 5))).to(device)
3 output = autoencoder.decoder(z)
4 plot_gallery(output.data.cpu().permute(0, 2, 3, 1).numpy(), IMAGE_H, IMAGE_W, n_row=5, n_col=5);
```



```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```



▼ Почему не получилось

Для свёрточного автоенкодера нужно не только угадать распределение но и расположение в 100 слоях по 5x5. Я не смог получить что то дельное таким образом



▼ Congrats!

Time to make fun!

Давайте научимся пририсовывать людям улыбки =)



so linear

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



this is you after all

План такой:

1) Нужно выделить "вектор улыбки": для этого нужно из выборки изображений найти несколько (~15 сойдет) 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 грустных

3) Вычислить, собственно, вектор улыбки -- посчитать разность между средним латентным вектором улыбающихся людей и средним латентным вектором грустных людей

3) А теперь приделаем улыбку грустному человеку: добавим полученный в пункте 3 вектор к латентному вектору грустного чувака и прогоним полученный вектор через decoder. Получим того же человека, но уже не грустненького!

```
1 with_smile = attrs[attrs['Smiling']>2.5].index
2 without_smile = attrs[attrs['Mouth Closed']>2.2].index

1 def get_latent_code(model,indexes):
2     latent_code = torch.zeros((100,5,5))
3     tr = transforms.ToTensor()
4     data = [X_train[i] for i in range(len(X_train)) if i in indexes]
5     data = [tr(x).to(device) for x in data]
6     encoder = model.encoder
7     for i in data:
8         latent = encoder(i.unsqueeze(0))
9         latent_code += latent.squeeze().detach().cpu()
10    return latent_code / len(data)

1 latent_smile = get_latent_code(autoencoder,with_smile)
2 latent_without_smile = get_latent_code(autoencoder,without_smile)
3 smile_code = latent_smile-latent_without_smile
```

```
1 def get_photo_changer_function(model, code, alpha = 1):
2     encoder = model.encoder
3     decoder = model.decoder
4     code = code.to(device).unsqueeze(0)
5     def f(image):
6         image = image.to(device)
7         latent = encoder(image)
8         latent += code #* alpha
9         latent = decoder(latent)
10    return latent
```

```
9     image = decoder(latent)
10    image = image.detach().cpu().squeeze().permute(0,2,3,1)
11    return image
12    return f
```

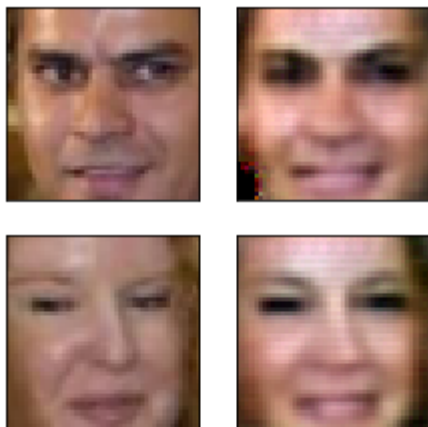
```
1 add_smile = get_photo_changer_function(autoencoder,latent_smile)
```

```
1 def show_changed(count, function):
2     images = Val_dataset[:count]
3     changed_images = function(images)
4     items = list(zip(images.permute(0,2,3,1),changed_images))
5     for x in items:
6         plot_gallery([*x],IMAGE_H,IMAGE_W,n_row=1,n_col=2)
7
```

```
1 show_changed(5,add_smile)
```



Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
 Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
 Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).



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 lwf_deepfinetuned.txt =)



▼ Variational Autoencoder. (2 балла)



Представляю вам проапгрейженную версию автоэнкодеров – вариационные автоэнкодеры.



<https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf>



```
1 class VEncoder(nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.fc1 = nn.Linear(45*45*3, 3000)
5         self.fc2 = nn.Linear(3000, 1500)
```



```

6     self.fc3 = nn.Linear(1500,500)
7     self.fc4_1 = nn.Linear(500,100)
8     self.fc4_2 = nn.Linear(500,100)
9
10    def forward(self,X):
11        X = torch.flatten(X,start_dim=1)
12        X = torch.relu(self.fc1(X))
13        X = torch.relu(self.fc2(X))
14        X = torch.relu(self.fc3(X))
15        mu = torch.tanh(self.fc4_1(X))
16        logsigma = torch.tanh(self.fc4_2(X))
17        return mu, logsigma
18
19

```

```

1 class VDecoder(nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.fc1 = nn.Linear(100,500)
5         self.fc2 = nn.Linear(500,1500)
6         self.fc3 = nn.Linear(1500,3000)
7         self.fc4 = nn.Linear(3000,45*45*3)
8
9     def forward(self,X):
10        X = torch.relu(self.fc1(X))
11        X = torch.relu(self.fc2(X))
12        X = torch.relu(self.fc3(X))
13        reconstruction = torch.sigmoid(self.fc4(X))
14        return reconstruction
15

```

```

1 class VAE(nn.Module):
2     def __init__(self,training = True):
3         super().__init__()
4         self.encoder = VEncoder()
5         self.decoder = VDecoder()
6

```

```

6         self.training = training
7
8     def encode(self, x):
9         mu, logsigma = self.encoder(x)
10        return mu, logsigma
11
12    def gaussian_sampler(self, mu, logsigma):
13        """
14        Функция сэмплирует латентные векторы из нормального распределения с параметрами mu и sigma
15        """
16        if self.training:
17            std = logsigma.exp()
18            eps = std.data.new(std.size()).normal_()
19            return eps.mul(std).add(mu)
20        else:
21            return mu
22
23    def decode(self, z):
24        reconstruction = self.decoder(z)
25        return reconstruction
26
27    def forward(self, x):
28        mu, logsigma = self.encode(x)
29        z = self.gaussian_sampler(mu, logsigma)
30        reconstruction = self.decode(z)
31        return x, mu, logsigma, reconstruction

```

Определим лосс и его компоненты для VAE:

Надеюсь, вы уже прочитали материал в [towardsdatascience](https://towardsdatascience.com/variational-autoencoders-vae-4a1e1e1e1e1e) (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:

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

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

```

1 def KL_divergence(mu, logsigma):
2     """
3     часть функции потерь, которая отвечает за "близость" латентных представлений разных людей
4     """
5     logsigma = logsigma.exp()
6     loss = -0.5*torch.sum(1+torch.log(longsigma.pow(2))-mu.pow(2)-logsigma.pow(2))
7     return loss
8
9 def log_likelihood(x, reconstruction):
10    """
11    часть функции потерь, которая отвечает за качество реконструкции (как mse в обычном autoencoder)
12    """
13    loss = nn.BCELoss(reduction='sum')
14    x = torch.flatten(x,start_dim=1)
15    return loss(reconstruction, x)
16
17 def loss_vae(x, mu, logsigma, reconstruction):
18    return (KL_divergence(mu, logsigma) + log_likelihood(x, reconstruction)).mean()

