UNIVERSITY OF
EASTERN FINLAND

AI-22 Report Template
GAN for Generative Art

Done By:
Haritha Weerathunga Arachchige
Olumide Ojo
Faria Ferdowsy
Nada Drissi El Bouzaidi

1. Abstract:

Generative Adversarial Networks, or GANs for short, is an image-to-image translation and deep-learning techniques that propagates a synthetic interpretation with a certain variation. To perform with a good result, deep learning algorithms require a large dataset that provides maximum accuracy. In this work, we propound a novel generative adversarial network application that turns photographs into paintings. CycleGAN is able to translate images of one kind to another without needing pairs of images. In our project, we train two generators and this is a necessary criterion for training. We present a large-scale architectural style dataset and 25-new classes. We trained it according to forward mapping and inverse mapping. Using several architectures accommodates our model to acquire what a building looks like as well as what is its main common features. After that, the model successfully generates an artistic structure. Here, we captured images of our city and placed them in our model to transform the images into art.

2. Work Contribution:

Haritha:

- Training the CycleGAN model using an existing art-generating dataset
- Improving the accuracy of the model

Nada:

- Working collectively with Haritha to implement the best-optimized cycleGAN for generating art
- Writing sections 2, 4, and 5 in the report

Olumide:

- Collecting pictures needed for the testing part
- Testing them following the trained model

Faria:

- Informing the members of the following meetings and following the improvement made by each one of them
- Writing sections 1, 3, and 6 in the report

3. Introduction:

CycleGAN is an architecture introduced in 2017 designed specifically to perform image-to-image translation on unpaired sets of images. GAN is a solid way of training a generative pattern by framing the problem as a supervised learning problem with 2

sub-models which are the generator model and the discriminator. The two generators are often variations of autoencoders and take an image as input and return an image as output. The discriminator takes an image as input and outputs one single number. The CycleGAN has two objectives for its main generator, generator one. Here, the discriminator goal is to classify whether an image is a real picture, or a fake picture made up by the generator. The generator training alongside the discriminator tries to make images that fool the discriminator into thinking they are real. In some cases, images can be split up into two distinct domains. The first generator translates the image then the second generator learns to translate the image back to the original.

Let’s consider we have 2 datasets: one contains paintings, and the other contains photographs. In order to convert images into paintings or conversely, we need to learn a mapping between paintings and photos. The photos in our datasets do not, however, correspond exactly to one another. CycleGAN has already attempted to solve the unpaired picture translation problem. Two generators are trained in oppositional directions using CycleGAN. If we train a CycleGAN using a painting dataset and a photograph dataset, it will simultaneously develop a model for turning photographs into paintings and paintings into photographs. The notion is that the output should be somewhat close to the input as we move from the photograph domain to the painting domain and back again. In CycleGAN, a cycle consistency loss that reduces the difference between the input and the reconstructed image enforces this behavior. This loss function is utilized in addition to the loss that results from the discriminators' ability to determine whether the input is a produced image or a real image from the dataset. However, training generators and discriminators using this model can have many issues:

- One problem with CycleGAN is that, even if we are only concerned regarding a particular direction, we must always train two generators and that's what we have to do in this project, as well. However, training would take longer and that's what we notice when we were training our model to learn the patterns.
- In addition, the training environment would assume that converting a photograph into a painting and vice versa are equally difficult and significant tasks. This could not always be the situation. I contend that it is simpler to convert a photograph into a painting than the reverse. Therefore, learning inverse mapping shouldn't need much effort if our application simply does the one-way mapping.
- CycleGAN makes the assumption that the connection between two domains is invertible and has a 1-to-1 correspondence, which

may not always be the case. For instance, for a particular photo, there can be several viable painting counterparts.

Overall, the cycle consistency loss is a decent proposal, but it is not flawless. The study discovered that CycleGAN is actually quite effective at concealing data to meet the cyclic consistency criteria.

