WARSAW UNIVERSITY OF TECHNOLOGY
FACULTY OF MATHEMATICS AND INFORMATION SCIENCE

PROJECT 3: REPORT

Image generation with generative adversarial networks

Dora Doljanin, Filip Pavičić

Warsaw, June 2022.

Table of contents

Introduction 1

1. Research problem 2

1.1. Description of the problem 2

1.2. Requirements 2

2. Theoretical introduction 3

2.1. Image generation 3

2.2. Generative modelling 3

2.3. Generative adversarial networks 3

3. Instruction of the application 4

3.1. Code 4

3.1.1. Preparing the dataset 4

3.1.2. Defining the models 5

3.1.3. Defining the hyper-parameters 11

4. Description of the conducted experiments 13

4.1. Process of training the models 13

4.2. Examining an influence of parameters' change on the obtained results 15

4.3. Interpolation of Images 15

5. Result analysis 17

5.1. Images generated by DCGAN 17

5.2. Images generated by WGAN-GP 18

5.3. Additional Commentary on the Results 19

Conclusion 20

Literature 21

Introduction

Image generation has become an exceptionally popular topic recently, with numerous practical applications in many different fields. From generating new Anime characters, to predicting what your future children might look like, image generation is an exciting and rapidly growing field.

Image generation can produce photorealistic photos of objects, scenes, and people that even humans cannot tell are fake. It can be used to transform low resolution photos and videos into high resolution ones. Also, it can generate realistic examples for solving translation tasks such as translating photos of summer to winter or day to night. It can also be used for synthesis faces in different poses: with a single input image, we create faces in different viewing angles, which can be used to transform images that will be easier for face recognition. [3]

Image generation is a great tool for data augmentation, as it enables us to multiply the amount of data that we are working with. Interestingly, image generation is becoming very popular in the medical and beauty industry, as well. Since it enables generating images of people who do not actually exist, it can be used without ethical concerns and privacy limitations. These are just a few of the numerous examples where image generation techniques can be used in practice.

In the following chapter, we will first describe the research problem. Then, in the second chapter, we will provide a theoretical introduction to image generation and generative adversarial networks. In the third chapter, we will present the application developed to solve the proposed problem and provide an overview of the technologies and tools used in the implementation. In the last two chapters, we will explain the conducted experiments and analyse the results.

1. Research problem

1.1. Description of the problem

We aim to solve the task of generating new images from an existing dataset. We use the "LSUN bedroom scene 20% sample" dataset, which consists of 303,125 jpgs containing bedroom scenes. [1] This data is particularly useful for generative models. We want to solve the proposed problem using deep learning and generative adversarial networks (GANs). We further want to test and compare the performances of different neural network architectures by conducting several experiments.

1.2. Requirements

The requirements are the following:

- At least one of the network architectures should converge to generate satisfactory images.
- We want to investigate the influence of hyperparameters' change on the obtained results.
- We should assess our results qualitatively.
- We should calculate the Fréchet Inception Distance (FID) for our generated images and compare it to results from the literature.

2. Theoretical introduction

2.1. Image generation

Image generation (synthesis) is the task of generating new images from an existing dataset. One of the most successful approaches to solving the image generation task is using deep learning and Generative Adversarial Networks (GANs). There are several such models in the literature which have demonstrated high-quality performances. [5][6]

2.2. Generative modelling

Generative modelling is an unsupervised learning task in machine learning that involves automatically discovering and learning the regularities or patterns in input data. [1] This way the model can be used to generate or output new examples from the original dataset.

2.3. Generative adversarial networks

Generative adversarial networks (GANs) are a deep-learning-based generative model firstly proposed in the paper „Generative Adversarial Nets”. [4] They are used for training a generative model. They frame the problem as a supervised learning problem with two sub-models: the generator model that we train to generate new examples, and the discriminator model that tries to classify examples as either real or fake. The two models are trained together in an adversarial game, until the discriminator model is fooled about half of the time, meaning that the generator model is generating examples that look real enough. [1]

3. Instruction of the application

We implemented the application in the Python programming language in the form of a Jupyter notebook. We used the PyTorch framework, and we utilized the resources provided by Kaggle. The following are the details of the implementation process and the corresponding code snippets.

3.1. Code

3.1.1. Preparing the dataset

We downloaded the dataset from Kaggle into the working environment's file system. For loading the data, we created our class *ImageDataset* which has a getter for retrieving images.

```
def _load_folder_list(root):
    return  list(str(p)[len(root)+ 1:] for p in Path(root).rglob("*.jpg"))

class ImageDataset(Dataset):
    def __init__(self, img_dir, transform=None, ):
        self.img_dir = img_dir
        self.transform = transform
        self.filenames = _load_folder_list(img_dir)


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

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.filenames[idx])
        image = (read_image(img_path,) / 255)
        label = 0
        if self.transform:
            image = self.transform(image)
        return image, label
```

After that, we load the data and crop all images to a fixed size 64*64 so that we can train them using models which expect a fixed image size.

```
dataset = ImageDataset(dataroot,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                           ]))
```

Additionally, we created some help functions which will help us later in the coding.