```

```

1 def VAEtrain(model, crit, opt, num_epochs):
2     try:
3         train_losses = []
4         val_losses = []
5         for epoch in tqdm(range(num_epochs)):
6             for stage in ['train', 'val']:
7                 if stage == 'train':
8                     model.training = True
9                     batches_losses = []
10                    for x in train_loader:

```

```

10         for x in train_loader:
11             X = X.to(device)
12             opt.zero_grad()
13             X, mu, logsigma, reconstruction = model(X)
14             loss = crit(X, mu, logsigma, reconstruction)
15             loss.backward()
16             opt.step()
17             batches_losses.append(np.mean(loss.item()))
18         train_losses.append(np.mean(batches_losses))
19
20     elif stage == 'val':
21         model.training = False
22         batches_losses = []
23         for y in Val_loader:
24             y = y.to(device)
25             with torch.no_grad():
26                 y, mu, logsigma, reconstruction = model(y)
27                 loss = crit(y, mu, logsigma, reconstruction)
28                 batches_losses.append(np.mean(loss.item()))
29         val_losses.append(np.mean(batches_losses))
30     else:
31         raise KeyError('Wrong stage parameter')
32     return train_losses, val_losses
33 except KeyboardInterrupt:
34     return train_losses, val_losses

```

```

1 criterion = loss_vae
2
3 Vautoencoder = VAE().to(device)
4
5 optimizer = torch.optim.Adam(Vautoencoder.parameters(), lr=0.0001)

```

```

1 if LOAD_WEIGHTS:
2     Vautoencoder.load_state_dict(torch.load('Vautoencoder'))

```

И обучим модель:

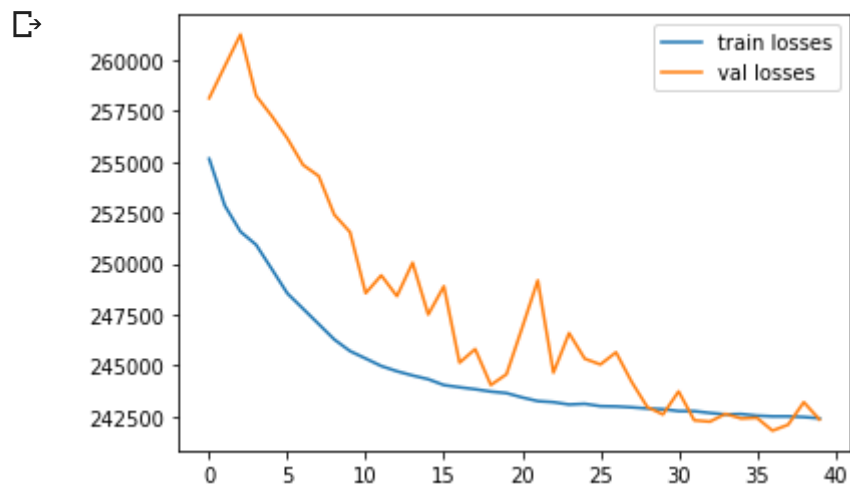
```
1 Train_dataset = Data(X_train)
2 Val_dataset = Data(X_val)
3 train_loader = data_utils.DataLoader(Train_dataset,64)
4 Val_loader = data_utils.DataLoader(Val_dataset,64)

1 train_losses = []
2 val_losses = []

1 train_losses, val_losses = VAEtrain(Vautoencoder,criterion,optimizer,40)
```

100% 40/40 [04:53<00:00, 7.33s/it]

```
1 plt.plot(train_losses,label='train losses')
2 plt.plot(val_losses, label = 'val losses')
3 plt.legend()
4 plt.show()
```

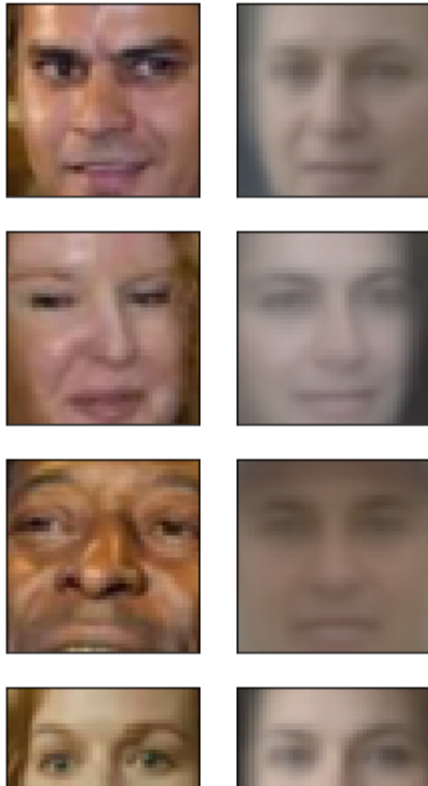


Давайте посмотрим, как наш тренированный VAE кодирует и восстанавливает картинки:

```
1 def show_results_VAE(count):
2     X = Val_dataset[:count].to(device)
3     autoencoder.training=True
4     reconstruction = Vautoencoder(X)[3].detach().cpu().reshape(-1,3,IMAGE_H,IMAGE_W)
5     reconstruction = reconstruction.permute(0,2,3,1)
6     X = X.permute(0,2,3,1).cpu()
7     images = list(zip(X,reconstruction))
8     for i in range(count):
9         plot_gallery(images[i],IMAGE_H,IMAGE_W,n_row=1, n_col=2)
10
```

```
1 show_results_VAE(7)
```





And finally sample from VAE.