4. Datasets:

We decided to use a dataset that we found in Kaggle, made public in a research paper called “Architectural Style Classification Using Multinomial Latent Logistic Regression”. Due to a lack of publicly accessible data sources, we present a brand-new, 25-class, large-scale architectural-style dataset.
Some of the 25 styles employed are American craftsman design, Ancient Egyptian architecture, Deconstructivism, Chicago school architecture, Baroque style, Colonial-era building, Gothic architecture, Queen Anne architecture, Romanesque architecture, Russian Revival architecture, and Georgian architecture.
Therefore, we notice that the use of different architectures would allow our model to learn how a building is and what are its main structures and then transform it into an artistic one.

Now, let’s understand how the training environment of the model works:

- We have a sample of input. Just as for illustration, let's say we have a generator that transforms horses into zebras. It can also be taught to create paintings from images.
- We also have a discriminator that the generator must persuade that the zebras it produces appear identical to genuine ones.
- When the generator transforms the horses into zebras, then back to a zebra image, we should get the very same input photo back. That is what we call “cycle consistency”

```python
class ArtAndArchitecureDataset(Dataset):

    def __init__(self, root_architecture, root_arts, transform= None):

        self.root_architecture = root_architecture
        self.root_arts = root_arts
        self.transform = transform
        self.architecture_images = os.listdir(root_architecture)
        self.art_images = os.listdir(root_arts)
        self.length_dataset = max(len(self.architecture_images), len(self.art_images)) # 1000, 1500
        self.architecture_len = len(self.architecture_images)
        self.art_len = len(self.art_images)

    def __len__(self):
        return self.length_dataset

    def __getitem__(self, index):
        archi_image = self.architecture_images[index % self.architecture_len]
        art_image = self.art_images[index % self.art_len]

        archi_path = os.path.join(self.root_architecture, archi_image)
        art_path = os.path.join(self.root_arts, art_image)

        archi_image = np.array(Image.open(archi_path).convert("RGB"))
        art_image = np.array(Image.open(art_path).convert("RGB"))

        if self.transform:
            augmentations = self.transform(image=archi_image, image0=art_image)
            archi_image = augmentations["image"]
            art_image = augmentations["image0"]

        return archi_image, art_image
```

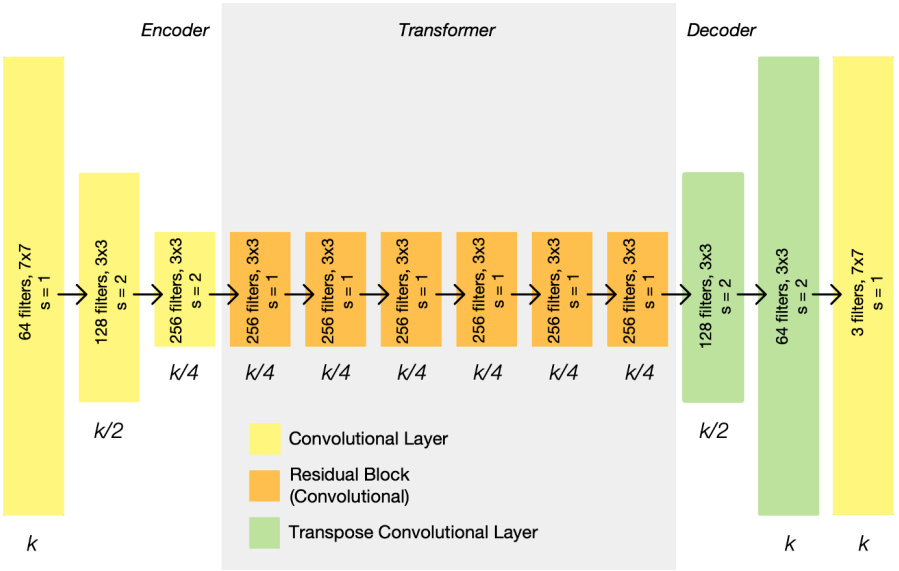
- The following is the implementation we coded to initialize the dataset.
- In the initialization part, we tried to include length_dataset since in our dataset we might not always have the same number of images of both documents containing data. We did that by checking the maximum length of each dataset.
- In the len function, we just return the length of each dataset.
- In the function getitem, we get an index between for example 0 to 1499 (if we suppose the length of the dataset is 1500), of the art image and the building image. Since the index could be greater than the dataset that we have because we are taking the maximum of the 2 not only 1, therefore we use % to fix this issue.
- Archi_path and art_path are used to load the image. Then, we convert them into an RGB image.

