# Deep Learning Project - Group 20

## Transformer based Generative Adversarial Networks (T-GAN) for medical image generation


### Group Members

- BA12-045 - Ứng Quốc Thành Đạt
- BI12-210 - Nguyễn Tuấn Khiêm
- BI12-148 - Hoàng Hải
- 22BI13412 - Trần Minh Thành
- 22BI13438 - Nguyễn Hoàng Trung
- 22BI13435 - Nguyễn Đức Trung


## 1. Background

Currently, many common diseases affect populations around us, making the management and treatment of these conditions highly dependent on accurate and extensive information. Comprehensive datasets play a critical role in enabling research and the development of effective treatment methods.

To contribute to this effort, we have developed a model based on a GAN with Transformer architecture designed to generate simulated images of common diseases. These images can serve as valuable references for researchers and healthcare professionals, expanding their knowledge and enhancing the quality of research on these diseases.

## 2. Purpose

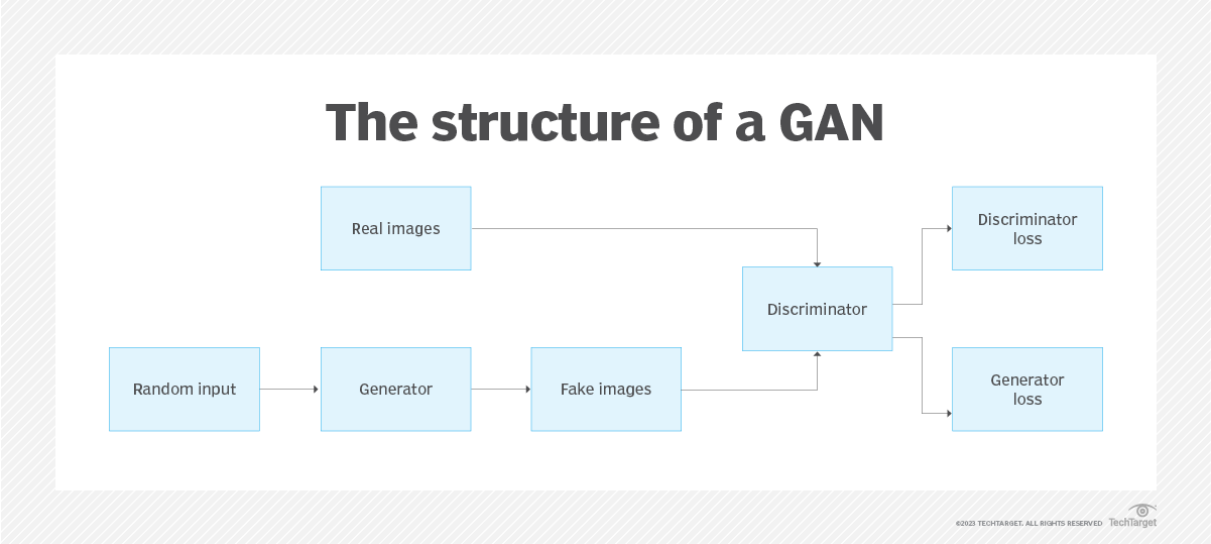
The main goal of the model is to generate a set of simulated images of common diseases with the highest realism, thereby supporting research and exploration of these diseases. The creation of simulated images can provide a richer resource for diagnosis, treatment, and the development of medical solutions.

In addition, this model aims to:

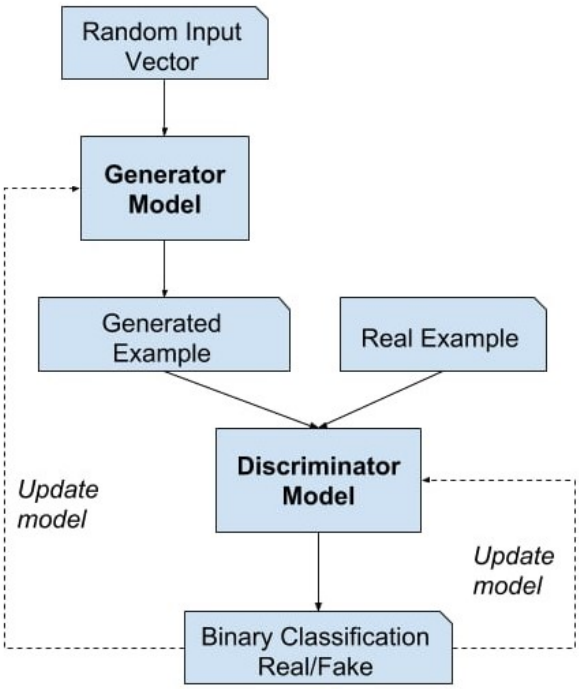
- Provide a powerful tool for developing and testing new medical image analysis methods, enhancing the ability to identify and diagnose diseases.
- Assist researchers and medical experts in generating larger datasets, thereby improving the accuracy and performance of machine learning models in the medical field.
- Help expand knowledge about diseases through the creation of clearer and more realistic images, serving research and teaching purposes.