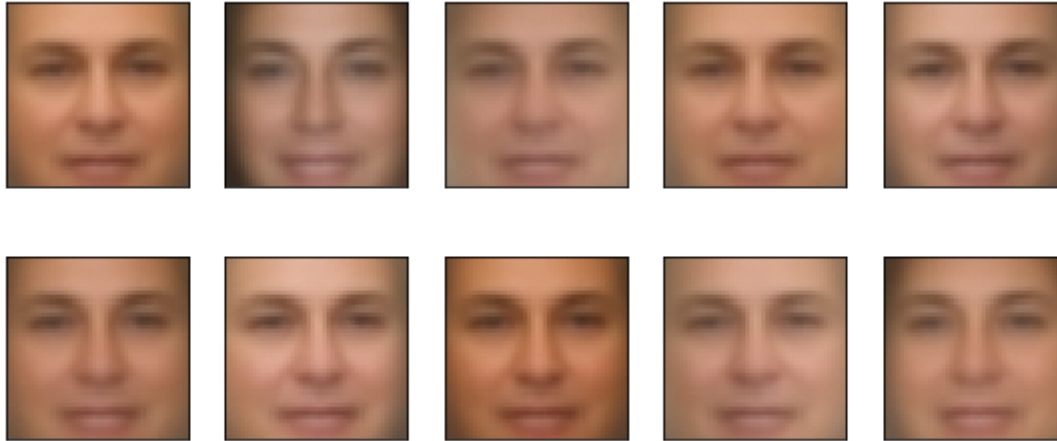
▼ Sampling



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:



```
1 # вспомните про замечание из этого же пункта обычного AE про распределение латентных переменных
2 z = torch.Tensor(np.array([np.random.normal(0, 1, 100) for i in range(10)])).to(device)
3 output = Vautoencoder.decode(z).detach().cpu().reshape(-1,3,IMAGE_H,IMAGE_W).permute(0,2,3,1)
4 plot_gallery(output.data.cpu().numpy(), IMAGE_H, IMAGE_W, n_row=2, n_col=5)
```



▼ Latent Representation

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 лиц в пространстве.

Плюс давайте сделаем такую вещь: у вас есть файл с атрибутами `lwf_deepfinetuned.txt`, который скачался вместе с базой картинок. Там для каждой картинке описаны атрибуты картинки (имя человека, его пол, цвет кожи и т.п.). Когда будете визуализировать точки латентного пространства на картинке, возьмите какой-нибудь атрибут и покрасьте точки в соответствии со значением атрибута, соответствующего этой точке.

Например, возьмем атрибут "пол". Давайте покрасим точки, которые соответствуют картинкам женщин, в один цвет, а точки, которые соответствуют картинкам мужчин – в другой.

Подсказка: красить – это просто =) У `plt.scatter` есть параметр `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 с разными атрибутами.

```
1 from sklearn.manifold import TSNE

1 def latent_Representation(Vautoencoder, data):
2     data = torch.Tensor(data).to(device).permute(0,3,1,2)
3     mu, logsigma = Vautoencoder.encode(data)
4     mu = mu.detach().cpu()
5     logsigma = logsigma.detach().cpu()
6     mu = Vautoencoder.gaussian_sampler(mu,logsigma)
7     X_embedded = TSNE(n_components=2).fit_transform(mu)
8     return X_embedded
9
```

```
1 def get_mask(labels,parameter, border):
2     mask = labels[parameter] >= border
3     return mask
```

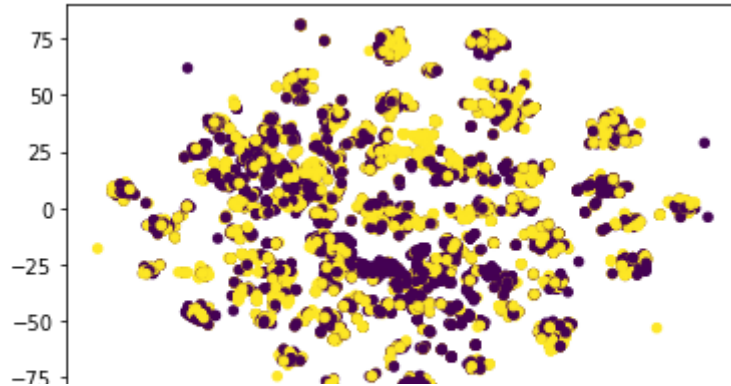
```
1 Emb = latent_Representation(Vautoencoder,X_train)
```

```
1 mask = get_mask(attrs, 'Male',0.7)[:10000]
2 X = Emb[:,0]
3 Y = Emb[:,1]
```

```
1 plt.scatter(X,Y,c=mask,s=20)
```



```
<matplotlib.collections.PathCollection at 0x7f883db58358>
```



Что вы думаете о виде латентного представления?

-- -- -- -- --

Енкодер работает плохо. Чёткая граница не наблюдается

Congrats v2.0!

▼ Conditional VAE (2 балла)

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 случайные лица). И вот мне понадобилось сгенерировать цифру 8. И я подставляю разные варианты шума, и все никак не генерится восьмерка -- у меня получаются то пятерки, то тройки, то четверки. Гадость!

Хотелось бы добавить к нашему АЕ функцию "выдай мне пожалуйста случайное число из вот этого вот класса", где классов десять (цифры от 0 до 9 образуют десять классов). Типа я такая говорю "выдай мне случайную восьмерку" и оно генерит случайную восьмерку!

Conditional AE – так называется вид автоэнкодера, который предоставляет такую возможность. Ну, название "conditional" уже говорит само за себя.

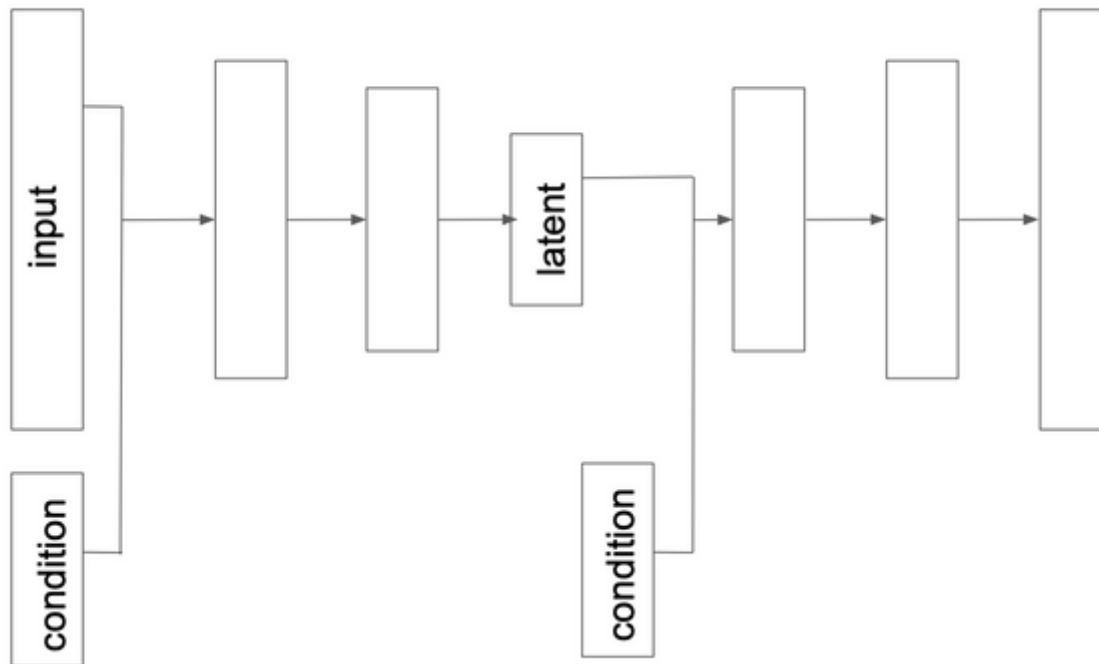
И в этой части проекта мы научимся такие обучать.