5. Model Description:

Let's understand the architecture of the model. Let's start first with the generator:

- Encoder, transformer, and decoder are the 3 components that make up each CycleGAN generator.
- The encoder receives the input photo and reduces the output size while increasing the number of channels. There are 3 convolutional layers in the encoder.

- The transformer, a collection of 6 residual blocks, receives the resulting activation after that.
- It is then extended one more by the decoder, which creates the final RGB photo using one output layer and 2 transpose convolutions to increase the representation size.
- The instance normalization and ReLU layers that come after each layer have been left out for the sake of simplicity.
- The details are illustrated in the image below.



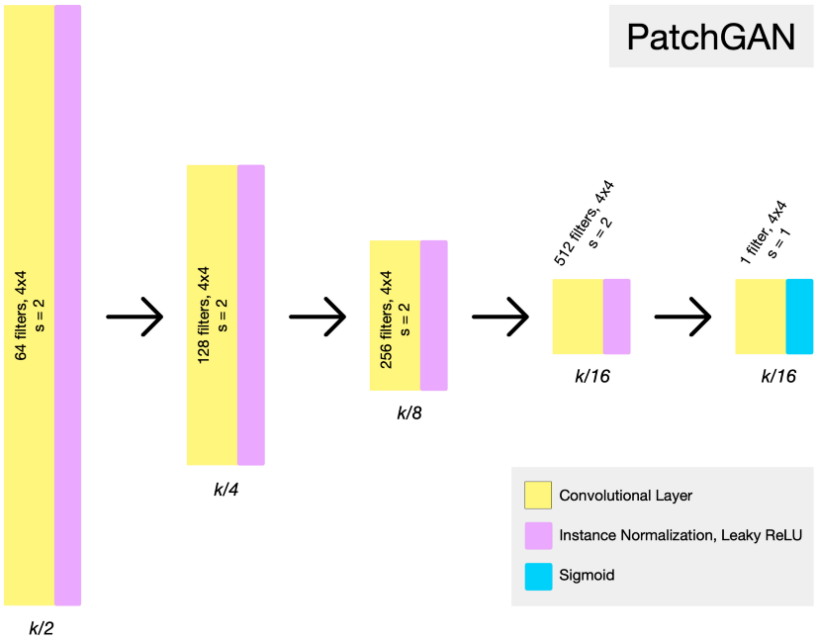
- This is exactly what we implemented in our model.
- In the down_block, e started with a normal initial image, then we started encoding it using the convolutional layer. In the code, you can see both instance normalization and the ReLU function being implemented.
- Then, we implemented res_block, which is a block of residuals.
- Finally, we have the up_block, we each the decoder of the image using the transpose convolutional layer.

```python
class Generator(nn.Module):
    def __init__(self, img_channels, num_features = 64, num_residuals=9):
        super().__init__()
        self.initial = nn.Sequential(
            nn.Conv2d(img_channels, num_features, kernel_size=7, stride=1, padding=3, padding_mode="reflect"),
            nn.InstanceNorm2d(num_features),
            nn.ReLU(inplace=True),
        )
        self.down_blocks = nn.ModuleList(
            [
                ConvBlock(num_features, num_features*2, kernel_size=3, stride=2, padding=1),
                ConvBlock(num_features*2, num_features*4, kernel_size=3, stride=2, padding=1),
            ]
        )
        self.res_blocks = nn.Sequential(
            *[ResidualBlock(num_features*4) for _ in range(num_residuals)]
        )
        self.up_blocks = nn.ModuleList(
            [
                ConvBlock(num_features*4, num_features*2, down=False, kernel_size=3, stride=2, padding=1, output_padding=1),
                ConvBlock(num_features*2, num_features*1, down=False, kernel_size=3, stride=2, padding=1, output_padding=1),
            ]
        )

        self.last = nn.Conv2d(num_features*1, img_channels, kernel_size=7, stride=1, padding=3, padding_mode="reflect")
```

Moving on to the discriminator, we find the following architecture:

- The discriminators are PatchGANs, fully CNN that output the probability that a patch of the input image is "real" by examining it. This is both more successful and more computationally practical than attempting to analyze the complete input photo since it enables the discriminator to target surface-level features.
- Up until the required output size is attained, PatchGAN divides the representation size in half and multiplies the number of channels by 2. In this instance, using the PatchGAN to analyze input patches with a size of 70x70 worked best.