We created the *UnNormalize* class which reverses the effect of the image normalization, so that we can se images in a correct way when we display them.

```
class UnNormalize(object):
    def __init__(self, mean, std):
        self.mean = mean
        self.std = std

    def __call__(self, tensor):
        """
        Args:
            tensor (Tensor): Tensor image of size (C, H, W) to be normalized.
        Returns:
            Tensor: Normalized image.
        """
        for t, m, s in zip(tensor, self.mean, self.std):
            t.mul_(s).add_(m)
            # The normalize code -> t.sub_(m).div_(s)
        return tensor
```

According to the advice mentioned in the literature, we want to initialize the weights before training. To do that, we created a function called *weights_init*.

```
def weights_init(model):
    # Initializes weights according to the DCGAN paper
    for m in model.modules():
        if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d, nn.BatchNorm2d)):
            nn.init.normal_(m.weight.data, 0.0, 0.02)
```

3.1.2. Defining the models

We define three different models with DCGAN, WGAN and WGAN-GP architectures. All of them include two models: the generator and the discriminator.

DCGAN

We define the generator model as suggested in the paper. [5] The only change we did was that we changed the *ReLu* layer into the *LeakyReLu* layer, because it produced a little bit better result.

```
chanals_generator= [1024,512,256,128]
generator = nn.Sequential(
        # (1,1,100)
        nn.ConvTranspose2d(noiseDimension, chanals_generator[0], 4, 1, 0, bias=False),
        nn.BatchNorm2d(chanals_generator[0]),
        nn.LeakyReLU(0.2, inplace=True),
        # (4,4,1024)
        nn.ConvTranspose2d(chanals_generator[0], chanals_generator[1], 4, 2, 1, bias=False),
        nn.BatchNorm2d(chanals_generator[1]),
        nn.LeakyReLU(0.2, inplace=True),
        # (8,8,512)
        nn.ConvTranspose2d(chanals_generator[1], chanals_generator[2], 4, 2, 1, bias=False),
        nn.BatchNorm2d(chanals_generator[2]),
        nn.LeakyReLU(0.2, inplace=True),
        # (16,16,256)
        nn.ConvTranspose2d(chanals_generator[2], chanals_generator[3], 4, 2, 1, bias=False),
        nn.BatchNorm2d(chanals_generator[3]),
        nn.LeakyReLU(0.2, inplace=True),
        # (32,32,128)
        nn.ConvTranspose2d( chanals_generator[3], 3, 4, 2, 1, bias=False),
        #(64,64,3)
        nn.Tanh()
).cuda()
weights_init(generator)
```

We define the discriminator model according to the paper instructions, with addition of adding a dropout layer to achieve a better generalization of the discriminator.

```
chanals_discriminator= [64,128,256,512]
discriminator = nn.Sequential(
        nn.Conv2d(3, chanals_discriminator[0], 4, 2, 1, bias=False),
        nn.LeakyReLU(0.2, inplace=True),
        nn.Conv2d(chanals_discriminator[0], chanals_discriminator[1], 4, 2, 1, bias=False),
        nn.BatchNorm2d(chanals_discriminator[1]),
        nn.LeakyReLU(0.2, inplace=True),
        nn.Dropout(0,4),
        # state size. (ndf*2) x 16 x 16
        nn.Conv2d(chanals_discriminator[1], chanals_discriminator[2], 4, 2, 1, bias=False),
        nn.BatchNorm2d(chanals_discriminator[2]),
        nn.LeakyReLU(0.2, inplace=True),
        nn.Dropout(0,4),
        # state size. (ndf*4) x 8 x 8
        nn.Conv2d(chanals_discriminator[2], chanals_discriminator[3], 4, 2, 1, bias=False),
        nn.BatchNorm2d(chanals_discriminator[3]),
        nn.LeakyReLU(0.2, inplace=True),
        nn.Dropout(0,4),
        # state size. (ndf*8) x 4 x 4
        nn.Conv2d(chanals_discriminator[3], 1, 4, 1, 0, bias=False),
        nn.Sigmoid()
).cuda()
weights_init(discriminator)
```

We used the Adam optimizer and the Binary Cross Entropy loss function. Additionally, we created 6 latent vectors which will help us track the progression of the network through the epochs.

```
# Initialize BCELoss function
loss_function = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
#  the progression of the generator
fixed_noise = torch.randn(6, noiseDimension, 1, 1).cuda()

# Establish convention for real and fake labels during training
real_label = 1.
fake_label = 0.

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(discriminator.parameters(), lr=lr, betas=(0.5, 0.999))
optimizerG = optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.999))
```

We define the train function. Firstly, we train the discriminator in a way that it labels real images as 1 and fake images as 0. After that, we train the generator to trick the discriminator into labelling fake images as 1. Then, according to the results of the discriminator, the generator calculates its loss.

```
def train(generator,discriminator, dataloader, num_epochs, optimizerD, optimizerG, loss_function, printAfteBatch = 500,
          cols = 3,chunk = 100, n_critic = 1):
    G_losses = []; D_losses = []; iter = 0; last_iter = 0; chunk_errD = []; chunk_errG = []; chunk_lossD = []; chunk_lossG = []
    chunk_mean_D = []; chunk_mean_G = []; chunk_mean_lossD = []; chunk_mean_lossG = []; norm_gradD = []; norm_gradG = []

    for epoch in range(num_epochs):
        with tqdm(enumerate(dataloader), total=len(dataloader), desc=f'Training (epoch={epoch}/{num_epochs})') as epoch_progress:
            for batch_idx, train_batch in epoch_progress:
                #Update discriminator
                for _ in range(n_critic):
                    #Train on real images
                    discriminator.zero_grad()
                    x,_ = train_batch
                    x = x.cuda()
                    y =  torch.full((x.shape[0],), real_label, dtype=torch.float).cuda()
                    output = discriminator(x).view(-1)
                    lossD_real = loss_function(output,y)
                    errD = output.mean().item()

                    #Train on fake images
                    noise = torch.randn(x.shape[0], noiseDimension, 1, 1).cuda()
                    y =  torch.full((x.shape[0],), fake_label, dtype=torch.float).cuda()
                    fake_image = generator(noise)
                    output = discriminator(fake_image).view(-1)
                    lossD_fake = loss_function(output,y)

                    errG_disc = output.mean().item()

                    lossD = (lossD_real + lossD_fake) / 2
                    lossD.backward(retain_graph = True)
                    optimizerD.step()

                #Update generator
                generator.zero_grad()
                y_real =  torch.full((x.shape[0],), real_label, dtype=torch.float).cuda()
                output = discriminator(fake_image).view(-1)
                lossG = loss_function(output,y_real)
                lossG.backward()
                errG_gen = output.mean().item()

                optimizerG.step()
```

WGAN

We define the generator model the same way we did for the DCGAN.

```
chanals_generator= [1024,512,256,128]
generator1 = nn.Sequential(
        # (1,1,100)
        nn.ConvTranspose2d(noiseDimension, chanals_generator[0], 4, 1, 0, bias=False),
        nn.BatchNorm2d(chanals_generator[0]),
        nn.LeakyReLU(0.2, inplace=True),
        # (4,4,512)
        nn.ConvTranspose2d(chanals_generator[0], chanals_generator[1], 4, 2, 1, bias=False),
        nn.BatchNorm2d(chanals_generator[1]),
        nn.LeakyReLU(0.2, inplace=True),
        # (0,8,256)
        nn.ConvTranspose2d(chanals_generator[1], chanals_generator[2], 4, 2, 1, bias=False),
        nn.BatchNorm2d(chanals_generator[2]),
        nn.LeakyReLU(0.2, inplace=True),
        # (16,16,128)
        nn.ConvTranspose2d(chanals_generator[2], chanals_generator[3], 4, 2, 1, bias=False),
        nn.BatchNorm2d(chanals_generator[3]),
        nn.LeakyReLU(0.2, inplace=True),
        # (32,32,64)
        nn.ConvTranspose2d( chanals_generator[3], 3, 4, 2, 1, bias=False),
        nn.Tanh()
).cuda()
generator1.apply(weights_init)
```

The discriminator model is almost the same, the only difference is that we remove the last sigmoid layer.

```
chanals_discriminator= [64,128,256,512]
discriminator1 = nn.Sequential(
        nn.Conv2d(3, chanals_discriminator[0], 4, 2, 1, bias=False,),
        nn.LeakyReLU(0.2, inplace=True),
        nn.Conv2d(chanals_discriminator[0], chanals_discriminator[1], 4, 2, 1, bias=False),
        nn.BatchNorm2d(chanals_discriminator[1]),
        nn.LeakyReLU(0.2, inplace=True),
        # state size. (ndf*2) x 16 x 16
        nn.Conv2d(chanals_discriminator[1], chanals_discriminator[2], 4, 2, 1, bias=False),
        nn.BatchNorm2d(chanals_discriminator[2]),
        nn.LeakyReLU(0.2, inplace=True),
        # state size. (ndf*4) x 8 x 8
        nn.Conv2d(chanals_discriminator[2], chanals_discriminator[3], 4, 2, 1, bias=False),
        nn.BatchNorm2d(chanals_discriminator[3]),
        nn.LeakyReLU(0.2, inplace=True),
         # state size. (ndf*4) x 4 x 4
        nn.Conv2d(chanals_discriminator[3],1, 4, 2, 0, bias=False),

).cuda()
```

We use the RMSprop optimizer, and we create 6 latent vectors.

```
# Setup RMSprop optimizers for both G and D
optimizerD1 = optim.RMSprop(discriminator1.parameters(), lr=lr1)
optimizerG1 = optim.RMSprop(generator1.parameters(), lr=lr1)
```

In the training method, we do the clipping of the weights after each epoch to satisfy the Lipschitz constraint, as specified in the paper. [7] This way we clip every weight parameter in the network to be between –*weight_clip*, and +*weight_clip* (in our case, between - 0.01 and 0.01). Also, we define the loss function of the discriminator (*lossD*) and the loss function of the generator (*lossG*), which are highlighted in the following picture.

```
def train1(generator,discriminator, dataloader, num_epochs, optimizerD, optimizerG, printAfteBatch = 500, cols = 3,chunk = 100,
           n_critic = 1):
    G_losses = []; D_losses = []; iter = 0; last_iter = 0; chunk_errD = []; chunk_errG = []; chunk_lossD = []; chunk_lossG = [];
    chunk_mean_D = []; chunk_mean_G = []; chunk_mean_lossD = []; chunk_mean_lossG = []; norm_gradD = []; norm_gradG = []
    for epoch in range(num_epochs):
        with tqdm(enumerate(dataloader), total=len(dataloader), desc=f'Training (epoch={epoch}/{num_epochs})') as epoch_progress:
            for batch_idx, train_batch in epoch_progress:
                #Update discriminator
                for _ in range(n_critic):
                    #Train on real images
                    discriminator.zero_grad()
                    x,_ = train_batch
                    x = x.cuda()
                    y =  torch.full((x.shape[0],), real_label, dtype=torch.float).cuda()
                    output_real = discriminator(x).view(-1)

                    errD = output_real.mean().item()

                    #Train on fake images
                    noise = torch.randn(x.shape[0], noiseDimension, 1, 1).cuda()
                    y.fill_(fake_label)
                    fake_image = generator(noise)
                    output_fake = discriminator(fake_image.detach()).view(-1)

                    errG_disc = output_fake.mean().item()

                    lossD = -(torch.mean(output_real) - torch.mean(output_fake))
                    lossD.backward(retain_graph = True)
                    optimizerD.step()

                    for p in discriminator.parameters():
                        p.data.clamp_(-weight_clip,weight_clip)

                #Update generator
                generator.zero_grad()
                y_real =  torch.full((x.shape[0],), real_label, dtype=torch.float).cuda()
                output = discriminator(fake_image).view(-1)
                lossG = -torch.mean(output)
                lossG.backward()

                errG_gen = output.mean().item()

                optimizerG.step()
```

However, weight clipping is not the best way to make sure that the Lipschitz constraint is satisfied. If the clipping parameter is big, then it can take a long time for any weights to reach their limit, which makes it harder to train the discriminator until it reaches optimal performance. On the other hand, if the clipping parameter is small, while number of layers is big or batch normalization is not used, it can lead to a vanishing gradient. [7]

We can escape this problem by using WGAN-GP (WGAN with gradient penalty), which, instead of clipping, satisfies the Lipschitz constraint by interpolating between real and generated images and checking if the norm of the gradient for each interpolations is equal to 1.

WGAN-GP

For the WGAN-GP, the generator model is the same as for the previous two models. In the discriminator model, instead of *BatchNorm2d* layer, we put the *InstanceNorm2d* layer.

```
chanals_discriminator= [64,128,256,512]
discriminator2 = nn.Sequential(
        nn.Conv2d(3, chanals_discriminator[0], 4, 2, 1, bias=False,),
        nn.LeakyReLU(0.2, inplace=True),
        nn.Conv2d(chanals_discriminator[0], chanals_discriminator[1], 4, 2, 1, bias=False),
        nn.InstanceNorm2d(chanals_discriminator[1],affine = True),
        nn.LeakyReLU(0.2, inplace=True),
        # state size. (ndf*2) x 16 x 16
        nn.Conv2d(chanals_discriminator[1], chanals_discriminator[2], 4, 2, 1, bias=False),
        nn.InstanceNorm2d(chanals_discriminator[2],affine = True),
        nn.LeakyReLU(0.2, inplace=True),
        # state size. (ndf*4) x 8 x 8
        nn.Conv2d(chanals_discriminator[2], chanals_discriminator[3], 4, 2, 1, bias=False),
        nn.InstanceNorm2d(chanals_discriminator[3],affine = True),
        nn.LeakyReLU(0.2, inplace=True),
         # state size. (ndf*4) x 4 x 4
        nn.Conv2d(chanals_discriminator[3], 1, 4, 2, 0, bias=False),

).cuda()
weights_init(discriminator2)
```

We use the Adam optimizer, and we create 6 latent vectors.

```
# Setup Adam optimizers for both G and D
optimizerD2 = optim.Adam(discriminator2.parameters(), lr=lr2,betas= (0.0,0.90))
optimizerG2 = optim.Adam(generator2.parameters(), lr=lr2,betas= (0.0,0.90))
```

We define the *gradient_penalty* function, which creates new images by interpolating between real and fake image (with a random ratio *epsilon*). Then, it calculates their scores, as well as the gradient of the scores with respect to the interpolated images. Finally, we take the norm of the gradient and calculate the penalty.

```
def gradient_penalty(discriminator, real_iamges, fake_images):
    batch_size, c, h, w = real_iamges.shape
    epsilon = torch.rand((batch_size,1,1,1)).repeat(1,c,h,w).cuda()
    new_image = real_iamges * epsilon + (1 - epsilon) * fake_images
    score = discriminator(new_image)
    gradient = torch.autograd.grad(
        inputs = new_image, outputs = score, grad_outputs = torch.ones_like(score),create_graph = True ,retain_graph = True
    )[0]
    gradient = gradient.view(gradient.shape[0], - 1)
    norm = gradient.norm(2, dim = 1)
    penalty = torch.mean((norm-1) ** 2)
    return penalty
```

We define the training function, where we change the loss function by adding the *lambda_gp* * *gp* element (*gp* = gradient penalty).

```
def train2(generator,discriminator, dataloader, num_epochs, optimizerD, optimizerG, printAfteBatch = 500, cols = 3,
           chunk = 100, n_critic = 1):
    G_losses = []; D_losses = []; iter = 0; last_iter = 0; errD_real = []; errD_fake = []; norm_gradD = []; norm_gradG = []
    for epoch in range(num_epochs):
        with tqdm(enumerate(dataloader), total=len(dataloader), desc=f'Training (epoch={epoch}/{num_epochs})') as epoch_progress:
            for batch_idx, train_batch in epoch_progress:
                #Update discriminator
                for _ in range(n_critic):
                    #Train real image
                    discriminator.zero_grad()
                    x,_ = train_batch
                    x = x.cuda()
                    output_real = discriminator(x).view(-1)

                    #Train fak image
                    noise = torch.randn(x.shape[0], noiseDimension, 1, 1).cuda()
                    fake_image = generator(noise)
                    output_fake = discriminator(fake_image).view(-1)

                    gp = gradient_penalty(discriminator,x,fake_image)
                    lossD = -(torch.mean(output_real) - torch.mean(output_fake)) + lambda_gp * gp

                    lossD.backward(retain_graph = True)
                    optimizerD.step()

                    errD_real.append(output_real.mean().item())
                    errD_fake.append(output_fake.mean().item())
                    D_losses.append(lossD.item())

                #Update generator
                generator.zero_grad()
                output = discriminator(fake_image).view(-1)

                lossG = - torch.mean(output)

                lossG.backward()
                optimizerG.step()
```

3.1.3. Defining the hyper-parameters

The hyper-parameters we used are the following:

- ***Batch size*** – the number of examples in one training group
- ***Learning rate*** – the number that multiplies the gradient, which determines how much the weights will be adjusted in the direction opposite of the gradient
- ***Noise dimension*** – the size of the vector used by the generator for generating the images
- ***Weight clip*** – the number that defines the bound of the weights' parameter as [-weight_clip, + weight_clip]
- ***N_critic*** – the number of times that the discriminator will be trained more than the generator in one training
- ***Lambda_gp –*** the number which the gradient penalty is multiplied by in the loss function of the discriminator

DCGAN

- ***Learning rate*** = 0.0002
- ***batch_size*** = 128
- ***noiseDimension*** = 100

WGAN

- ***Learning rate*** = 0.00005
- ***weight_clip*** = 0.01
- ***n_critic*** = 5
- ***batch_size*** **=** 64
- ***noiseDimension*** **=** 100

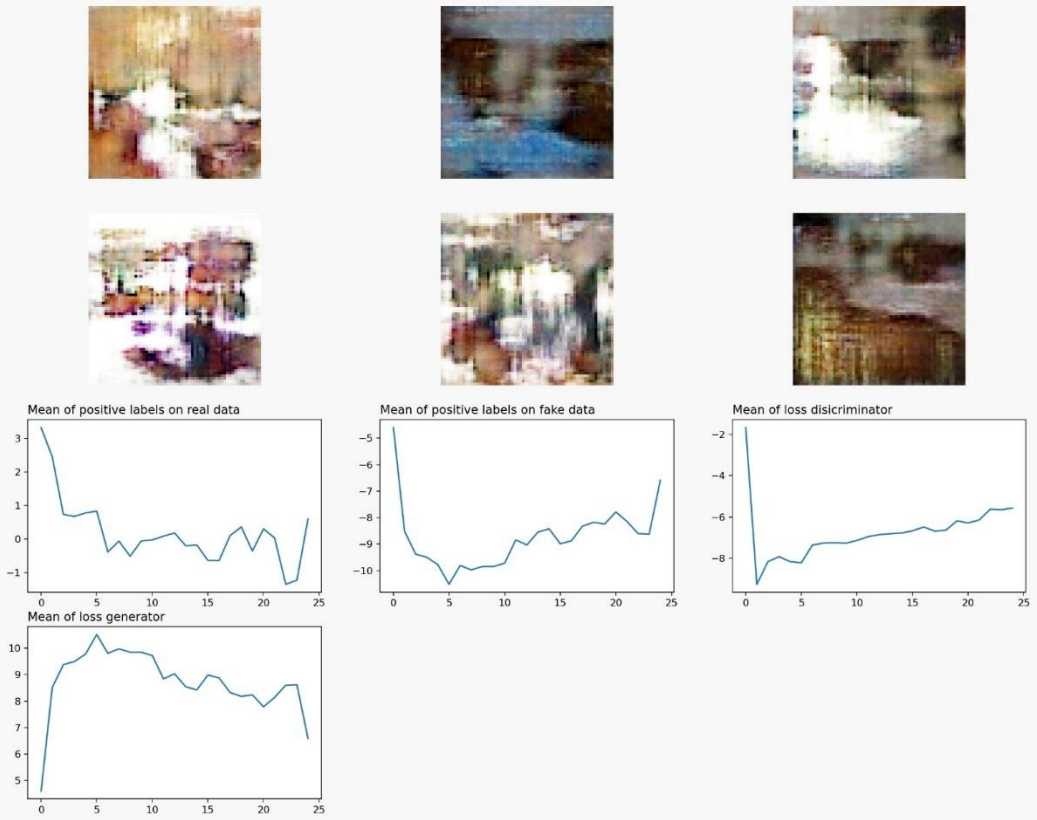
WGAN-GP

- ***Dropout rate*** = 0.0001
- ***n_critic2*** = 5
- ***lambda_gp*** = 10
- ***batch_size*** = 64
- ***noiseDimension*** = 100

4. Description of the conducted experiments

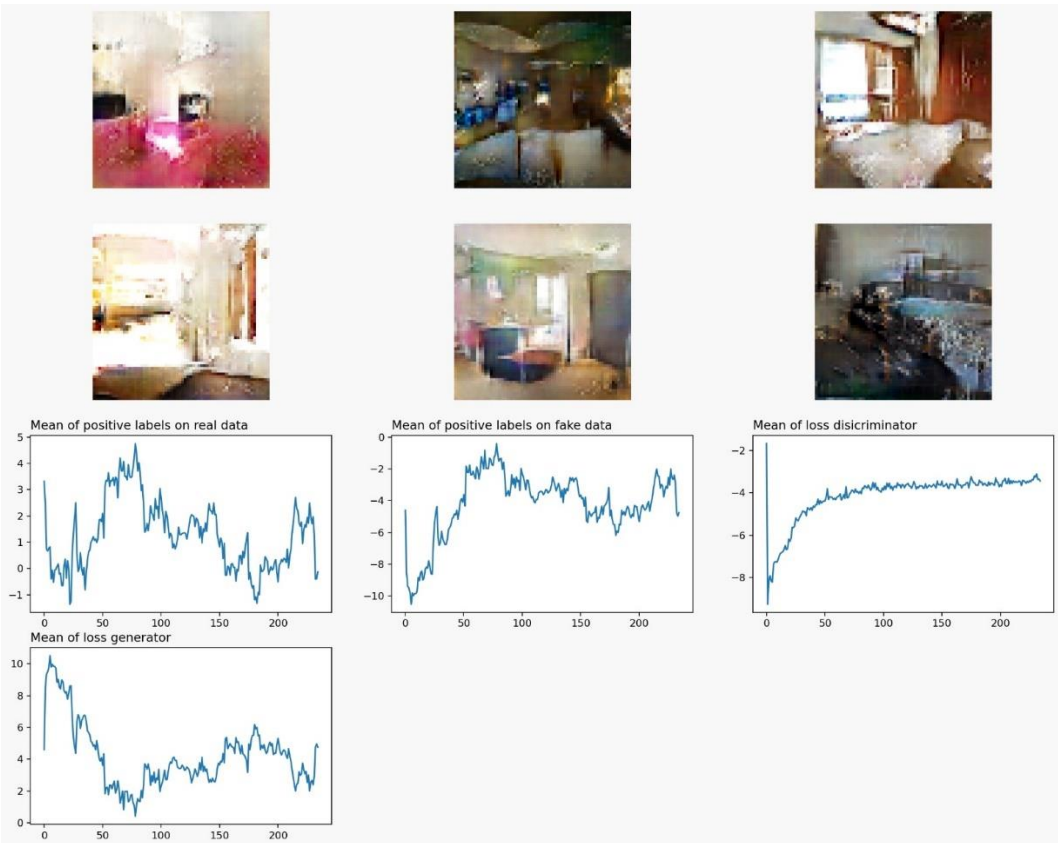
4.1. Process of training the models

Unlike in the last two projects, the models used in this project have a very long training time. The process of training for 5 epochs lasted approximately 75 min for DCGAN, 115 min for WGAN and 235 min for WGAN-GP. This makes it quite challenging to repeat many experiments. Thus, we tried to obtain some knowledge from the training process. After each 500 training processes, we printed out the generated images using the latent vectors that we had previously defined. Also, we printed some graphs of the mean loss of the discriminator and the generator in order to gain a better insight in the training process.



1. After 500 training processes

2. After 11,000 training processes



3. After 25,000 training processes

4.2. Examining an influence of parameters' change on the obtained results

Due to a long training process and a lack of method for quantitative validation of the obtained results, we decided to try changing the hyperparameters and models according to suggestions in the literature. [16] To improve the performance, we tried adding dropout, smoothing the labels, and changing the *ReLu* layer to *LeakyReLu* layer. We noticed that the model gave a little bit better results after we put the *LeakyReLu* layer. However, for the other changes, we did not notice any significant improvement on the results.

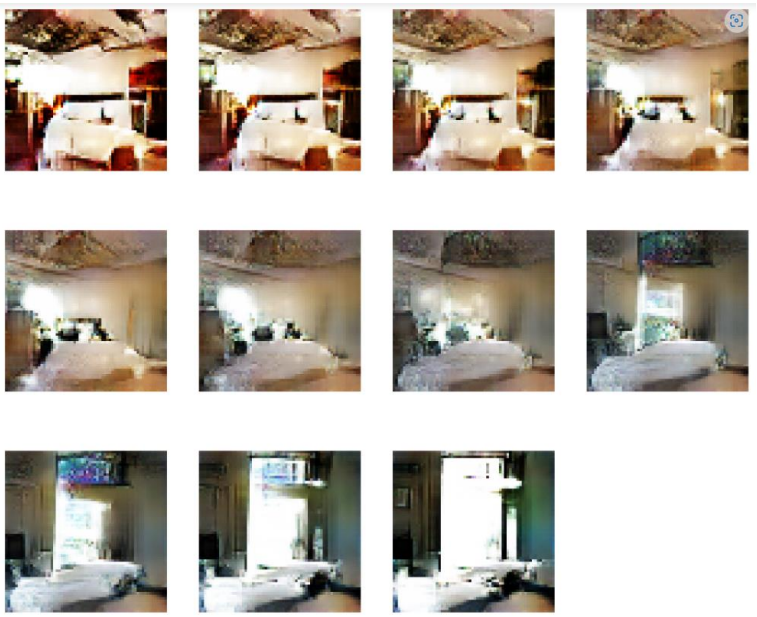
4.3. Interpolation of Images

In order to observe how the image interpolation works, we created 2 random vectors and interpolated them.

```
a =  torch.randn(1, noiseDimension, 1, 1)
b =  torch.randn(1, noiseDimension, 1, 1)
outputs = []
ratios = [0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]
for i in ratios:
    c = a * i + (1-i) * b
    outputs.append(c)
result = torch.cat(outputs, dim=0).cuda()
```

```
with torch.no_grad():
    interpolate = generator(result).detach().cpu()
```

The result images are the following:



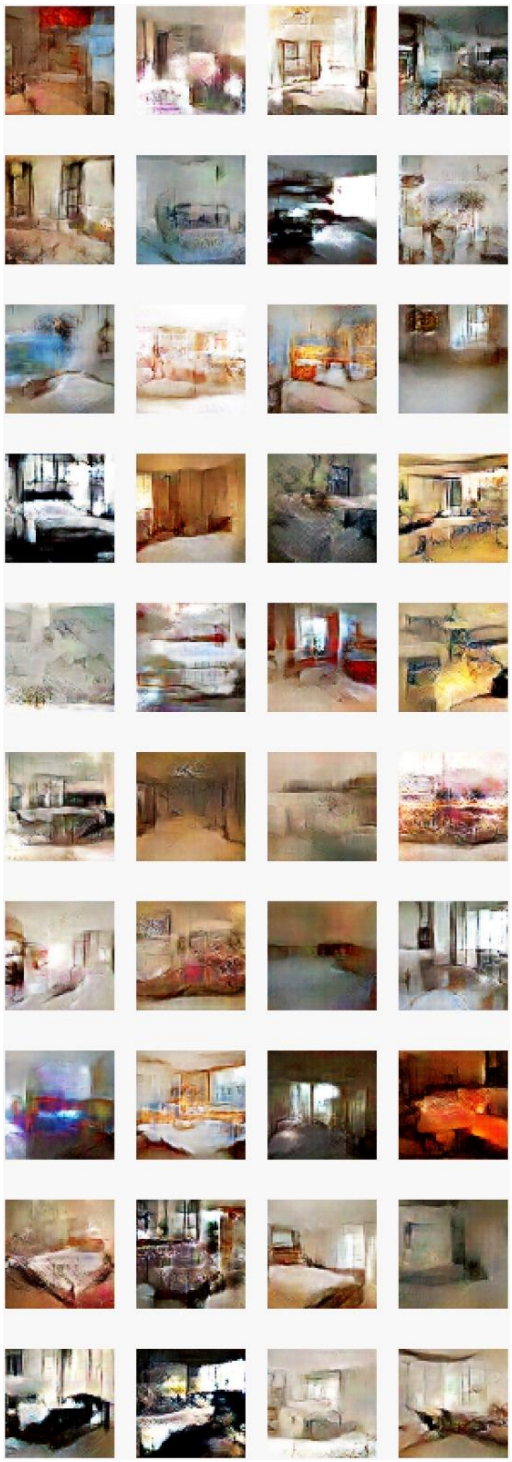
Theoretically, there are 256 to the power of 12,288 possible combinations of the image pixels for this problem, which is an incomprehensibly huge number. However, a vast majority of those combinations of pixels do not form any reasonable bedroom image. Actual bedroom images take up only a tiny subset of the parent's space of all the possible pixel combinations. That subset is highly structured and continuous. [17]

In this task, we represented that tiny subspace by the space containing the initial vectors. Using the generator model, we can move from that space to the parent's space of all possible images. We observe that the subspace of bedroom-looking images is indeed continuous and highly structured, since it is possible to interpolate between any two vectors from the set of initial vectors and obtain realistic bedroom images from the parent's space.

5. Result analysis

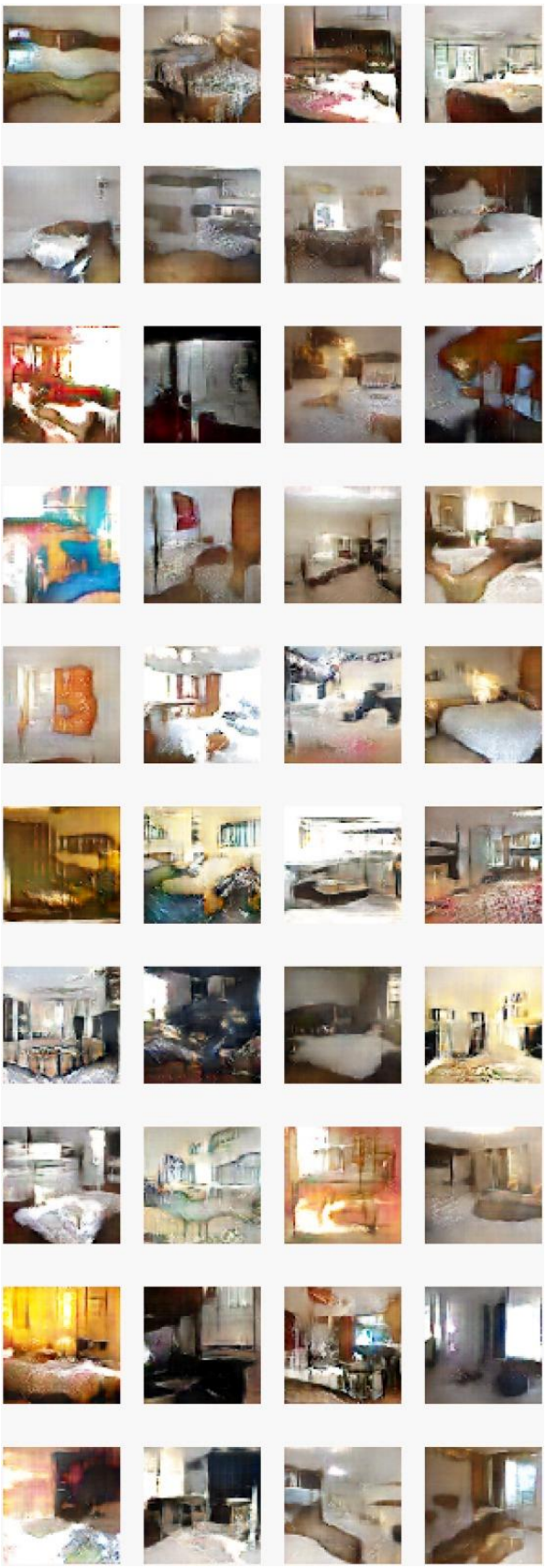
5.1. Images generated by DCGAN

Using the DCGAN we achieved an FID of 625.7 after 5 epochs.



5.2. Images generated by WGAN-GP

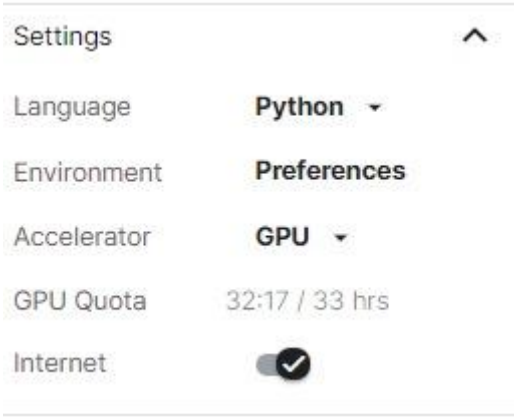
We did not manage to generate good images using the WGAN model. We suppose that the reason for that is the clipping of weights, so we went on to analyse the results of the WGAN-GP. They obtained an FID value of 943.18 after 5 epochs.



5.3. Additional Commentary on the Results

In the generated images we can notice the convergence of the results, which is good because our models managed to obtain some profound outcomes.

Current state-of-the-art for the task we want to solve in this project is between 1 and 10 FID value. We are aware that our results are not close to the current best results in literature. This is probably because we used some of the most basic GANs, along with the limited resources we had at our hand in terms of the GPU and memory and a lack of previous experience in working with GANs.



Conclusion

In this project, we aimed to solve the problem of generating new images from an existing dataset using deep learning and generative adversarial networks. We implemented, tested, and compared different network architectures, including DCGAN, WGAN, and WGAN-GP, to find the best possible solution.

We managed to achieve convergence for all three models. We achieved an FID of 625.7 after 5 epochs using the DCGAN, and an FID value of 943.18 after 5 epochs for the WGAN-GP. The images looked decent, but their quality and FID values were not close to the state-of-the-art results in literature.

However, we may conclude that we have successfully solved the problem of generating new images from an existing dataset. We suppose that if we had had more time and better resources, we could have obtained better quality images.

Literature

[1] Jason Brownlee, *A Gentle Introduction to Generative Adversarial Networks (GANs),* Generative Adversarial Networks (June 17, 2019). Link: https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/

[2] LSUN bedroom scene 20% sample, Link: https://www.kaggle.com/datasets/jhoward/lsun_bedroom

[3] Jonathan Hui, *GAN — Some cool applications of GAN,* (Jun 22, 2018). Link: https://jonathan-hui.medium.com/gan-some-cool-applications-of-gans-4c9ecca35900

[4] Goodfellow, Ian, et al. "Generative adversarial nets." Advances in neural information processing systems 27 (2014).

[5] Radford, Alec, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks." arXiv preprint arXiv:1511.06434 (2015).

[6] Karras, Tero, et al. "Alias-free generative adversarial networks." Advances in Neural Information Processing Systems 34 (2021).

[7] Arjovsky, Martin, Soumith Chintala, and Léon Bottou. "Wasserstein generative adversarial networks." International conference on machine learning. PMLR, 2017.

[8] Christian Versloot, *GANs – A Gentle Introduction,* Machine Learning Articles, GitHub. Link: https://github.com/christianversloot/machine-learning-articles/blob/main/generative-adversarial-networks-a-gentle-introduction.md

[9] Jason Brownlee, *How to Develop a Wasserstein Generative Adversarial Network (WGAN) From Scratch*, Generative Adversarial Networks (July 17, 2019). Link: https://machinelearningmastery.com/how-to-code-a-wasserstein-generative-adversarial-network-wgan-from-scratch/

[10] Guilhem Le Moigne, *GAN_lsun*, Kaggle (29 Jan, 2022). Link: https://www.kaggle.com/code/guilhemlemoigne/gan-lsun

[11] Nathan Inkawhich, *Dcgan Tutorial*, Tutorials, PyTorch. Link: https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

[12] Aladdin Persson, *WGAN implementation from scratch (with gradient penalty*), YouTube (Nov 3, 2020). Link: https://www.youtube.com/watch?v=pG0QZ7OddX4&ab_channel=AladdinPersson

[13] Ayush Thakur, *How to Evaluate GANs using Frechet Inception Distance (FID),* Weights&Biases (Apr 19, 2021). Link: https://wandb.ai/ayush-thakur/gan-evaluation/reports/How-to-Evaluate-GANs-using-Frechet-Inception-Distance-FID---Vmlldzo0MTAxOTI

[14] Jason Brownlee, *How to Implement the Frechet Inception Distance (FID) for Evaluating GANs*, Generative Adversarial Networks (30 Aug, 2019). Link: https://machinelearningmastery.com/how-to-implement-the-frechet-inception-distance-fid-from-scratch/

[15] Gulrajani, Ishaan, et al. "Improved training of wasserstein gans." Advances in neural information processing systems 30 (2017).

[16] Soumith, How to Train a GAN? Tips and tricks to make GANs work, ganhacks, GitHub. Link: https://github.com/soumith/ganhacks

[17] Chollet, Francois. Deep learning with Python. Simon and Schuster, 2021.