▼ Архитектура

На картинке ниже представлена архитектура простого Conditional AE.

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.

То есть, в первый (входной) слой энкодера есть конкатенация картинки и информации о классе (например, вектора из девяти нулей и одной единицы). Первый слой декодера есть конкатенация латентного вектора и информации о классе.



На всякий случай: это VAE, то есть, latent у него состоит из μ и σ все еще.

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

P.S.

Можно передавать класс картинки не только в первый слой, но и в каждый слой сети. То есть на каждом слое конкатенировать выход из предыдущего слоя и информацию о классе.

▼ Датасет

Здесь я предлагаю вам два варианта. Один попроще, другой -- посложнее, но поинтереснее =)

1. Использовать датасет MNIST (<http://yann.lecun.com/exdb/mnist/>). Обучать conditional VAE на этом датасете, condition -- класс цифры.
2. Использовать датасет лиц, с которым мы игрались выше. Condition -- пол/раса/улыбки/whatever из lfw_deepfinetuned.txt.

Почему второй вариант "посложнее" -- потому что я сама еще не знаю, получится ли такой CVAE с лицами или нет =) Вы -- исследователи! (не ну это же проект, так и должно быть)

```
1 mnist_data_train = MNIST('mnist/',download=True);
2 mnist_data_val = MNIST('mnist/',train=False ,download=True);
```

```
1 class CVAE_dataset(torch.utils.data.Dataset):
2     def __init__(self, data,numClasses):
3         super().__init__()
4         self.num_classes = numClasses
5         self.data = data
```

```
6     self.transforms = transforms.Compose([
7         transforms.ToTensor()
8     ])
9
10    def __getitem__(self, index):
11        image, number = self.data[index]
12        one_hot = torch.zeros(self.num_classes)
13        one_hot[number] = 1
14        image = self.transforms(image)
15        return image, one_hot
16
17    def __len__(self):
18        return len(self.data)
19
20
21    1 Train_dataset = CVAE_dataset(mnist_data_train,10)
22    2 Val_dataset = CVAE_dataset(mnist_data_val,10)
23    3 train_loader = data_utils.DataLoader(Train_dataset,64,drop_last=True)
24    4 Val_loader = data_utils.DataLoader(Val_dataset,64)
25
26
27    1 class VEncoder(nn.Module):
28    2     def __init__(self):
29    3         super().__init__()
30    4         self.fc1 = nn.Linear(28*28+10,512)
31    5         self.fc2 = nn.Linear(512,256)
32    6         self.fc3 = nn.Linear(256,128)
33    7         self.fc4_1 = nn.Linear(128,100)
34    8         self.fc4_2 = nn.Linear(128,100)
35    9
36    10    def forward(self,X, cls):
37    11        X = torch.flatten(X,start_dim=1)
38    12        X = torch.cat((X, cls),dim=1)
39    13        X = torch.relu(self.fc1(X))
40    14        X = torch.relu(self.fc2(X))
41    15        X = torch.relu(self.fc3(X))
42    16        mu = torch.tanh(self.fc4_1(X))
```

```
17     logsigma = torch.tanh(self.fc4_2(X))
18     return mu, logsigma
19
20
```



```
1 class VDecoder(nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.fc1 = nn.Linear(110,128)
5         self.fc2 = nn.Linear(128,256)
6         self.fc3 = nn.Linear(256,512)
7         self.fc4 = nn.Linear(512,28*28)
8
9     def forward(self,X, cls):
10         X = torch.cat((X,cls),dim=1)
11         X = torch.relu(self.fc1(X))
12         X = torch.relu(self.fc2(X))
13         X = torch.relu(self.fc3(X))
14         reconstruction = torch.sigmoid(self.fc4(X))
15         return reconstruction
16
```

```
1 class CVAE(nn.Module):
2     def __init__(self,training = True):
3         super().__init__()
4         self.encoder = VEncoder()
5         self.decoder = VDecoder()
6         self.training = training
7
8     def encode(self, x, cls):
9         mu, logsigma = self.encoder(x, cls)
10         return mu, logsigma
11
12     def gaussian_sampler(self, mu, logsigma):
```

```

13     """
14     Функция сэмпليрует латентные векторы из нормального распределения с параметрами  $\mu$  и  $\sigma$ 
15     """
16     if self.training:
17         std = logsigma.exp()
18         eps = std.data.new(std.size()).normal_()
19         return eps.mul(std).add(mu)
20     else:
21         return mu
22
23     def decode(self, z, cls):
24         reconstruction = self.decoder(z, cls)
25         return reconstruction
26
27     def forward(self, x, cls):
28         mu, logsigma = self.encode(x, cls)
29         z = self.gaussian_sampler(mu, logsigma)
30         reconstruction = self.decode(z, cls)
31         return x, mu, logsigma, reconstruction
32

```

```

1 def CVAEtrain(model, crit, opt, num_epochs):
2     try:
3         train_losses = []
4         val_losses = []
5         for epoch in tqdm(range(num_epochs)):
6             for stage in ['train', 'val']:
7                 if stage == 'train':
8                     model.training = True
9                     batches_losses = []
10                    for X, cls in train_loader:
11                        X = X.to(device)
12                        cls = cls.to(device)
13                        opt.zero_grad()
14                        X, mu, logsigma, reconstruction = model(X, cls)
15                        loss = crit(X, mu, logsigma, reconstruction)
16                        loss.backward()