- One basic component that serves as the basis for the discriminator is Ck which is a 4 x 4 Convolution InstanceNorm-LeakyReLU layer with k kernels and stride 2. Here, we used 4 as the kernel size. With that taken into consideration, we can build the discriminator, with an architecture of C64, C128, C256, and C512

```
class Discriminator(nn.Module):
    def __init__(self, in_channels=3, features=[64, 128, 256, 512]):
        super().__init__()
        self.initial = nn.Sequential(
            nn.Conv2d(
                in_channels,
                features[0],
                kernel_size=4,
                stride=2,
                padding=1,
                padding_mode="reflect",
            ),
            nn.LeakyReLU(0.2, inplace=True),
        )

        layers = []
        in_channels = features[0]
        for feature in features[1:]:
            layers.append(Block(in_channels, feature, stride=1 if feature==features[-1] else 2))
            in_channels = feature
        layers.append(nn.Conv2d(in_channels, 1, kernel_size=4, stride=1, padding=1, padding_mode="reflect"))
        self.model = nn.Sequential(*layers)
```

- We add a final Conv2d layer and, as advised, turn off InstanceNormalization in the initial Ck layer.

```
class Block(nn.Module):
    def __init__(self, in_channels, out_channels, stride):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 4, stride, 1, bias=True, padding_mode="reflect"),
            nn.InstanceNorm2d(out_channels),
            nn.LeakyReLU(0.2, inplace=True),
```

For the training of the model, since we had a large amount of data (around 6738 pictures). We ended up training 4090 photos from the given data which represent roughly 61% of the data. We suspected that it will be good enough for testing it on a real model since we were able to generate art from the different architectures.
We also included the sum of identity loss, adversarial loss, and cycle loss for both generators in the implementation.

```
  0%|          | 1/6738 [00:03<6:07:20,  3.27s/it, H_fake=0.246, H_real=0.675]saving images
  3%|▏         | 201/6738 [00:50<27:49,  3.92it/s, H_fake=0.372, H_real=0.623]saving images
  6%|▏         | 401/6738 [01:39<27:12,  3.88it/s, H_fake=0.378, H_real=0.619]saving images
  9%|▎         | 601/6738 [02:27<26:12,  3.90it/s, H_fake=0.381, H_real=0.615]saving images
 12%|▎         | 801/6738 [03:16<24:58,  3.96it/s, H_fake=0.38, H_real=0.616]saving images
 15%|▍         | 1001/6738 [04:04<24:30,  3.90it/s, H_fake=0.378, H_real=0.615]saving images
 18%|▍         | 1201/6738 [04:53<23:20,  3.95it/s, H_fake=0.378, H_real=0.617]saving images
 21%|▌         | 1401/6738 [05:41<22:38,  3.93it/s, H_fake=0.377, H_real=0.618]saving images
 24%|▌         | 1601/6738 [06:29<21:56,  3.90it/s, H_fake=0.376, H_real=0.619]saving images
 27%|▋         | 1801/6738 [07:18<21:08,  3.89it/s, H_fake=0.376, H_real=0.617]saving images
 30%|▋         | 2001/6738 [08:06<20:06,  3.93it/s, H_fake=0.377, H_real=0.617]saving images
 33%|▊         | 2201/6738 [08:55<19:11,  3.94it/s, H_fake=0.377, H_real=0.616]saving images
 36%|▊         | 2401/6738 [09:43<18:19,  3.94it/s, H_fake=0.378, H_real=0.615]saving images
 39%|▉         | 2601/6738 [10:32<17:35,  3.92it/s, H_fake=0.377, H_real=0.617]saving images
 42%|▉         | 2801/6738 [11:20<16:46,  3.91it/s, H_fake=0.377, H_real=0.617]saving images
 45%|▉         | 3001/6738 [12:08<15:45,  3.95it/s, H_fake=0.377, H_real=0.617]saving images
 48%|▉         | 3201/6738 [12:56<15:05,  3.91it/s, H_fake=0.377, H_real=0.618]saving images
 50%|▉         | 3401/6738 [13:45<14:19,  3.88it/s, H_fake=0.377, H_real=0.617]saving images
 53%|▉         | 3601/6738 [14:34<13:29,  3.88it/s, H_fake=0.377, H_real=0.617]saving images
 56%|▉         | 3801/6738 [15:22<12:35,  3.89it/s, H_fake=0.376, H_real=0.618]saving images
 59%|▉         | 4001/6738 [16:10<11:37,  3.92it/s, H_fake=0.376, H_real=0.617]saving images
 61%|▉         | 4090/6738 [16:32<10:42,  4.12it/s, H_fake=0.377, H_real=0.617]
```