# 3. Methodology

The **Generative Adversarial Network (GAN)** consists of two main components - the **Generator** and the **Discriminator**. The Generator creates fake images from random noise, while the Discriminator classifies these images as real (from the original dataset) or fake (generated). During training, the Discriminator computes its loss based on classification accuracy, and the Generator computes its loss based on success in fooling the Discriminator.




**GAN Architecture Diagram** illustrates the interaction between the generator and discriminator models in a GAN setup



**Dataset:** We use the MedNIST dataset, a medical dataset containing MRI scans. This dataset is preprocessed into grayscale images and normalized for the training process.

# 4. Detailed Breakdown

## 4.1 File generator.py

**Purpose:** This file defines the *Generator* class in the *DCGAN* architecture, responsible for generating fake images from random noise.

**Structure of the Generator Class:**

- **Inherited class:** The *Generator* class inherits from *nn.Module* of *PyTorch*.
- **Layer structure:**
  - The generator consists of multiple transposed convolutional layers. Each layer upsamples the feature maps and reduces the depth, gradually transforming the random noise into a 64x64 image.
  - Input parameters: *nz* (size of the noise vector), *ngf* (number of feature maps), *nc* (number of image channels).
  - It includes deconvolution layers like *ConvTranspose2d*, *BatchNorm2d*, *ReLU*, and the final activation function *Tanh*.
  - Batch normalization is applied after each layer (except the last) to stabilize the training process, while ReLU activations introduce non-linearity.

```
# First deconvolutional layer
nn.ConvTranspose2d(nz, ngf * 8, kernel_size=4, stride=1, padding=0,
bias=False),
nn.BatchNorm2d(ngf * 8),
nn.ReLU(True),
# out: (64 * 8 =) 512 x 4 x 4
```

```
# Layer 2: Upsample to 256 feature maps
nn.ConvTranspose2d(ngf * 8, ngf * 4, kernel_size=4, stride=2, padding=1,
bias=False),
nn.BatchNorm2d(ngf * 4),
nn.ReLU(True),
# out: (64 * 4 =) 256 x 8 x 8
```

```
# Output layer: Generate a gray color image
nn.ConvTranspose2d(ngf, nc, kernel_size=4, stride=2, padding=1, bias=False),
nn.Tanh()
# out: 1 x 64 x 64
```

- The final layer outputs an image with the specified number of channels (nc = 1 since the desired output is a grayscale image), using the Tanh activation function to scale the output pixel values between -1 and 1.
- The output size scales from $4 \times 4$ to $8 \times 8$, and so on, and the output is a $64 \times 64$ image. The formula for the output size is:

$$\text{Output size} = (\text{input\_size} - 1) \times \text{stride} - 2 \times \text{padding} + \text{kernel\_size}$$

- **Summary:** The Generator creates images from random noise vectors, expanding the feature space through deconvolutional layers to produce images.

## 4.2 File discriminator.py

**Purpose:** Defines the *Discriminator* class, responsible for classifying images as real or fake.

**Structure of the Discriminator Class:**

- **Inherited class:** The *Discriminator* class inherits from *nn.Module*.
- **Layer structure:**
  - The discriminator consists of multiple convolutional layers. Each layer downsamples the feature maps and increases the depth, gradually transforming the input image into a single output value.
  - Batch normalization is applied after each layer (except the last) to stabilize the training process, while Leaky ReLU activations introduce non-linearity.
  - Input parameters: *nc* (number of image channels), *ndf* (number of feature maps).
  - Includes layers such as *Conv2d*, *BatchNorm2d*, *LeakyReLU*, *Flatten*, and *Sigmoid*.

```
# Layer 1: Downsample to 64 feature maps
nn.Conv2d(nc, ndf, kernel_size=4, stride=2, padding=1, bias=False),
nn.BatchNorm2d(ndf),
nn.LeakyReLU(0.2, inplace=True),
# out: 64 x 32 x 32
```

```
# Layer 2: Downsample to 128 feature maps
nn.Conv2d(ndf, ndf * 2, kernel_size=4, stride=2, padding=1, bias=False),
nn.BatchNorm2d(ndf * 2),
nn.LeakyReLU(0.2, inplace=True),
# out: 128 x 16 x 16
```

```
# Output layer: Single value output (real or fake)
nn.Conv2d(ndf * 8, 1, kernel_size=4, stride=1, padding=0, bias=False),
# out: 1 x 1 x 1
```

  - The final layer output is a single value indicating the probability that the input image is real.
  - The output size scales from $64 \times 64$ down to $4 \times 4$, finally producing a single scalar value. The formula for the output size is:

$$\text{Output size} = \left( \frac{\text{input\_size} - \text{kernel\_size} + 2 \times \text{padding}}{\text{stride}} \right) + 1$$

- **Summary:** The Discriminator transforms images into a single output value representing the likelihood of the image being real.

## 4.3 File train.py

**Objective:** This file contains the source code for training the GAN model, including two main components: the Generator and the Discriminator. It uses a series of convolutional layers to downsample the input image and output a single probability score to determine whether they are real (from the dataset) or fake (generated by the Generator).

### 4.3.1 Initialization:

- Uses libraries such as *PyTorch*, *torchvision*, and *matplotlib*.
- Defines parameters such as *DATA_DIR*, *batch_size*, *image_size*, *nc*, *nz*, etc.
- Function `get_default_device()` to select the device (GPU/CPU)

### 4.3.2 Data preparation:

- Dataset is loaded from *Mednist*, following preprocessing steps such as: converting to grayscale, resizing, center cropping, and normalization.

```
# Define Training Dataset, DataLoader and Transformations
train_ds = ImageFolder(DATA_DIR, transform=trans.Compose([
   trans.Grayscale(num_output_channels=1),
   trans.Resize(image_size),
   trans.CenterCrop(image_size),
   trans.ToTensor(),
   trans.Normalize(*stats)
]))
```

### 4.3.3 Training process:

- **Training the Discriminator:** to take real images and the discriminator optimizer as input and compute the loss for real and fake images, updating the discriminator's weights based on the total loss.

```
# Training Discriminator
def train_discriminator(real_images, opt_d):
    # Move real_images to the same device as the model
    real_images = real_images.to(device)
    # Clear discriminator gradients
    opt_d.zero_grad()

    # Pass real images through discriminator
    real_preds = discriminator(real_images)
    real_targets = torch.ones(real_images.size(0), 1, device=device)
    real_loss = func.binary_cross_entropy(real_preds, real_targets)
    real_score = torch.mean(real_preds).item()

    # Generate fake images
    latent = torch.randn(batch_size, nz, 1, 1, device=device)
    fake_images = generator(latent)

    # Pass fake images through discriminator
    fake_targets = torch.zeros(fake_images.size(0), 1, device=device)
    fake_preds = discriminator(fake_images)
    fake_loss = func.binary_cross_entropy(fake_preds, fake_targets)
    fake_score = torch.mean(fake_preds).item()

    # Update discriminator weights
    loss = real_loss + fake_loss
    loss.backward()
    opt_d.step()
    return loss.item(), real_score, fake_score
```

- **Training the Generator:** to clear gradients, generate fake images, and compute the loss of the discriminator's predictions in order to update the generator's weights.

```
# Training Generator
def train_generator(opt_g):
    # Clear generator gradients
    opt_g.zero_grad()

    # Generate fake images
    latent = torch.randn(batch_size, nz, 1, 1, device=device)
    fake_images = generator(latent)

    # Try to fool the discriminator
    preds = discriminator(fake_images)
    targets = torch.ones(batch_size, 1, device=device)
    loss = func.binary_cross_entropy(preds, targets)

    # Update generator weights
    loss.backward()
    opt_g.step()

    return loss.item()
```

- **Start training loop:**
  - Main training loop that iterates through epochs, calling the training functions for both the discriminator and generator, and logs the losses and scores.

```
# Training loop
def fit(epochs, lr, start_idx=1):
    torch.cuda.empty_cache()

    # Losses & scores
    losses_g = []
    losses_d = []
    real_scores = []
    fake_scores = []

    # Create optimizers
    opt_d = torch.optim.Adam(discriminator.parameters(), lr=lr, betas=betas)
    opt_g = torch.optim.Adam(generator.parameters(), lr=lr, betas=betas)

    # Resume from checkpoint if exists
    start_epoch = 0
    checkpoint_path = 'model_checkpoint.pth'
    if os.path.exists(checkpoint_path):
        checkpoint = torch.load(checkpoint_path)
        generator.load_state_dict(checkpoint['generator_state_dict'])
        discriminator.load_state_dict(checkpoint['discriminator_state_dict'])
        opt_g.load_state_dict(checkpoint['optimizer_g_state_dict'])
        opt_d.load_state_dict(checkpoint['optimizer_d_state_dict'])
        start_epoch = checkpoint['epoch'] + 1
        print(f"Resuming training from epoch {start_epoch}")

    for epoch in range(epochs):
        for real_images, _ in tqdm(train_dl):
            # Train discriminator
            loss_d, real_score, fake_score = train_discriminator(real_images,
opt_d)
            # Train generator
            loss_g = train_generator(opt_g)
```

  - After each loop, the computer saves the generated images and shows the loss and scores. Once training is complete, the pre-trained model is saved for future use.

```
        # Record losses & scores
        losses_g.append(loss_g)
        losses_d.append(loss_d)
        real_scores.append(real_score)
        fake_scores.append(fake_score)

        # Log losses & scores (last batch)
        print("Epoch [{}/{}], loss_g: {:.4f}, loss_d: {:.4f}, real_score:
{:.4f}, fake_score: {:.4f}".format(
            epoch + 1, epochs, loss_g, loss_d, real_score, fake_score))

        # Save generated images
        save_samples(epoch + start_idx, fixed_latent, show=False)

        # Save the model checkpoints (including optimizer state and epoch)
        torch.save({
            'epoch': epoch,
            'generator_state_dict': generator.state_dict(),
            'discriminator_state_dict': discriminator.state_dict(),
            'optimizer_g_state_dict': opt_g.state_dict(),
            'optimizer_d_state_dict': opt_d.state_dict(),
        }, checkpoint_path)

    return losses_g, losses_d, real_scores, fake_scores
```

- **Saving results:**

```
# Denormalization function
def denorm(img_tensors):
    return img_tensors * stats[1][0] + stats[0][0]


# Show images function
def show_images(images, nmax=nmax, nrow=nrow):
    fig, ax = plt.subplots(figsize=(8, 8))
    ax.set_xticks([])
    ax.set_yticks([])
    ax.imshow(make_grid(denorm(images.detach()[:nmax]), nrow=nrow).permute(1,
2, 0))


# Show batch function
def show_batch(dl, nmax=nmax):
    for images, _ in dl:
        show_images(images, nmax)
        break
```

  - Generated images will be saved in the "generated" folder using the format "generated-images-xxxx.png" allowing us to easily locate and view each one.

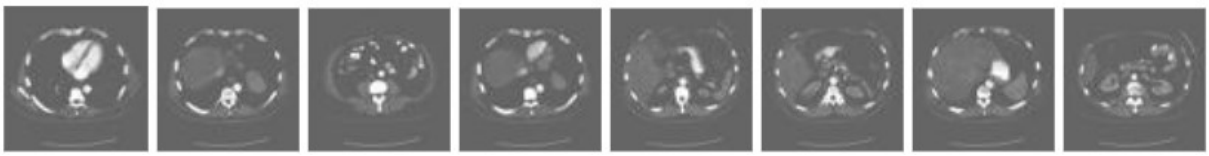
```
# Create folder to store generated images
sample_dir = 'generated'
os.makedirs(sample_dir, exist_ok=True)


# Define function to generating, saving by format and displaying images
def save_samples(index, latent_tensors, show=True):
   fake_images = generator(latent_tensors)
   fake_fname = 'generated-images-{0:0=4d}.png'.format(index)
   save_image(denorm(fake_images), os.path.join(sample_dir, fake_fname),
nrow=nrow)
   print('Saving', fake_fname)
   if show:
       fig, ax = plt.subplots(figsize=(8, 8))
       ax.set_xticks([])
       ax.set_yticks([])
       ax.imshow(make_grid(fake_images.cpu().detach(), nrow=nrow).permute(1,
2, 0))
```

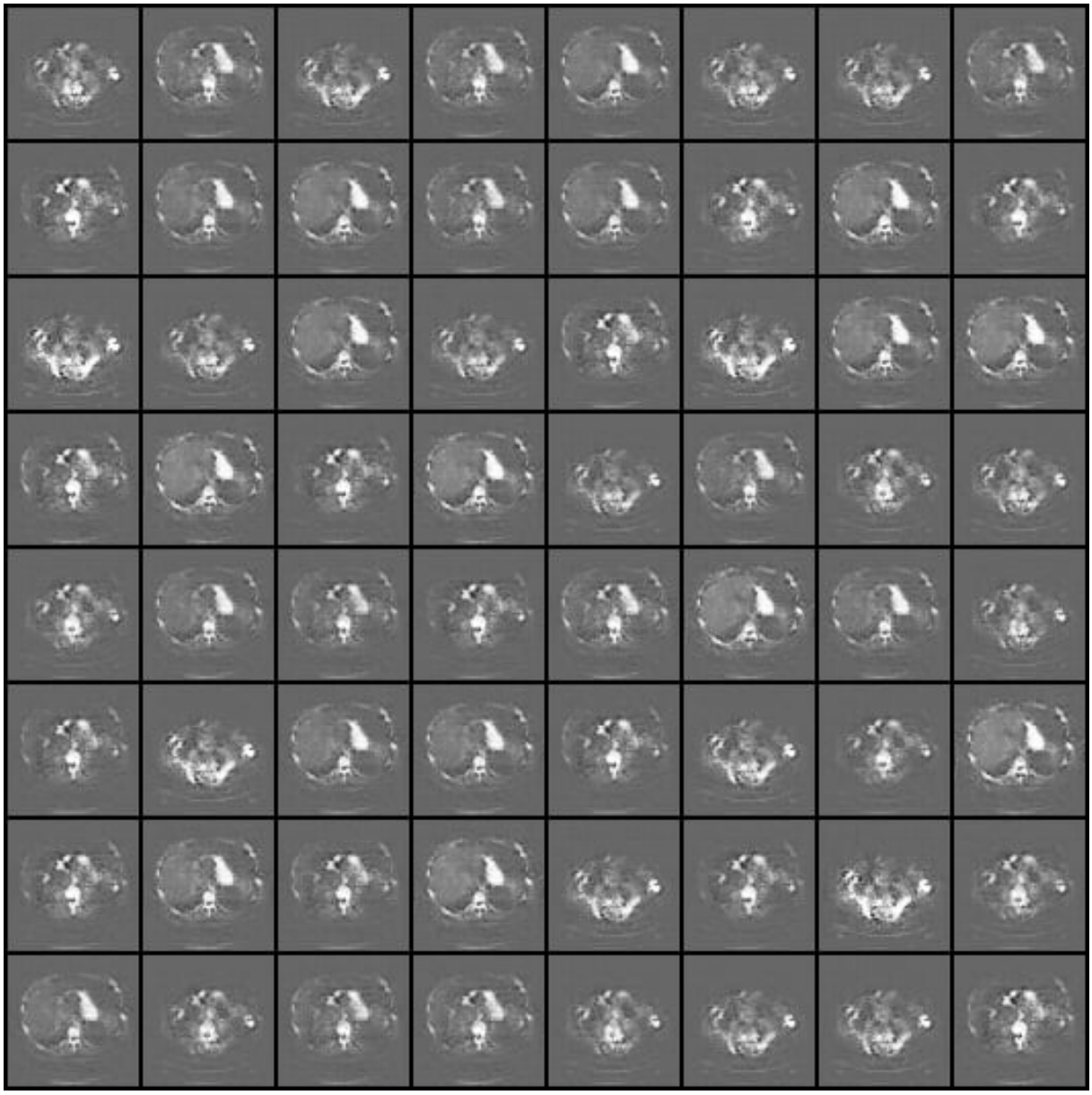
- **Summary:** The *train.py* file sets up and runs the training process for the GAN model, including the optimization of both models and storing the results.

# 5. Demonstration of the Model

We trained the model using the AbdomenCT dataset, which contains 10,000 images, to generate CT scans of the abdomen. The model was trained for 1000 epochs with a learning rate of 0.0001. Below are the results:



Real image



The generated images after the 1000th epoch.

Although the generated image has flaws like blurriness and noise, we can still see some similarity to the real image.

# 6. Conclusion

The *Transformer-based GAN* model holds great potential for enhancing the understanding of diseases. The model utilizes a dataset of common diseases to generate valuable simulated images. By continuing to refine this approach, the model can enhance its ability to produce relevant simulations, significantly contributing to medical research by providing essential references for diagnosis and treatment.

# 7. References

In this report, we used data from the Medical MNIST dataset [3]. We also looked at instructional videos on YouTube [2], along with images showing the structure of a Generative Adversarial Network (GAN) [4] and its architecture diagram [1]. Below are the sources we used for this report.


[1] Jason Brownlee. A gentle introduction to generative adversarial networks (gans), 2019. https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/.

[2] Jovian. Image generation using gans | deep learning with pytorch (6/6), 13 Jan 2021. https://youtu.be/79IvwU3G5_Q.

[3] Andrew Mvd. Medical mnist dataset on kaggle, 2020. https://www.kaggle.com/datasets/andrewmvd/medical-mnist.

[4] Kinza Yasar. generative adversarial network (gan), March 2023. https://www.techtarget.com/searchenterpriseai/definition/generative-adversarial-network-GAN.