```

```


17         opt.step()
18         batches_losses.append(np.mean(loss.item()))
19     train_losses.append(np.mean(batches_losses))
20
21     elif stage == 'val':
22         model.training = False
23         batches_losses = []
24         for y, cls in Val_loader:
25             y = y.to(device)
26             cls = cls.to(device)
27             with torch.no_grad():
28                 y, mu, logsigma, reconstruction = model(y, cls)
29                 loss = crit(y, mu, logsigma, reconstruction)
30                 batches_losses.append(np.mean(loss.item()))
31         val_losses.append(np.mean(batches_losses))
32     else:
33         raise KeyError('Wrong stage parameter')
34     return train_losses, val_losses
35 except KeyboardInterrupt:
36     return train_losses, val_losses

1 criterion = loss_vae
2
3 Cautoencoder = CVAE().to(device)
4
5 optimizer = torch.optim.Adam(Cautoencoder.parameters(),lr=0.0001)

1 if LOAD_WEIGHTS:
2     Cautoencoder.load_state_dict(torch.load('Cautoencoder'))

1 train_losses, val_losses = CVAEtrain(Cautoencoder,criterion,optimizer,40)

```

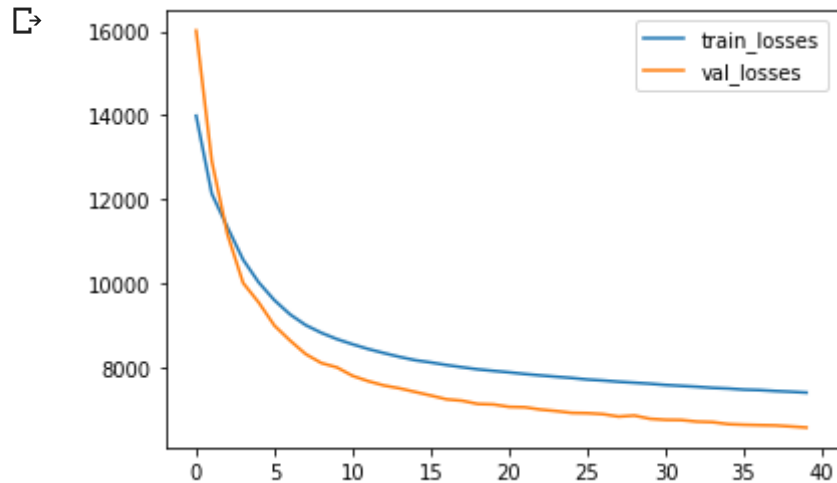
 100%

40/40 [06:33<00:00, 9.85s/it]


```

1 plt.plot(train_losses,label = 'train_losses')
2 plt.plot(val_losses, label = 'val_losses')
3 plt.legend()
4 plt.show()

```



```

1 def C_plot_gallery(images, h, w, n_row=3, n_col=6):
2     """Helper function to plot a gallery of portraits"""
3     plt.figure(figsize=(1.5 * n_col, 1.7 * n_row))
4     plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
5     for i in range(n_row * n_col):
6         plt.subplot(n_row, n_col, i + 1)
7         #try:
8         plt.imshow(images[i].reshape((h, w)), cmap=plt.cm.gray, vmin=-1, vmax=1, interpolation='nearest')
9         plt.xticks(())
10        plt.yticks(())
11        #except:
12        #    pass

```

```

1 def show_results_CVAE(count):
2
3     X, cls = Val_dataset[:count].to(device)
4     # Counterdown training Time

```

```

4  cautoencoder.training=True
5  reconstruction = Cautoencoder(X,cls)[3].detach().cpu().reshape(-1,1,28,28)
6  reconstruction = reconstruction.permute(0,2,3,1)
7  X = X.permute(0,2,3,1).cpu()
8  images = list(zip(X,reconstruction))
9  for i in range(count):
10     plot_gallery(images[i],IMAGE_H,IMAGE_W,n_row=1, n_col=2)
11

```

▼ 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 картинки цифры 5 и 7, а для лиц людей – восстановить лицо улыбающегося и хмурого человека или лица людей разного пола (смотря на чем был ваш кондишен)

```

1 cls = 8
2
3 one_hot = torch.zeros((10,10))
4 one_hot[cls]=1
5 one_hot = one_hot.to(device)
6 z = torch.Tensor(np.array([np.random.normal(0, 1, 100) for i in range(10)])).to(device)
7 output = Cautoencoder.decode(z,one_hot).detach().cpu().reshape(-1,1,28,28).permute(0,2,3,1).squeeze()
8 C_plot_gallery(output.data.cpu().numpy(), 28, 28, n_row=2, n_col=5)

```





Splendid! Вы великолепны!

Ну круто же, ну?



▼ 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.

```
1 from sklearn.manifold import TSNE

1 def C_latent_Representation(Cautoencoder):
2     latent_space = []
3     numbers = []
4     tf = transforms.ToTensor()
5     for x, cls in train_loader:
6         cls = cls.to(device)
7         x = x.to(device)
8         mu, logsigma = Cautoencoder.encode(x, cls)
9         mu = mu.detach().cpu()
10        logsigma = logsigma.detach().cpu()
11        mu = Cautoencoder.gaussian_sampler(mu, logsigma)
12        mu = torch.reshape(mu, (-1, 100))
13        mu = torch.cat((mu, cls.cpu()), dim = 1) #13123123
14        cls = cls.reshape(-1, 10)
15        latent_space.append(mu.data)
16        numbers.append(cls.data)
17    latent_space = torch.cat(latent_space).reshape(-1, 110)
```

```

17 latent_space = torch.stack(latent_space).reshape(-1,110)
18 numbers = torch.stack(numbers).reshape(-1,10).squeeze().cpu()
19 numbers = [(e1 == 1).nonzero().squeeze() for e1 in numbers]
20 return latent_space, numbers
21

```

```

1 latent_space, numbers = C_latent_Representation(Cautoencoder)

```

```

1 sp = np.array(latent_space)

```

```

1 raise AssertionError

```

```

1 X_embedded = TSNE(n_components=2,verbose=3).fit_transform(sp) #Осторожно, очень долго

```

```

1 X = X_embedded[:,0]

```

```

2 Y = X_embedded[:,1]

```

```

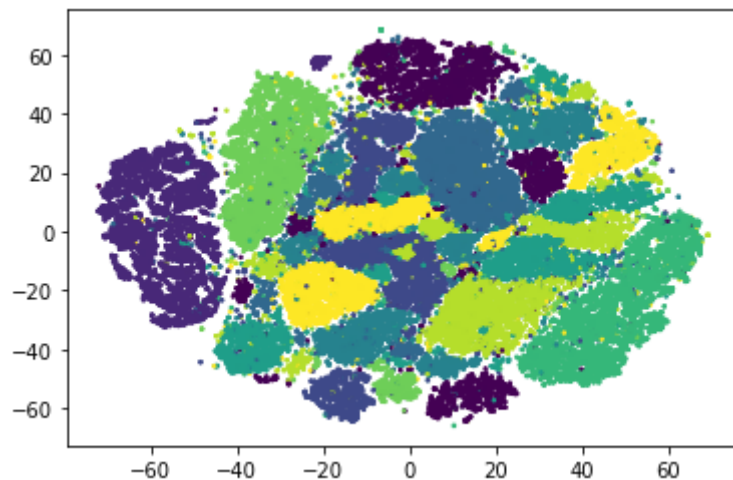
1 plt.scatter(X,Y,c=numbers,s=2)

```

```

↳ <matplotlib.collections.PathCollection at 0x7f8820a8cb70>

```



Уже видны границы но распределение латентных факторов по прежнему не идеальное

▼ BONUS 1: Image Morphing (1 балл)



Предлагаю вам поиграться не только с улыбками, но и с получением из одного человека другого!

План:

1. Берем две картинки разных людей из датасета
2. Получаем их латентные представления X и Y
3. Складываем латентные представления с коэффициентом α :

$$\alpha X + (1 - \alpha)Y$$

где α принимает несколько значений от 0 до 1

4. Визуализируем, как один человек превращается в другого!

```
1 Train_dataset = Data(X_train)
2 Val_dataset = Data(X_val)
3 train_loader = data_utils.DataLoader(Ttrain_dataset, 64)
4 Val_loader = data_utils.DataLoader(Val_dataset, 64)
```

```
1 batch = iter(train_loader).next()
2 Image_a = batch[0].unsqueeze(0).to(device)
3 Image_b = batch[5].unsqueeze(0).to(device)
4 latent_a = autoencoder.encoder(Image_a)
5 latent_b = autoencoder.encoder(Image_b)
```

```
6 a = [0,0.2,0.4,0.6,0.8,1]
7 latent = [k*latent_a +(1-k)*latent_b for k in a]
8 Images = [autoencoder.decoder(lat).detach().cpu().permute(0,2,3,1) for lat in latent]
```

```
1 plot_gallery(Images, IMAGE_H, IMAGE_W, n_row=1, n_col=6)
```

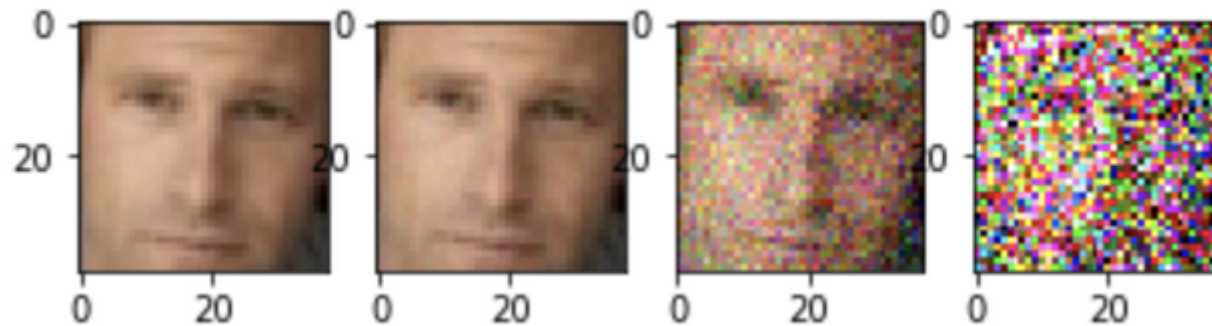
↳ Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).



▼ BONUS 2: Denoising (2 балла)

У автоэнкодеров, кроме сжатия и генерации изображений, есть другие практические применения. Про одно из них это бонусное задание.

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 AE останется той же (MSE между реальной картинкой и выданной), а на вход автоэнкодеру будет подаваться зашумленная картинка.



Для того, чтобы поставить эксперимент, нужно взять ваш любимый датасет (датасет лиц или MSE с прошлых заданий или любой другой) и сделать копию этого датасета с шумом.

В питоне шум можно добавить так:

```
1 class NoisyDataset(data_utils.Dataset):
2     def __init__(self, data, noise_factor):
3         super().__init__()
4         self.data = data
5         self.transforms = transforms.Compose([
6             transforms.ToTensor()
7         ])
8         self.noise_factor = noise_factor
9     def __getitem__(self, index):
10        image = self.data[index]
11        noisy_image = image + self.noise_factor * np.random.normal(loc=0.0, scale=30.0, size=image.shape)
12        image = self.transforms(image).float()
13        noisy_image = self.transforms(noisy_image)/255
14        noisy_image.requires_grad=False
15        noisy_image=noisy_image.float()
16        return noisy_image, image
17
18    def len(self):
```

```
18 def __len__(self):
```

```
19     return len(self.data)
```

```
1 noisy_train = NoisyDataset(X_train,0.5)
```

```
2 noisy_val = NoisyDataset(X_val,0.5)
```

```
3 train_loader = data_utils.DataLoader(noisy_train,batch_size=32)
```

```
4 Val_loader = data_utils.DataLoader(noisy_val,batch_size=32)
```

```
1 criterion = nn.MSELoss()
```

```
2
```

```
3 Dautoencoder = Autoencoder()
```

```
4 Dautoencoder.to(device)
```

```
5
```

```
6 optimizer = torch.optim.Adam(Dautoencoder.parameters())
```

```
1 if LOAD_WEIGHTS:
```

```
2     Dautoencoder.load_state_dict(torch.load('Dautoencoder'))
```

```
1 def Dtrain(model, crit, opt, num_epochs):
```

```
2     try:
```

```
3         train_losses = []
```

```
4         val_losses = []
```

```
5         for epoch in tqdm(range(num_epochs)):
```

```
6             for stage in ['train', 'val']:
```

```
7                 if stage == 'train':
```

```
8                     batches_losses = []
```

```
9                     for X, z in train_loader:
```

```
10                         X = X.to(device)
```

```
11                         z = z.to(device)
```

```
12                         opt.zero_grad()
```

```
13                         reconstruction = model(X)[0]
```

```
14                         loss = crit(reconstruction, z)
```

```
15                         loss.backward()
```

```
16                         opt.step()
```

```
17                         batches_losses.append(loss.item())
```

```
18     train_losses.append(np.mean(batches_losses))
```



```
18         train_losses.append(np.mean(batches_losses))
19
20     elif stage == 'val':
21         batches_losses = []
22         for y, z in Val_loader:
23             y = y.to(device)
24             z = z.to(device)
25             with torch.no_grad():
26                 reconstruction = model(y)[0]
27                 loss = crit(reconstruction,z)
28                 batches_losses.append(loss.item())
29         val_losses.append(np.mean(batches_losses))
30     else:
31         raise KeyError('Wrong stage parameter')
32     return train_losses, val_losses
33 except KeyboardInterrupt:
34     return train_losses, val_losses
```

```
1 train_losses, val_losses = Dtrain(Dautoencoder,criterion,optimizer,20)
```

↳ 100% 20/20 [07:07<00:00, 21.39s/it]

```
1 plt.plot(train_losses,label='train_losses')
2 plt.plot(val_losses,label='val_losses')
3 plt.legend()
4 plt.show()
```

↳



```

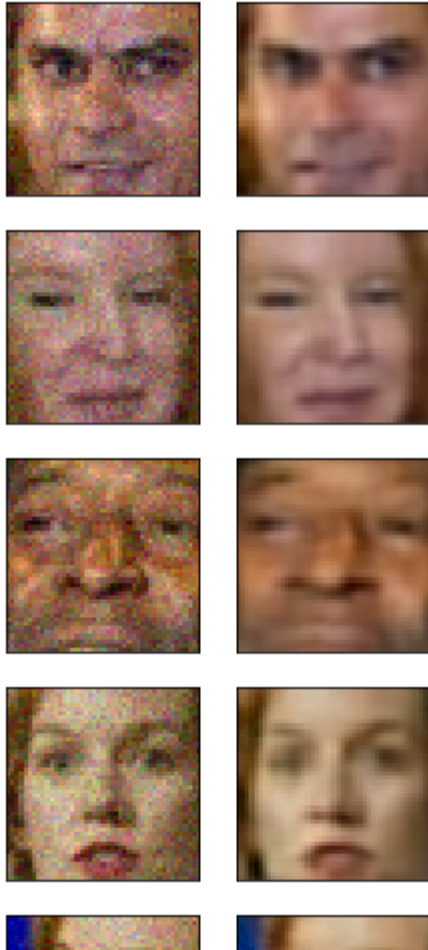
1 def d_show(model,count):
2     for x, z in Val_loader:
3         x = x.to(device)
4         z = z.to(device)
5         reconstruction = model(x)[0]
6         r_images = reconstruction.detach().cpu().permute(0,2,3,1)
7         x_images = x.detach().cpu().permute(0,2,3,1)
8         z_images = z.detach().cpu().permute(0,2,3,1)
9         images = list(zip(x_images,r_images))
10        break
11    for c, i in enumerate(images):
12        plot_gallery(i,IMAGE_H,IMAGE_W,n_row=1,n_col=2)
13        if c== count:
14            break
15
16

```

```
1 d_show(Dautoencoder,5)
```



Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).



▼ Бонус 2.1: Occlusion (+еще 1 балл)





Автоэнкодерами можно не только убирать шум, но и восстанавливать части картинки, которые чем-то закрыты!

Эксперимент здесь такой: вместо наложения шума на картинку, "закрываем" часть картинки заплаткой и тренируем AE/VAE восстанавливать закрытую часть картинки.

Важно, чтобы заплатка была не очень большая.

```
1 class OcclusionDataset(data_utils.Dataset):
2     def __init__(self, data, noise_factor):
3         super().__init__()
4         self.data = data
5         self.transforms1 = transforms.Compose([
6             transforms.ToTensor()
7         ])
8         self.transforms2 = transforms.Compose([
9             transforms.ToTensor(),
10            transforms.RandomErasing(p=1)
11        ])
```

```

11     ])
12     def __getitem__(self, index):
13         data = self.data[index]
14         image = self.transforms1(data)
15         oc = self.transforms2(data)
16         return oc, image
17
18     def __len__(self):
19         return len(self.data)

1 noisy_train = OcclusionDataset(X_train, 0.5)
2 noisy_val = OcclusionDataset(X_val, 0.5)
3 train_loader = data_utils.DataLoader(noisy_train, batch_size=32)
4 Val_loader = data_utils.DataLoader(noisy_val, batch_size=32)

1 def Otrain(model, crit, opt, num_epochs):
2     try:
3         train_losses = []
4         val_losses = []
5         for epoch in tqdm(range(num_epochs)):
6             for stage in ['train', 'val']:
7                 if stage == 'train':
8                     batches_losses = []
9                     for X, z in train_loader:
10                         X = X.to(device)
11                         z = z.to(device)
12                         opt.zero_grad()
13                         reconstruction = model(X)[0]
14                         loss = crit(reconstruction, z)
15                         loss.backward()
16                         opt.step()
17                         batches_losses.append(loss.item())
18                     train_losses.append(np.mean(batches_losses))
19
20                 elif stage == 'val':
21                     batches_losses = []
22                     for v, z in Val_loader:

```

```

22         for y, z in val_loader:
23             y = y.to(device)
24             z = z.to(device)
25             with torch.no_grad():
26                 reconstruction = model(y)[0]
27                 loss = crit(reconstruction, z)
28                 batches_losses.append(loss.item())
29             val_losses.append(np.mean(batches_losses))
30         else:
31             raise KeyError('Wrong stage parameter')
32     return train_losses, val_losses
33 except KeyboardInterrupt:
34     return train_losses, val_losses

```

```

1 criterion = nn.MSELoss()
2
3 Oautoencoder = Autoencoder()
4 Oautoencoder.to(device)
5
6 optimizer = torch.optim.Adam(Oautoencoder.parameters())

```

```

1 if LOAD_WEIGHTS:
2     Oautoencoder.load_state_dict(torch.load('Oautoencoder'))

```

```

1 train_losses, val_losses = Otrain(Oautoencoder, criterion, optimizer, 20)

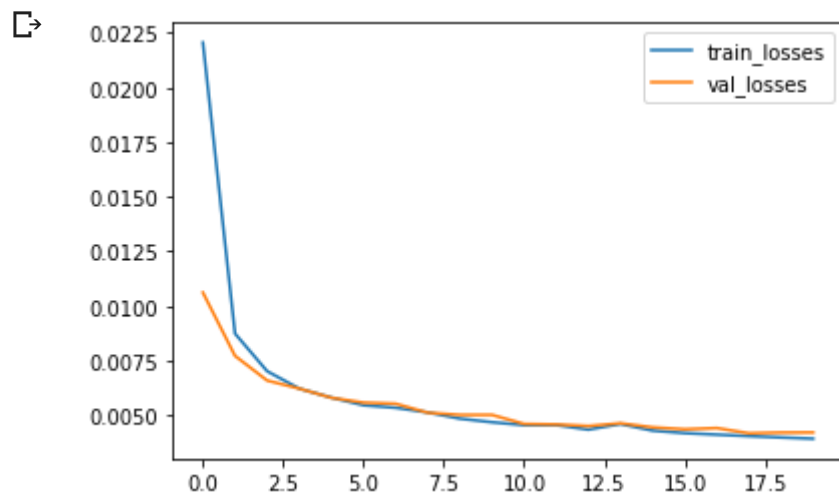
```