We also tried to optimize our learning rate since if it is not updated it will end up having the same value as the old checkpoints. Hence, many hours of debugging.

```python
def load_checkpoint(checkpoint_file, model, optimizer, lr):
    print("=> Loading checkpoint")
    checkpoint = torch.load(checkpoint_file, map_location=device)
    model.load_state_dict(checkpoint["state_dict"])
    optimizer.load_state_dict(checkpoint["optimizer"])

    for param_group in optimizer.param_groups:
        param_group["lr"] = lr
```

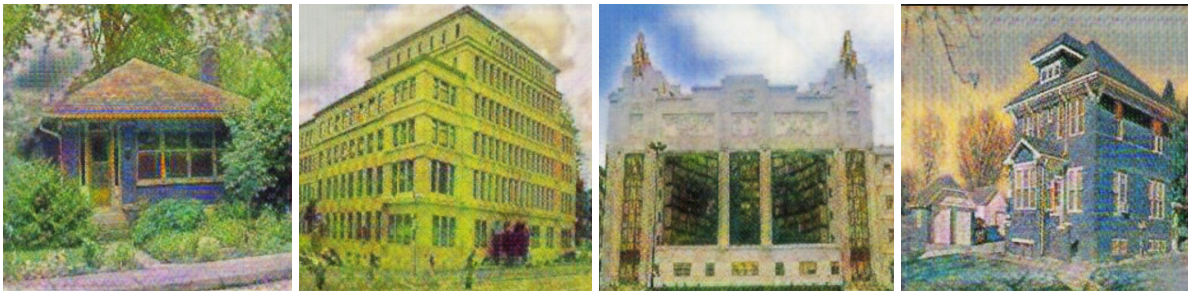
6. Experiments and Results:

For the training data, after running it for hours and hours. We got successful that seemed to be good to move to test the data on our images.
For the sake of using inverse mapping, our model considered both transforming photos into art and vice versa. So, we had 2 datasets: one of the images is about the architecture of the building, and another one is about art.
Some of the samples found in our dataset are:



Some art images generated from the architecture dataset at the first stages are:

Then, after training for a long time, the quality of the images gets better as the model is familiarizing itself with the common patterns of the photos.



In the last stage of this project, we were supposed to take pictures of Joensuu and test them with our trained model to convert those pictures into art.
The problem that we forgot about when selecting the dataset, is that our model is only trained to learn common features of different building architectures. So, we decided to make a small experiment in the testing stage and try it on both landscapes and buildings.
The pictures below show the results we got from testing the model on both a building with a landscape and a normal landscape. We noticed that our model could recognize patterns and learn them to come up with an artistic drawing of a real photo.
The results truly look like a painting, however, it is not as perfect as we wanted them to be.

Surprisingly, some of the art generated were quite good. The patterns were respected and the artistic effect was noticeable. We would conclude that our model worked quite perfectly with not only buildings but also landscapes.
Some examples would be: