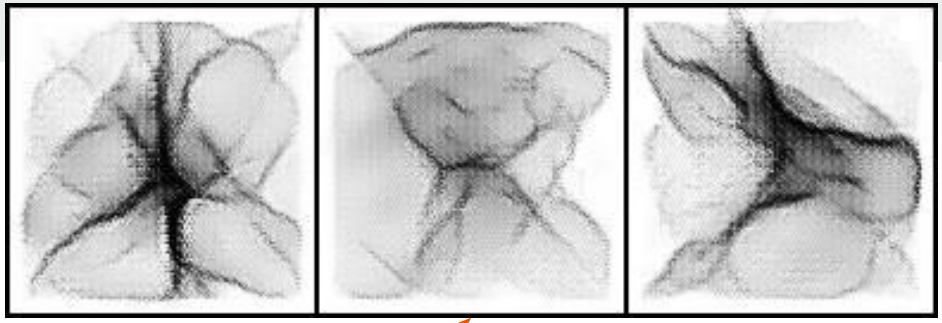
Generating Lissajous Figures using GAN

Duccio Meconcelli



Summary

The goal of this project is to create a GAN that is capable of generating Lissajous figures.
Generating Lissajous figures is not a straightforward task due to their great variability.

In this study, a significant challenge was quantifying the quality of the generated Lissajous figures. Unlike a classification problem, where the network can be evaluated based on its accuracy in categorizing the data, it is much more difficult to obtain precise metrics for the evaluation of generated figures.
To overcome this challenge, we used a combination of qualitative and quantitative methods to assess the quality of the generated figures, including visual comparison and mathematical analysis of the generated figures.

We also experimented with different parameters and architectures of the GAN in order to improve the quality of the generated figures. This allowed us to gain insights into the optimal parameters and architectures for generating Lissajous figures using GANs.

At the end, despite the challenges in comparing and analyzing GANs, the best model according to the metrics and comparing images between various models at a glance was the one from Experiment 22.

While the quality of the images produced is not the best, we can still see some shapes, but far from the harmony and geometry of the Lissajous figures.

LINKS
Github Repo
Report
Dataset
Experiment

Dataset

Lissajous figures are graphical representations of the superposition of two harmonic waves.
To create the Lissajous figures dataset, we started from `harmonograph.py` inspired by a Github Repo.
The creation of the images was made using the overlapping of multiple pendulums from 2 to 4 (from empirical experiments the number of pendulums 2 - 4 make it possible to create 'prettier' images).

We created the dataset to a total of 20'000 images 128 x 128 pixels.
Unsuitable images were removed to ensure a balanced distribution of Lissajous figures.
(It was seen that with the number of pendulums at 2 too many circular figures were created, so it was decided to distort the frequency a bit)

```python
def gen_harmonograph(
    n_images,n_pend,n_steps,my_dpi,dimension,...,line_width):
    for index in range(n_images[0], n_images[1]):
        if n_pend == None:
            npend = r.randint(2, 4)

        if n_steps == None:
            steps = r.randrange(25000, 46000, 5000)

        mf = npend  # number of pendulums & maximum frequency

        sigma = r.uniform(0.003, 0.004)
        step = 0.0110

        linew = line_width

        t = arange(steps) * step  # time axis
        d = 1 - arange(steps) / steps  # decay vector

        ax = [r.uniform(0.5, 1.5) for i in range(npend)]
        ay = [r.uniform(0.5, 2) for item in range(npend)]

        if r.randint(0, 6) == 3: #more randomness
            ax = ay  # when they are equal I get closed figures

        px = [r.uniform(0, 2 * pi) for i in range(npend)]
        py = [r.uniform(0, 2 * pi) for i in range(npend)]
        fx = [r.randint(1, mf) + r.gauss(0, sigma) for i in range(npend)]
        fy = [r.randint(1, mf) + r.gauss(0, sigma) for i in range(npend)]

        if npend == 2:
            fx[0] += 1
            fx[1] += 1

        x = y = 0
        for i in range(npend):
            x += d * (ax[i] * sin(t * fx[i] + px[i]))
            y += d * (ay[i] * sin(t * fy[i] + py[i]))

        plt.figure(
            facecolor="white",
            figsize=(dimension / my_dpi, dimension / my_dpi),
            dpi=my_dpi,
        )
        plt.plot(x, y, "k", linewidth=linew)
```

```python
import torchvision.transforms as transforms

transform = transforms.Compose(
    [
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.5], std=[0.5]),
        transforms.RandomHorizontalFlip(0.5),
        transforms.Grayscale(),
    ]
)
```

On `data_loader.py` the images were then normalized between -1 and 1 and with a 50% probability, they were horizontally mirrored. This was done to increase the variability of the input data and to make the network more robust to changes in the orientation of the figures. The order of the images in the dataset was randomized to ensure that the network was not biased towards any specific type of Lissajous figure. This allowed us to train the network on a diverse set of Lissajous figures and to generate new, diverse figures using the network.

Experiments

The experiments were carried out in order to meet the goal of improving upon a baseline DCGAN network ('CLEAN' in the report) consisting for the Generator 1 Linear layer and 4 layers of convolutional transpose, and for the Discriminator 5 convolutional layer and 2 Linear layer; no gradient penalty, a learning rate of 0.0002 for both the generator and the discriminator over 100 epochs with a seed set to 42 for reproducibility.

The approach taken involved experimenting with different network architecture modifications:

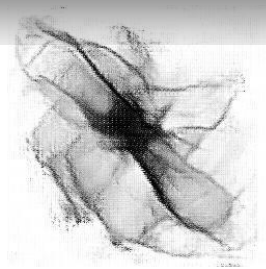
- adding layers for both generator and discriminator (adding more complexity produces more complex results)
- changing the type of the Generator Layers (Upsample with Convolutional 2D Layers or Convolutional 2D Transpose Layers)
- changing the loss function between BCE or Wasserstein Loss
- altering the learning rate for both the generator and the discriminator
- implementing or removing the gradient penalty
- modifying the batch size
- adjusting the relative training time for the discriminator compared to the generator (Increase or decrease the number of steps so that the network updates the generator parameters (disc_steps in the code)

We conducted *22 experiments* by modifying this parameters.

```python
class TrainParams:
    def __init__(self, **kwargs):
        # data parameters
        self.epoch =  100
        self.batch_size = 100
        self.train_size = 0

        # training parameters:
        self.disc_steps = 1
        self.rate_g =  0.0002
        self.rate_d = 0.0002

        self.gradient_penalty = 0
        self.betas = (0.5, 0.999)
        self.loss = "standard" #or "wasserstein"
        self.seed = None

        # saving and Logging:
        self.steps_per_log = 5
        self.steps_per_img_save = 100

        self.steps_per_checkpoint = 1000
        self.steps_per_clean = 2000
        self.telegram = False

        # testing parameters:
        self.steps_per_val = 500

        if self.train_size % self.batch_size != 0:
            self.batch_size = get_closest_batch_size(self.train_size, self.batch_size)

        self.step_per_epoch = int(self.train_size / self.batch_size)
        self.steps = self.step_per_epoch * self.epoch

    def save_params():
        #save the params in a json file


class ModelParams:
    def __init__(self, **kwargs):
        self.img_dim = 128
        self.random_seed = 42

        self.generator_transpose = True

        self.device = "cuda" if torch.cuda.is_available() else "cpu"
        self.out_dir = "OUTPUT"

        self.n_generate_img = 100
        self.telegram = True
```

The Generator Model

```python
class Generator_128(nn.Module):
    def __init__(self, transpose=True):
        super(Generator_128, self).__init__()
        self.transpose = transpose
        #create the layer

        # layer 1: Linear
        self.fc1 = nn.Linear(100, 128 * 16 * 16)
        if self.transpose == True:
            # Layer 2: Convolutional # input 16x16 output 32X32
            self.conv1 = nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1)
            # Layer 3: Convolutional # input 32X32 output 32X32
            self.conv2 = nn.ConvTranspose2d(64, 32, kernel_size=3, stride=1, padding=1)
            # Layer 4: Convolutional # input 32X32 output 64x64
            self.conv3 = nn.ConvTranspose2d(32, 16, kernel_size=4, stride=2, padding=1)
            # Layer 5: Convolutional # input 64x64 output 128x128
            self.conv4 = nn.ConvTranspose2d(16, 1, kernel_size=4, stride=2, padding=1)

        else:
            self.conv1 = nn.Conv2d(128, 64, kernel_size=4, stride=2, padding=1)
            #... all other Con2d layer...
            self.sample = nn.Upsample(
                scale_factor=4, mode="bilinear", align_corners=True
            )

        # BatchNorm
        self.batchnorm1D = nn.BatchNorm1d(128 * 16 * 16)
        self.batchnorm64 = nn.BatchNorm2d(64)
        self.batchnorm32 = nn.BatchNorm2d(32)
        self.batchnorm16 = nn.BatchNorm2d(16)

        # Dropout
        self.dropout = nn.Dropout(0.5)
```



While the network architecture varied during the experiments, the implementation is more or less always the same.
The basic model of the Network was chosen inspired by a DCGAN: by placing side by side convolutional/convolutional layers transposed to batch norm layers, and LeakyReLU activations.

On `model.py` in the first part, when we create the Generator object (subclass of Pytorch Module), we instantiate the layer modules that will be used by the network, for example: in this case, ConvTranspose2d layers are instantiated for the generator if the option is active, otherwise Conv2d and Upsample layers are created.
BarchNorm and Dropout modules are always instantiated because they are used with both transpose and non-transpose layers.

During the forwarding step, the modules instantiated in the `__initit__` are used to calculate the network output given a certain input.

During the 20 experiments were made changes both to the quantity of the layers and to the parameters (for example the number of output channels of a convolutional layer).

N.B.: during the `forward()` method we need a `view()` to switch from a 1D to a 2D representation in fact the generator takes as input a 1D Tensor (the noise) of magnitude 100.

```python
def forward(self, x):
        #using the layer for the feedforward step

        if self.transpose == True:
            # Layer 1
            x = self.fc1(x)
            x = self.batchnorm1D(x)
            x = x.view(-1, 128, 16, 16)
            x = F.leaky_relu(x, 0.2)
            x = self.dropout(x)
            # Layer 2
            x = self.conv1(x)
            x = self.batchnorm64(x)
            x = F.leaky_relu(x, 0.2)
            x = self.dropout(x)
            # Layer 3
            x = self.conv2(x)
            x = self.batchnorm32(x)
            x = F.leaky_relu(x, 0.2)
            x = self.dropout(x)
            # Layer 4
            x = self.conv3(x)
            x = self.batchnorm16(x)
            x = F.leaky_relu(x, 0.2)
            # Layer 5
            x = self.conv4(x)
            x = torch.tanh(x)
            return x

        else:
            # Layer 1
            x = self.fc1(x)
            x = x.view(-1, 128, 8, 8)
            x = F.leaky_relu(x, 0.2)
            x = self.dropout(x)
            # Layer 2
            x = self.sample(x)
            x = self.conv1(x)
            x = self.batchnorm64(x)
            x = F.leaky_relu(x, 0.2)
            x = self.dropout(x)
            # Layer 3 ...
            # Layer 4 ...
            # Layer 5 ...
            return x
```

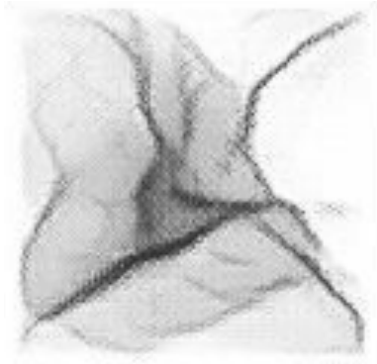
The Discriminator Model

```python
class Discriminator_128(nn.Module):
    def __init__(self, spectral_norm=False, n_conv=5):
        super(Discriminator_128, self).__init__()
        self.sn = spectral_norm

        if self.sn == False:
            # Layer: Convolutional # input 128x128 output 128x128
            self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
            # Layer: Convolutional  # input 128x128 output 64x64
            self.conv2 = nn.Conv2d(16, 32, kernel_size=4, stride=2, padding=1)
            # Layer: Convolutional  # input 64x64 output 64x64
            self.conv3 = nn.Conv2d(32, 50, kernel_size=3, stride=1, padding=1)
            # Layer: Convolutional  # input 64x64 output 32x32
            self.conv4 = nn.Conv2d(50, 100, kernel_size=4, stride=2, padding=1)
            # Layer: Convolutional # input 32x32 output 32x32
            self.conv5 = nn.Conv2d(100, 128, kernel_size=3, stride=1, padding=1)
            # Layer: Convolutional # input 32x32 output 16x16
            self.conv6 = nn.Conv2d(128, 200, kernel_size=4, stride=2, padding=1)
            # Layer: Convolutional  # input 16x16 output 8x8
            self.conv7 = nn.Conv2d(200, 256, kernel_size=4, stride=2, padding=1)
            # Layer: Convolutional # input 8x8 output 8x8
            self.conv8 = nn.Conv2d(256, 300, kernel_size=3, stride=1, padding=1)
            # Layer: Convolutional # input 8x8 output 4x4
            self.conv9 = nn.Conv2d(300, 400, kernel_size=4, stride=2, padding=1)
            # Layer: Convolutional # input 4x4 output 4x4
            self.conv10 = nn.Conv2d(400, 512, kernel_size=3, stride=1, padding=1)
            self.fc1 = nn.Linear(512 * 4 * 4, 1)

        else:
            # Adapted from https://github.com/christiancosgrove/pytorch-spectral-normalization-gan
            sn_fn = torch.nn.utils.spectral_norm
            self.conv1 = sn_fn(torch.nn.Conv2d(1, 64, 3, stride=1, padding=(1, 1)))
            #... other layer ...
            self.fc = sn_fn(torch.nn.Linear(16 * 16 * 512, 1))
            self.act = torch.nn.LeakyReLU(0.1)

        # BatchNorm
        self.batchnorm32 = nn.BatchNorm2d(32)
        self.batchnorm50 = nn.BatchNorm2d(50)
        self.batchnorm100 = nn.BatchNorm2d(100)
        self.batchnorm128 = nn.BatchNorm2d(128)
        self.batchnorm200 = nn.BatchNorm2d(200)
        self.batchnorm256 = nn.BatchNorm2d(256)
        self.batchnorm300 = nn.BatchNorm2d(300)
        self.batchnorm400 = nn.BatchNorm2d(400)
        self.batchnorm512 = nn.BatchNorm2d(512)
```

The discriminator, similarly to the generator, is composed of a constructor where the modules to be utilized are defined, followed by a forward method that applies these PyTorch modules to determine whether an image is real or generated. In each experiment, the number of layers and various options (such as the number of channels, kernel size, etc.) were slightly altered.

The structure of the model from experiment 22 is displayed, consisting of 10 convolutional layers and 1 linear layer.

(It is worth mentioning that the spectral normalization was also implemented but was not taken into consideration during the main experiments due to time constraints)



```python
def forward(self, x):
        if self.sn == False:
            # Layer 1
            x = self.conv1(x)
            x = F.leaky_relu(x, 0.2)
            # Layer 2
            x = self.conv2(x)
            x = self.batchnorm32(x)
            x = F.leaky_relu(x, 0.2)
            # Layer 3
            x = self.conv3(x)
            x = self.batchnorm50(x)
            x = F.leaky_relu(x, 0.2)
            # Layer 4
            x = self.conv4(x)
            x = self.batchnorm100(x)
            x = F.leaky_relu(x, 0.2)
            # Layer 5
            x = self.conv5(x)
            x = self.batchnorm128(x)
            x = F.leaky_relu(x, 0.2)
            # Layer 6
            x = self.conv6(x)
            x = self.batchnorm200(x)
            x = F.leaky_relu(x, 0.2)
            # Layer 6
            x = self.conv7(x)
            x = self.batchnorm256(x)
            x = F.leaky_relu(x, 0.2)
            # Layer 7
            x = self.conv8(x)
            x = self.batchnorm300(x)
            x = F.leaky_relu(x, 0.2)
            # Layer 8
            x = self.conv9(x)
            x = self.batchnorm400(x)
            x = F.leaky_relu(x, 0.2)
            x = self.conv10(x)
            x = self.batchnorm512(x)
            x = F.leaky_relu(x, 0.2)
            # Layer 8
            x = x.view(-1, 512 * 4 * 4)
            x = self.fc1(x)
            x = torch.sigmoid(x)
            return x
        else:

            m = self.act(self.conv1(x))
            m = self.act(self.conv2(m))
            #... other layer ...
            m = self.act(self.conv7(m))
            m = self.fc(m.view(-1, 16 * 16

            return torch.sigmoid(m)
```

Training Procedure 1

```python
def main():
    model_params = ModelParams()  # initialize model defaults parameters

    parser = argparse.ArgumentParser(description=__doc__)

    parser.add_argument("--mode", type=str, help="start training", default="train")
    parser.add_argument("--out", type=str, help="out directory", default=model_params.out_dir)

    #... Other command option implementation...

    if args.img_dim == 128:
        #create the istance of the generator
        generator = Generator_128(transpose=model_params.generator_transpose)
        discriminator = Discriminator_128(
            spectral_norm=model_params.discriminator_spectral_norm
        )
    else:
        print("unknown img_dim:", args.img_dim)
        exit()


    if args.mode == "train":
        # create dataset
        dataset = Data_L(data_folder=os.path.join("script", "input", str(args.img_dim)))
        # take parameters
        train_params = TrainParams(
            train_size=dataset.train_size, seed=args.random_seed, telegram=args.telegram
        )

        # mi salvo i parametri nella cartella di uscita
        os.makedirs(args.out, exist_ok=True)
        train_params.save_params(args.out)
        print(f"train params: {train_params.__dict__}")

        train(
            generator,
            discriminator,
            train_dataloader=dataset.get_train_set(batch_size=train_params.batch_size),
            val_dataloader=dataset.get_val_set(batch_size=train_params.batch_size),
            params=train_params,
            out_dir=args.out,
            device=args.device,
        )

        #...
```

The script start in the `main.py` file where after an initialization of the model parameter (default parameters like: output dir, img dim., etc...)and the creation of all the command options check if you want to train the model.

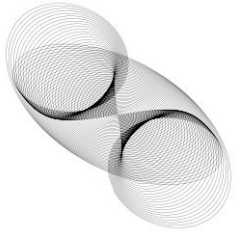
Subsequently the models of the generator and of the discriminator are instantiated, first checking the type of image supported (in this case 128x128) and specifying if the instances of the models are requested with any particular meaning.

Example: Generator accepts the 'transpose' argument to indicate if you want the layers to be Conv2d Transpose instead of Upsumple + Conv2d.

Then instantiate a `Data_L` object that is a wrapping class for manage the dataset. The class automatically splits the dataset into 80%, 10% and 10% (however, it is possible to set different percentages).

Then initialize all the train parameters (batch size, learning rate, etc...) with a specific object `train_params` and with the method `.save_params()` I save the parameters in a .json file on the output directory.

Finally we start the training procedure with its own function defined in another file. The function accepts 2 '`Dataloader`' types which are used to manage the dataset 1 for the Training Set and the other for the Validation Set, taking the appropriate images for each batch size, then we give, the instance of generator and discriminator, the parameters of the training (`params`), the output directory (`out_dir`) and the device to compute the training.



Training Procedure 2

```python
def train(..., out_dir,...):
    # initialize validation metrics
    psnr, ssim_v ,swd = 0

    # initialize log metrics
    step,g_step, epochs_saved = 0
    g_loss = torch.tensor(0).to(device)

    generator.train().to(device)
    discriminator.train().to(device)

    checkpints_dir = os.path.join(out_dir, "checkpoint")
    # optimizers
    optimizer_G = torch.optim.Adam(
        generator.parameters(), lr=params.rate_g, betas=params.betas
    )
    optimizer_D = torch.optim.SGD(discriminator.parameters(), lr=params.rate_d)

    # Checkpoint load
    if ((os.path.isdir(checkpints_dir)):
        print(f"trying to load latest checkpoint from directory {checkpints_dir}")
        checkpoint = latest_checkpoint(checkpints_dir, _checkpoint_base_name)

    if checkpoint is not None:
        if os.path.isfile(checkpoint):
            step, epochs_saved, g_step = load_from_checkpoint()

    # Saving the model architecture
    save_model(...)
    # ---------------------------------Training Loop -----------------------------------

    for epoch in range(epochs_saved, params.epoch):
        for real_samples, _ in train_dataloader:
            # D step
            z = generate_noise(real_samples.size(0), dim=100, n_distributions=3, device=device)
            fake_samples = generator(z)
            optimizer_D.zero_grad()
            d_loss = discriminator_loss(discriminator, real_samples.to(device), fake_samples, params)
            d_loss.backward()
            optimizer_D.step()
            # G step
            if step % params.disc_steps == 0:
                optimizer_G.zero_grad()
                fake_samples = generator(
                    generate_noise(real_samples.size(0), 100, 3, device=device)
                )
                g_loss = generator_loss(discriminator, fake_samples, params.loss)
                g_loss.backward()
                optimizer_G.step()

            if step % params.steps_per_val == 0 and step > 0:
                #validation:
                (psnr,ssim_v, swd, ...) = validation(val_dataloader, generator, discriminator)
            #logging
            log(...)

            step += 1

    #Saving the Final Model after All the training Phase
    save_checkpoint(...)
```

In the `train.py` function, the training process is carried out as follows:
1) First, the Generator and Discriminator are set to training mode and passed to a specific device (CPU or GPU).
2) The optimizer is set, with Adam used for the Generator and Stochastic Gradient Descent for the Discriminator.
3) The checkpoint is loaded, if present.
4) The model architecture is saved in a .json file in the output folder.
5) The training loop is initiated, cycling through the entire dataset `params.epoch` times, as defined by the number of epochs in the `TrainParams` object. For each epoch, the loop cycles through the batches of data from the `train_dataloader`.

Training the Discriminator:
- For each batch, noise of equal size is generated and input into the generator. The generator's forward method returns a tensor (of batch size, 1 channel, 128 pixels, 128 pixels).
- The tensor returned by the generator and the original images are used to calculate the discriminator's loss.
- Compute the gradient.
- The discriminator's weights are finally updated.

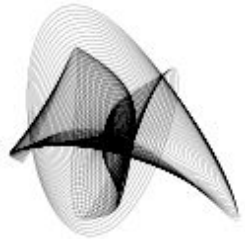
Every so often (every multiple of `.disc_steps`), the Generator is trained:
- Noise is generated again and used to create a generated image, which is then input into the Discriminator to obtain the Generator's loss.
- The Generator's weights are finally updated.

Every so often (every multiple of `.steps_per_val`), the model is evaluated.

Finally, the `log()` function is called to: save checkpoints, save any validation data, update the data displayed in the command line, and clean up any old checkpoints to optimize disk space.

Loss

Different types of loss can be used in the script, but during the experiments the '`standard`' was mainly chosen (i.e. a slight change of the BCE: in GAN papers, the loss function to optimize G is min (log 1-D) accordance with the objective presented in the original paper, but in practice use max (log D) or min -(logD) because the first formulation, which is the '`js`' type in the code, has vanishing gradients problem).



BCE Loss:
- D Loss = maximize the probability of the discriminator D correctly classifies real image and the probability of the discriminator correctly classifying the generated image G(z).

-G loss = is defined as the negative expectation of the log probability of the discriminator, D, classifying the generated image G(z) as real.

Wasserstein Loss:
(from the algorithm in the paper)
-D Loss = - (average D score on real images) + (average D score on fake images)

-G Loss = -(average D score on fake images)

```python
def compute_gradient_penalty(discriminator, real_samples, fake_samples, device="cpu"):
    """Calculates the gradient penalty loss"""
    # Random weight term for interpolation between real and fake samples
    alpha = torch.rand(
        [real_samples.size(0), 1, real_samples.size(2), real_samples.size(2)],
        device=device,
    )

    # Get random interpolation between real and fake samples
    interpolates = (
        alpha * real_samples + ((1.0 - alpha) * fake_samples)
    ).requires_grad_(True)
    d_interpolates = discriminator(interpolates)
    fake = torch.ones([real_samples.shape[0], 1], requires_grad=False, device=device)

    gradients = torch.autograd.grad(
        outputs=d_interpolates,
        inputs=interpolates,
        grad_outputs=fake,
        create_graph=True,
        retain_graph=True,
        only_inputs=True,
    )[0]
    gradients = gradients.view(gradients.size(0), -1)
    gradient_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean()
    return gradient_penalty
```

Gradient Penalty: If enable, it modifies the Discriminator loss by a factor calculated using the gradient of the interpolation between true and false images

```python
def generator_loss(discriminator, fake, loss_type):
    if loss_type == "standard":
        return -(torch.log(discriminator(fake))).mean()

    elif loss_type == "js":
        return (torch.log(1.0 - discriminator(fake))).mean()

    elif loss_type == "wasserstein" or loss_type == "hinge":
        return -(discriminator(fake)).mean()


def discriminator_loss(discriminator, real, fake, params, device="cpu"):
    loss = 0

    if params.loss == "standard" or params.loss == "js":
        loss = (
            -(torch.log(discriminator(real))).mean()
            - (torch.log(1 - discriminator(fake))).mean()
        )

    elif params.loss == "wasserstein":
        real_validity = discriminator(real).mean()
        fake_validity = discriminator(fake).mean()
        loss = -real_validity + fake_validity


    if params.gradient_penalty > 0.0:
        loss += params.gradient_penalty * compute_gradient_penalty(
            discriminator, real.data, fake.data, device=device
        )
    return loss
```

Evaluation

```python
def evaluate_generator(generator, real_data_loader, device="cpu", keep_training=True):

    generator.eval().to(device)
    with torch.no_grad():
        for img, _ in tqdm.tqdm(real_data_loader, leave=False):
            noise = generate_noise(batch_size=img.size(0), dim=100, n_distributions=3, device=device)
            fake_img = generator(noise.to(device))
            swd_value += swd(fake_img, img.to(device), device=device)
            psnr += calculate_psnr(fake_img, img.to(device))
            ssim_value += calculate_ssim(fake_img.to(device), img.to(device))

        swd_value = swd_value / len(real_data_loader)
        psnr = psnr / len(real_data_loader)
        ssim_value = ssim_value / len(real_data_loader)

    if keep_training:
        generator.train().to(device)

    return psnr.item(), ssim_value.item(), swd_value.item()


def calculate_ssim(img1, img2, data_range=1):
    return ssim(img1, img2, data_range=data_range)


def calculate_psnr(gen_images, real_images):
    mse = torch.mean((gen_images - real_images) ** 2)
    psnr = 10 * torch.log10(1 / mse)
    return psnr
```

In order to evaluate the performance of the GAN, several metrics were used.
For the generator:
- the Structural Similarity Index (SSIM),
- Peak Signal-to-Noise Ratio (PSNR),
- Sliced Wasserstein Distance (SWD)

were used to quantify the similarity between the generated and real Lissajous figures.

The implementations of SWD and SSIM were taken from Github repositories.

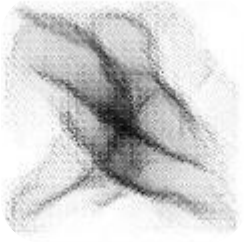
For the PSNR we use the definition.

Then we take the mean of all batches.

(The metrics used for the discriminator (accuracy, f1-scrore, ...) were largely ignored as they provided inconclusive results)

While these metrics provided quantitative evaluations of the generated figures, **the primary factor in determining the success of the GAN was subjective human evaluation**.
The generated figures that were deemed to have the most visually appealing and accurate Lissajous-like patterns were selected.
The focus of the evaluation was on the ability of the generator to produce visually convincing Lissajous figures.

NO



OK

Result

The results of our experiments show several key insights regarding the development of our Generative Adversarial Network (GAN) for generating Lissajous figures.

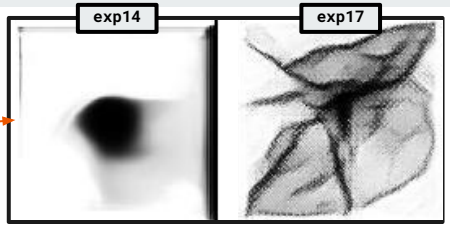
1. The use of upsample and conv2d (exp14) instead of conv2d transpose (exp17) significantly reduced the quality of the generated images (exp14 vs exp17).

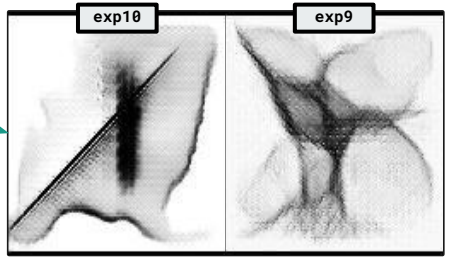
2. The introduction of a gradient penalty (exp9), although it increased the execution time, had a positive effect on the final image quality (exp10 vs exp9).

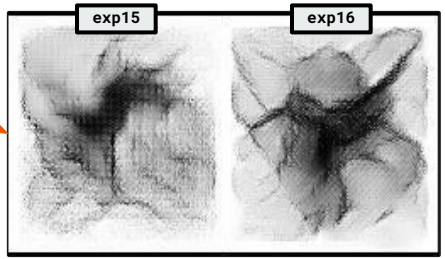
3. The results indicate that training the discriminator more heavily (exp16) than the generator leads to better quality images (using `.disc_steps` > 1). (15 vs 16).

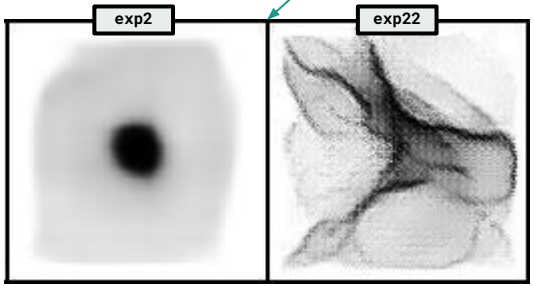
4. Our results showed that there was no difference between using the Wasserstein Loss and the Binary Cross Entropy (BCE) Loss.

5. The results obtained using metrics such as SSIM, SWD, and PSNR often contrasted with human perception (exp2 best for SSIM vs exp22 the best at a glance).









(A complete list of all the results is available in the report)