📄 100% 20/20 [02:57<00:00, 8.88s/it]

```

1 plt.plot(train_losses, label='train_losses')
2 plt.plot(val_losses, label='val_losses')
3 plt.legend()
4 plt.show()

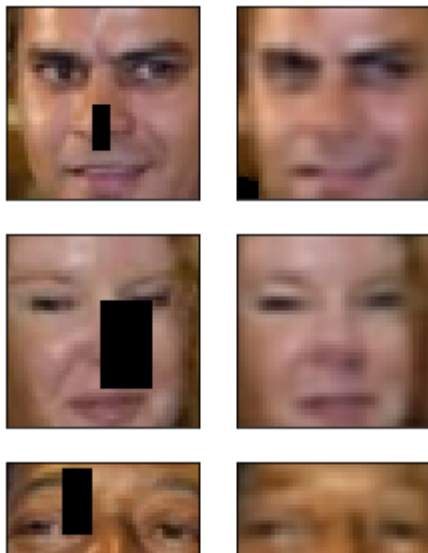
```



```
1 d_show(0autoencoder,5)
```



Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).



▼ Bonus 3: Image Retrieval (2 балла)



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 близжайшие!



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 ближайшие из латентных представлений тренировочной базы
5. Находим лица тренировочного датасета, которым соответствуют ближайшие латентные представления, визуализируем!

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

```
1 codes = [autoencoder.encoder(torch.Tensor(x).to(device).unsqueeze(0).permute(0,3,1,2)).detach().cpu() for x in X_train]

1 codes = torch.stack(codes).squeeze()

1 codes = codes.reshape(-1,2500)

1 # обучаем LSHForest
2 from sklearn.neighbors import NearestNeighbors
3 NN = NearestNeighbors().fit(codes)

1 def get_similar(image, n_neighbors=5):
2     # функция, которая берет тестовый image и с помощью метода kneighbours у NearestNeighbors ищет ближайшие векторы
3     # прогоняет векторы через декодер и получает картинки ближайших людей
4
5     code = autoencoder.encoder(torch.Tensor(image).unsqueeze(0).permute(0,3,1,2).to(device)).detach().cpu().reshape(-1,2500)
6
7     (distances,)(idx,) = NN.kneighbors(code, n_neighbors=n_neighbors)
8
9     return distances, X_train[idx]

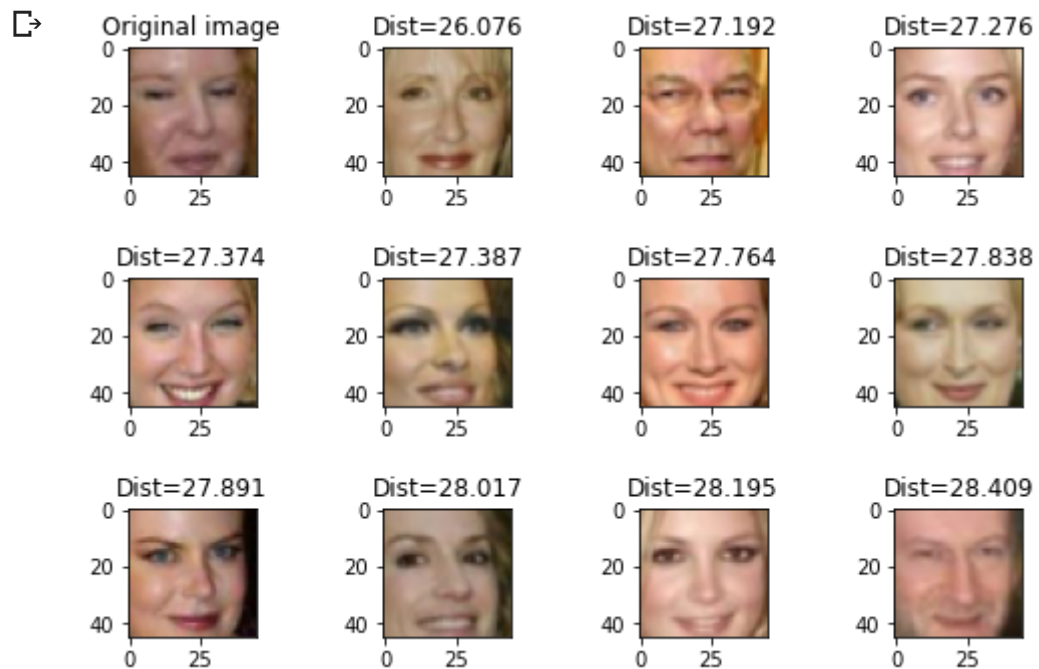
1 def show_similar(image):
2
3     # функция, которая принимает тестовый image, ищет ближайшие к нему и визуализирует результат
4
5     distances,neighbors = get_similar(image,n_neighbors=11)
6
7     plt.figure(figsize=[8,6])
8     plt.subplots_adjust(wspace=1)
9     plt.subplot(3,4,1)
10    plt.imshow(image)
11    plt.title("Original image")
```

```

12
13     for i in range(11):
14         plt.subplot(3,4,i+2)
15         plt.imshow(neighbors[i])
16         plt.title("Dist=%.3f"%distances[i])
17     plt.show()

```

```
1 show_similar(X_val[1])
```



▼ Эпилог

здесь мы рассмотрели не все применения автоэнкодеров. Еще есть, например:

-- поиск аномалий -- дополнение отсутствующих частей картины -- работа с sequential данными (например, временными рядами) -- гибриды ГАН+АЕ, которые активно изучаются в последнее время -- использование латентных переменных АЕ в качестве фичей ...

Они не были частью этого проекта, потому что для их реализации пришлось бы больше возиться с датасетами.

Но! Если вы хотите, вы, конечно, всегда можете реализовать еще что-то и получить за это еще допбаллы.

Надеюсь, вам понравилось!