assignment1-soc

June 20, 2024

1 Part 1

1.1 Training DCGAN on MNIST dataset

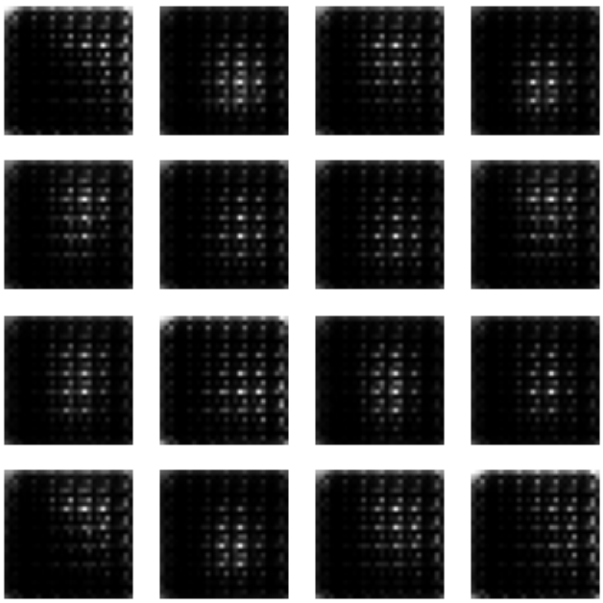
[ ]:
```
import tensorflow as tf
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import numpy as np

# Load and preprocess the MNIST dataset
(train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).
 ↪astype('float32')
train_images = (train_images - 127.5) / 127.5  # Normalize the images to [-1, 1]
BUFFER_SIZE = 60000
BATCH_SIZE = 256
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).
 ↪shuffle(BUFFER_SIZE).batch(BATCH_SIZE)

# Generator model
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1),␣
 ↪padding='same', use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2),␣
 ↪padding='same', use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
```

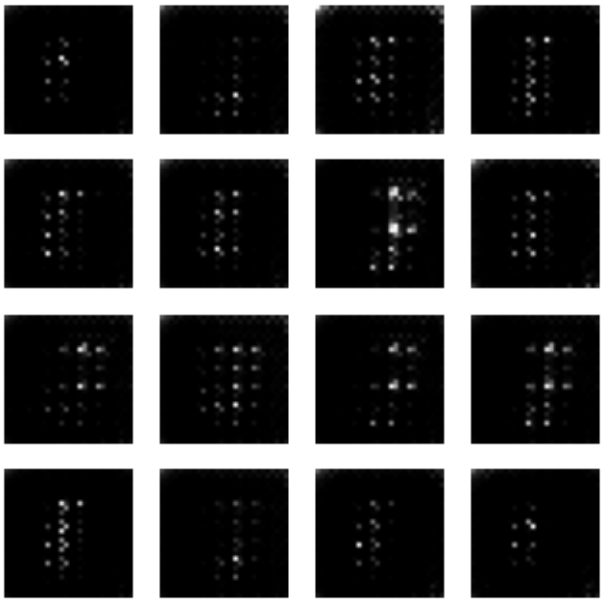
```
    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same',␣
 ↪use_bias=False, activation='tanh'))
    return model

# Discriminator model
def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',␣
 ↪input_shape=[28, 28, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))
    return model

# Define the generator and discriminator
generator = make_generator_model()
discriminator = make_discriminator_model()

# Define the loss and optimizers
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

# Training loop
EPOCHS = 50
noise_dim = 100
num_examples_to_generate = 16

# We will reuse this seed overtime (so it's easier) to visualize progress in␣
 ↪the animated GIF)
seed = tf.random.normal([num_examples_to_generate, noise_dim])
```

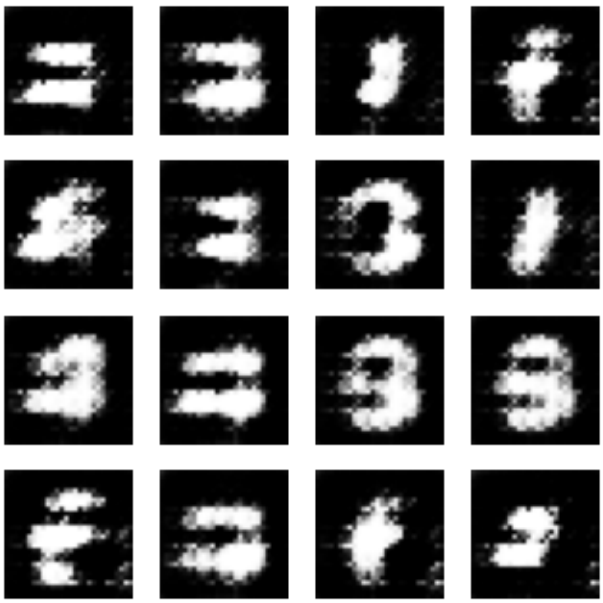
```
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.
 ↪trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.
 ↪trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.
 ↪trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,␣
 ↪discriminator.trainable_variables))

    return gen_loss, disc_loss

def train(dataset, epochs):
    gen_losses = []
    disc_losses = []
    for epoch in range(epochs):
        epoch_gen_loss = 0
        epoch_disc_loss = 0
        for image_batch in dataset:
            gen_loss, disc_loss = train_step(image_batch)
            epoch_gen_loss += gen_loss
            epoch_disc_loss += disc_loss

        # Calculate average loss for the epoch
        num_batches = len(dataset)
        avg_gen_loss = epoch_gen_loss / num_batches
        avg_disc_loss = epoch_disc_loss / num_batches

        gen_losses.append(avg_gen_loss)
        disc_losses.append(avg_disc_loss)

        # Produce images for the GIF as we go
        generate_and_save_images(generator, epoch + 1, seed)
```

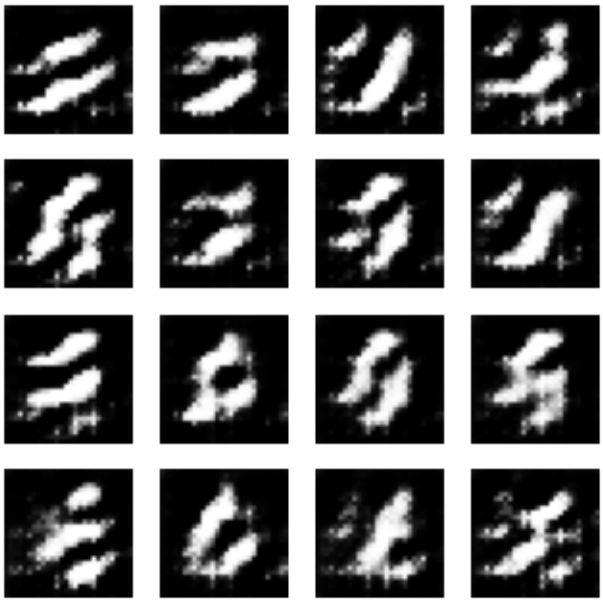
```
    # Generate after the final epoch
    generate_and_save_images(generator, epochs, seed)

    # Plot the loss curves
    plt.figure(figsize=(10, 5))
    plt.plot(range(epochs), gen_losses, label='Generator Loss')
    plt.plot(range(epochs), disc_losses, label='Discriminator Loss')
    plt.title('Loss Curves')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

def generate_and_save_images(model, epoch, test_input):
    # Notice `training` is set to False.
    # This is so all layers run in inference mode (batchnorm).
    predictions = model(test_input, training=False)

    fig = plt.figure(figsize=(4, 4))

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i + 1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
        plt.axis('off')

    plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
    plt.show()

# Train the model
train(train_dataset, EPOCHS)
```

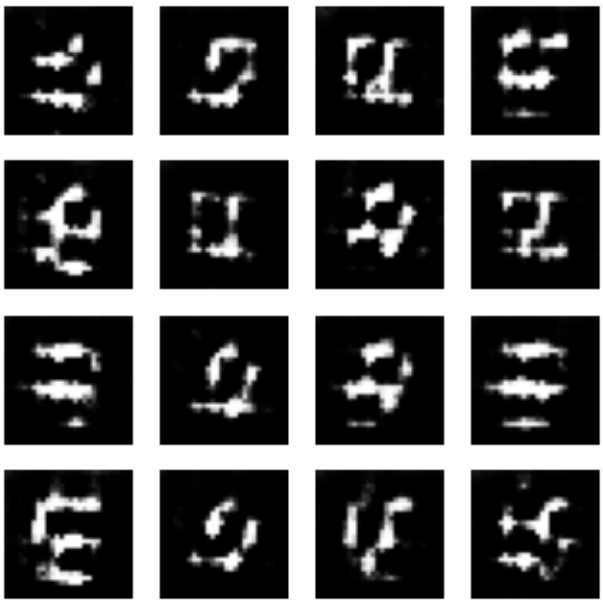
```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```

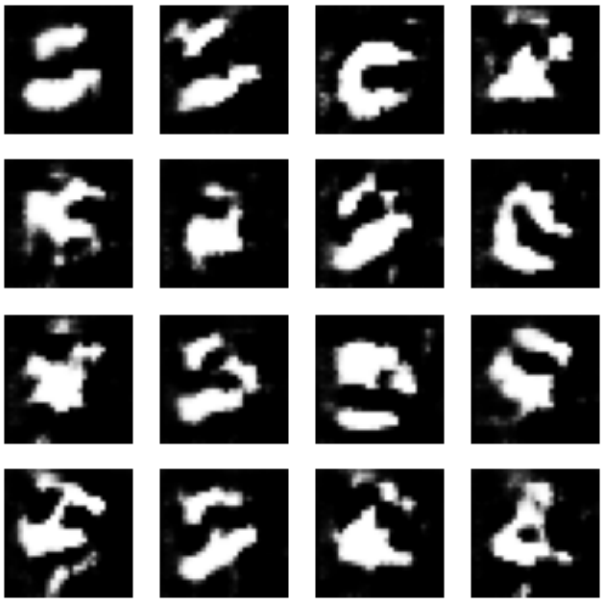


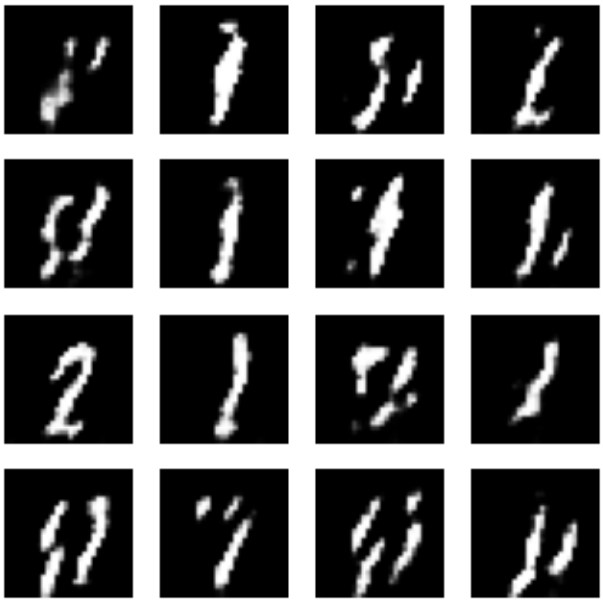


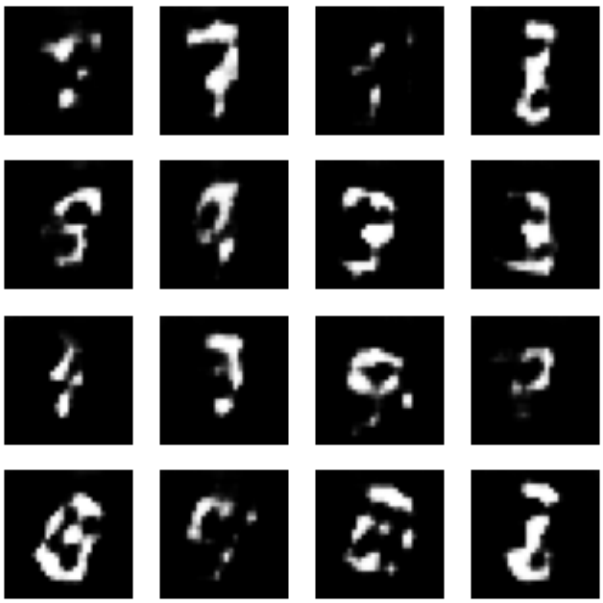


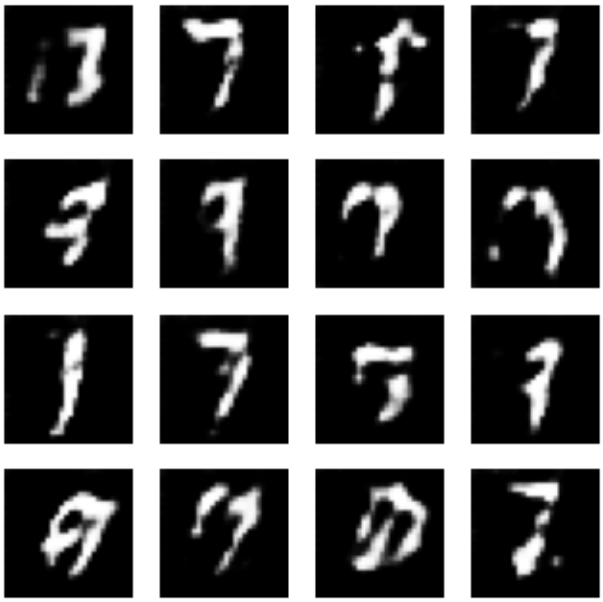


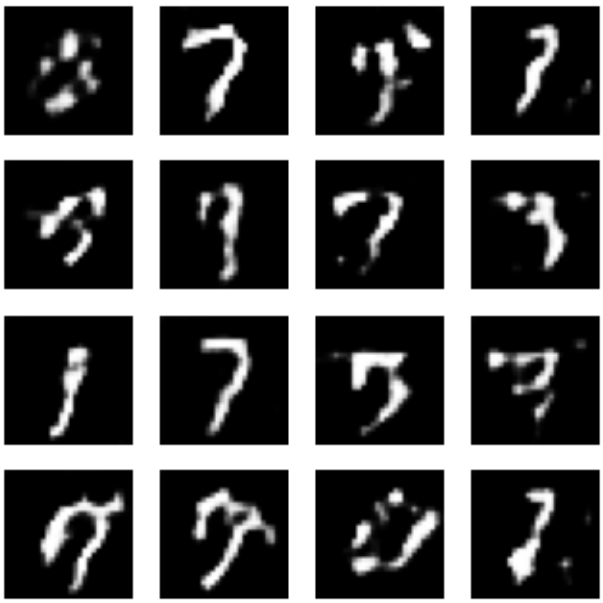


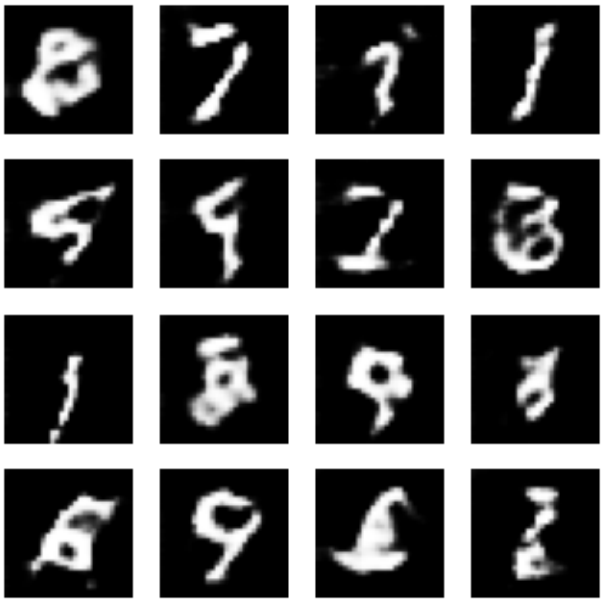


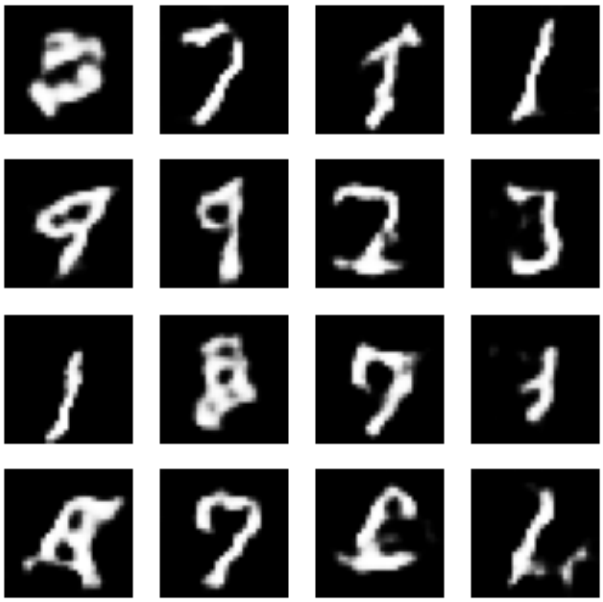


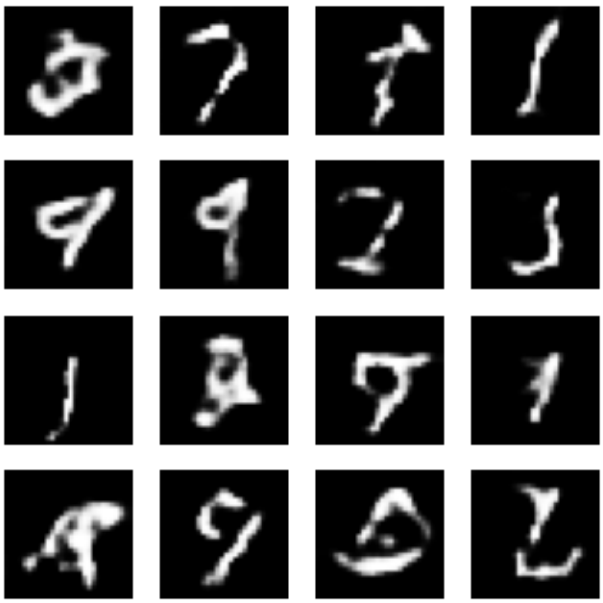


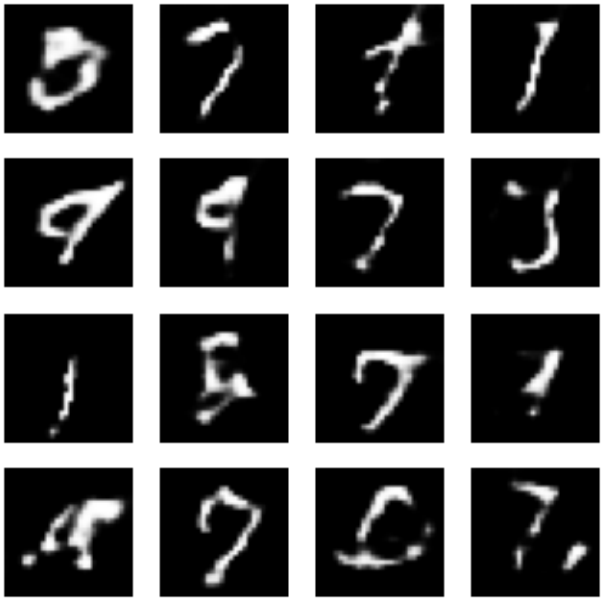


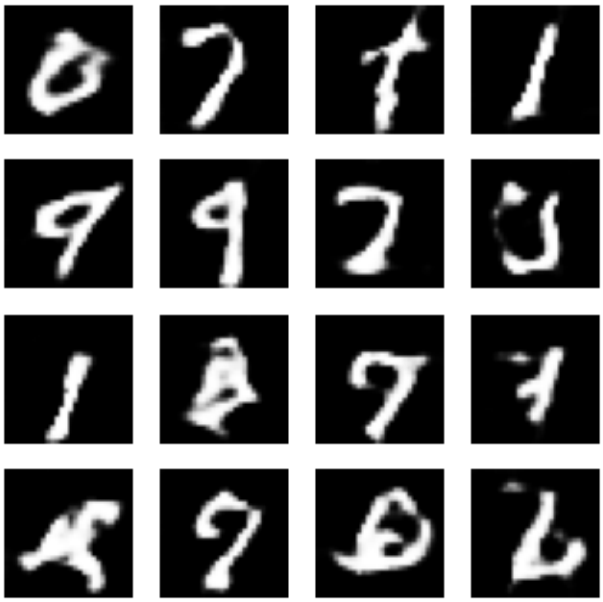


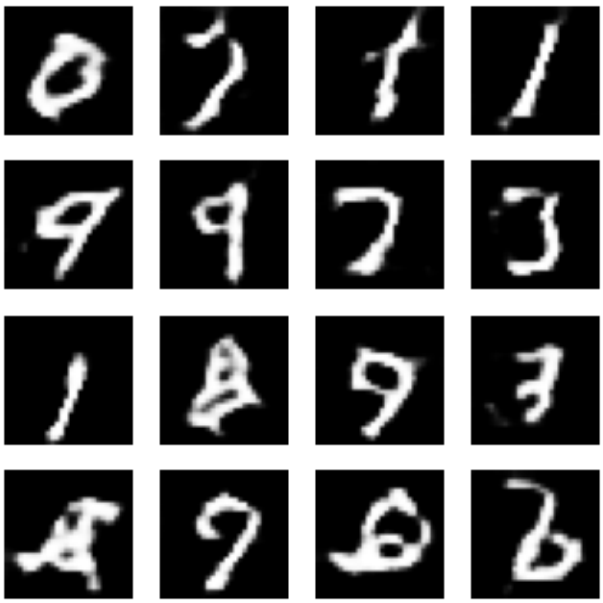


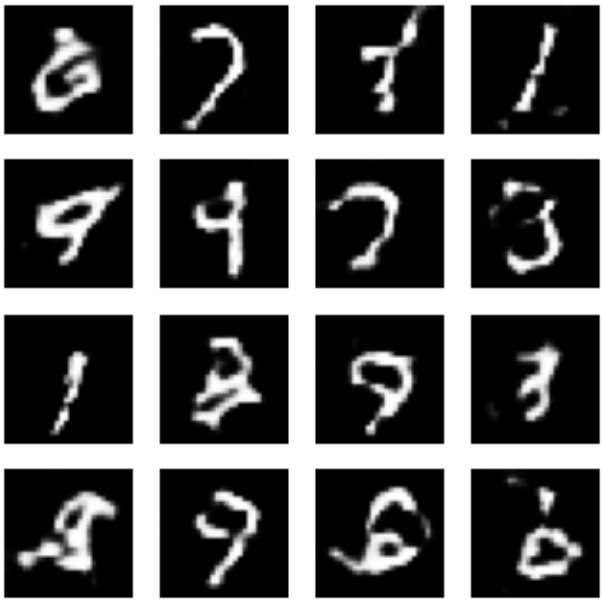


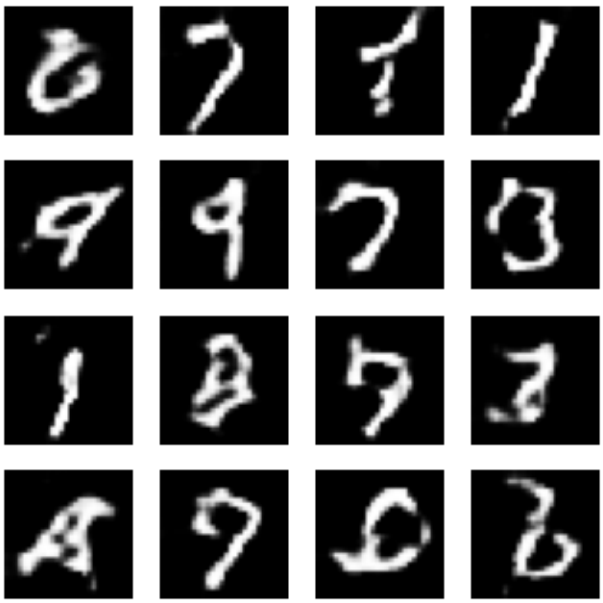


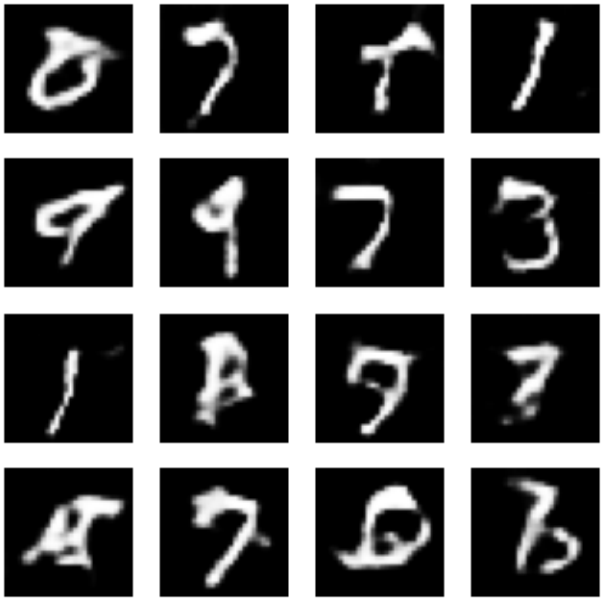


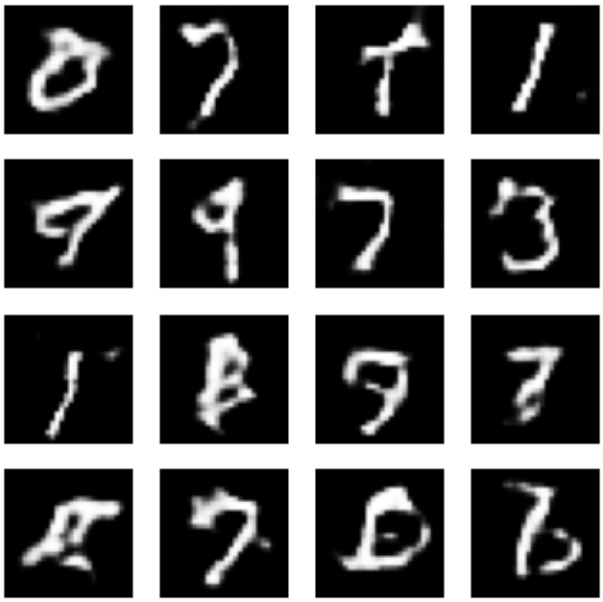


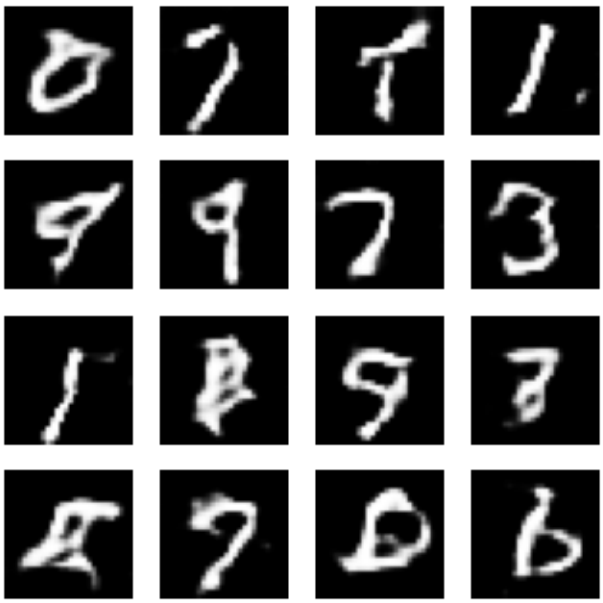


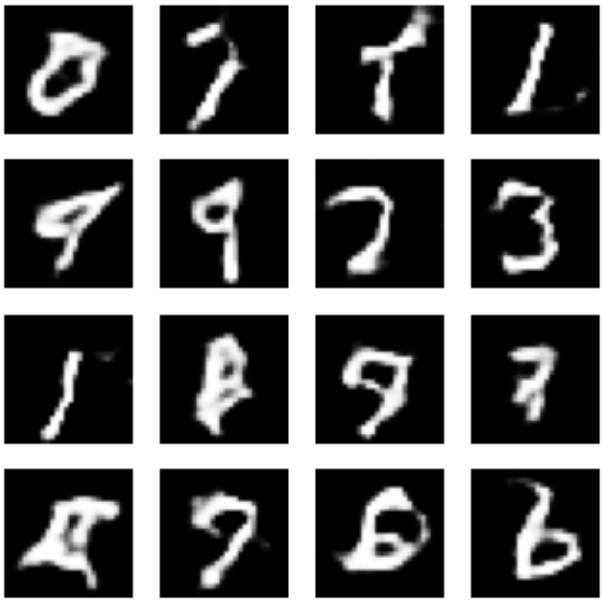


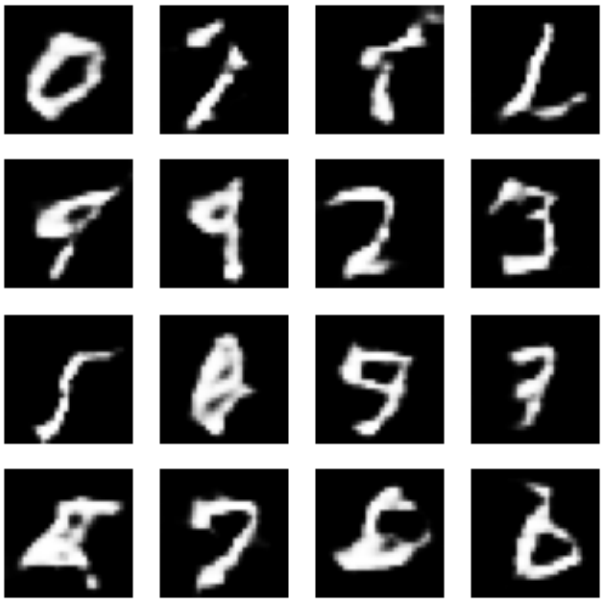


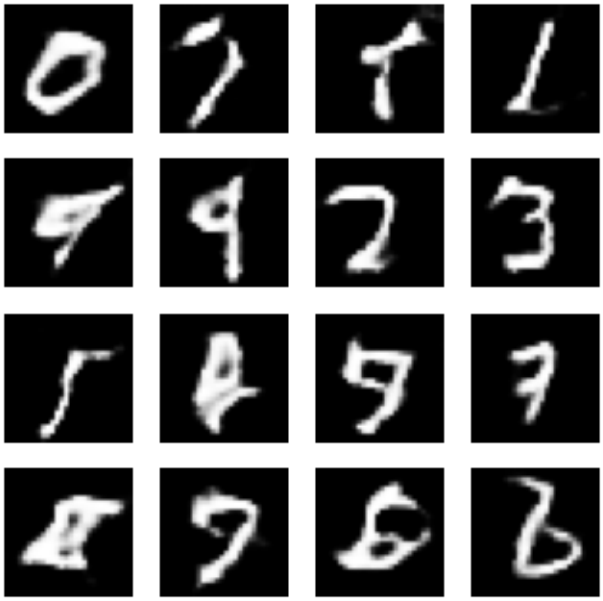


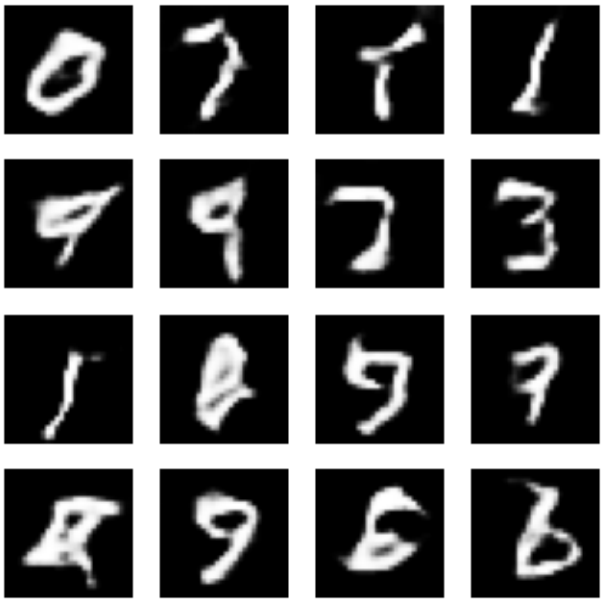


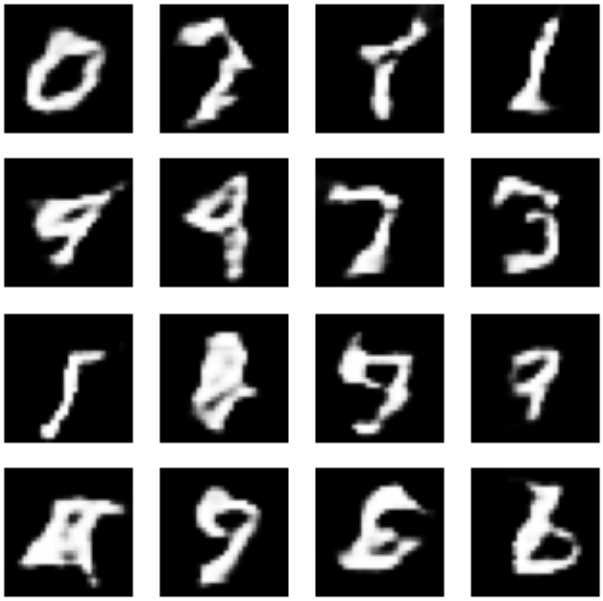


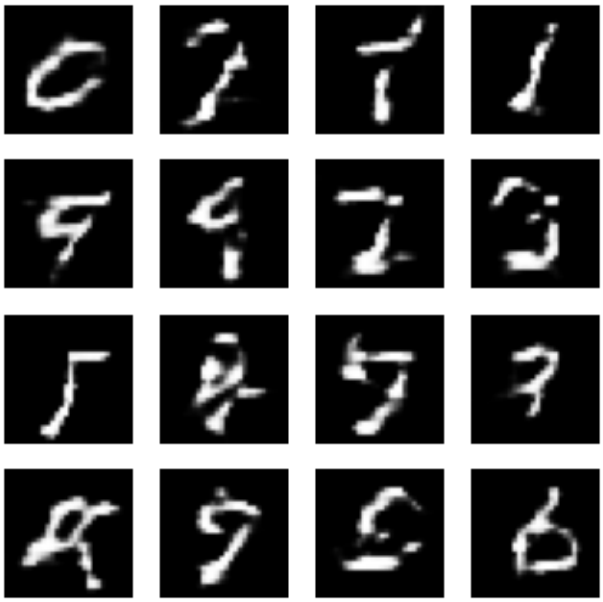


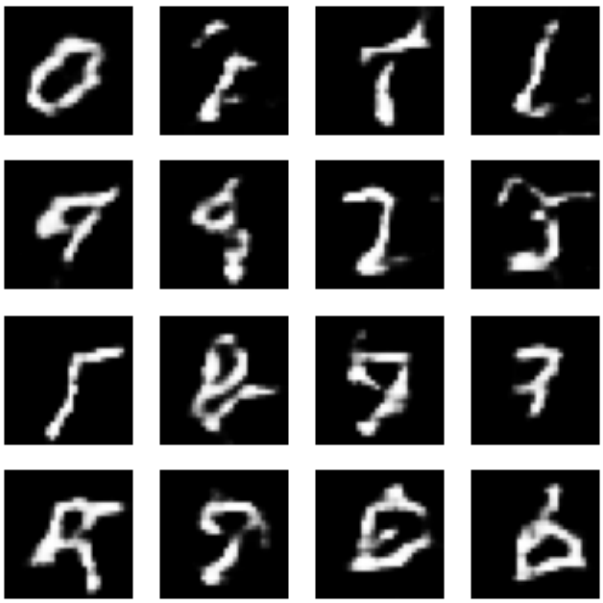


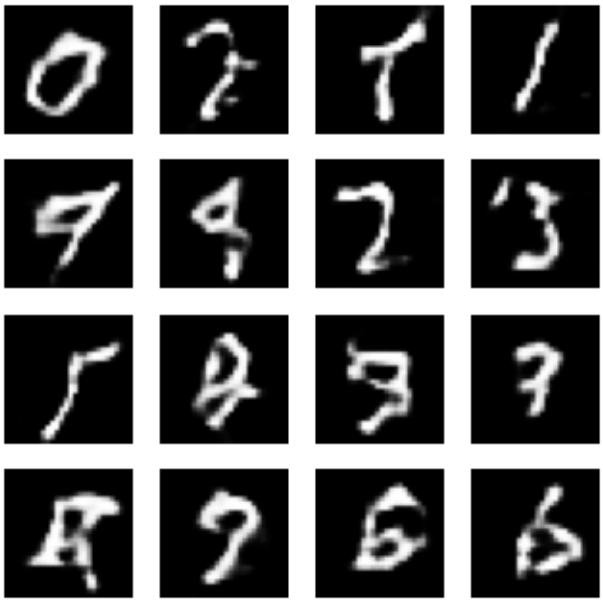


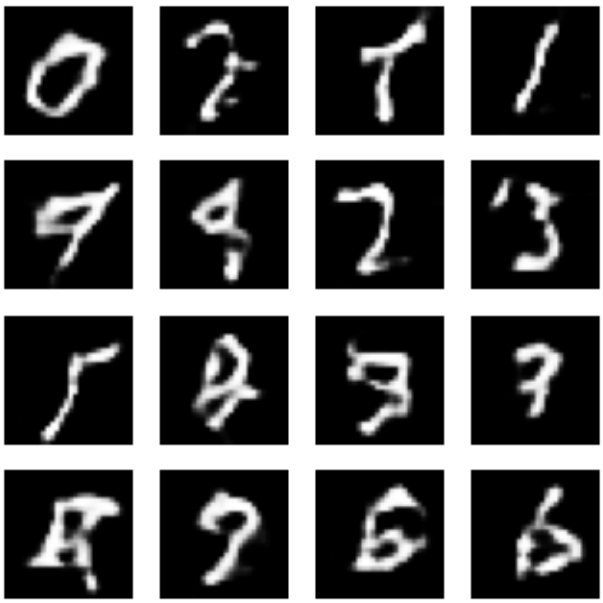


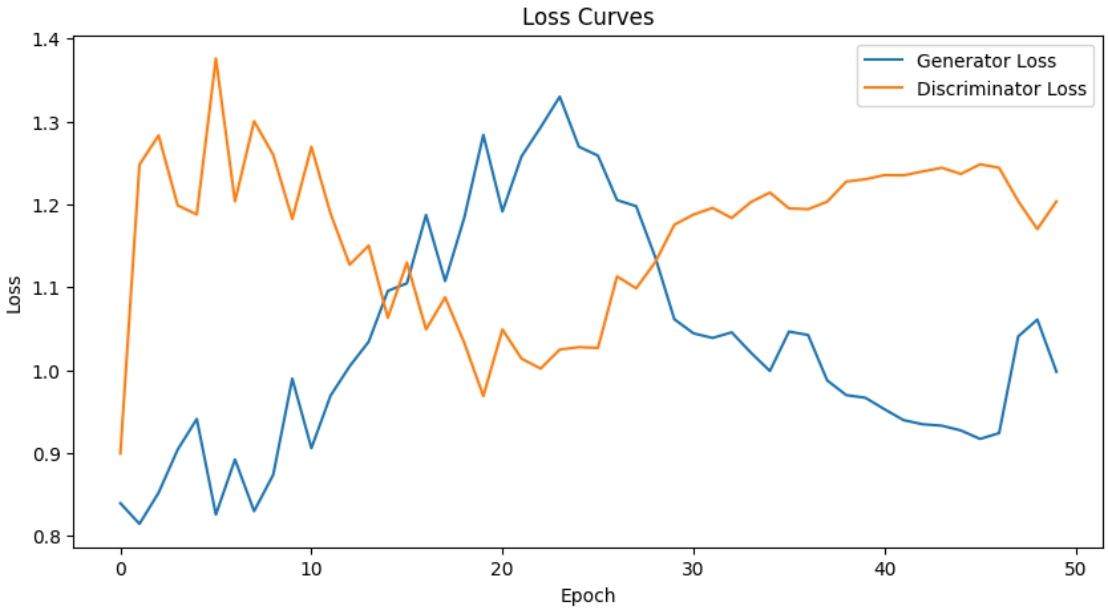











Loss Curves
Generator Loss
Discriminator Loss
Loss
Epoch
1.4
1.3
1.2
1.1
1.0
0.9
0.8
0
10
20
30
40
50

2 Part 2

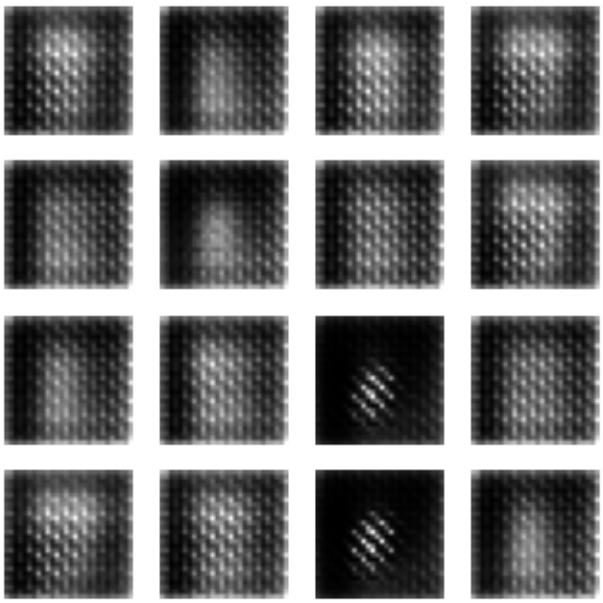
2.1 Adding layers to the model

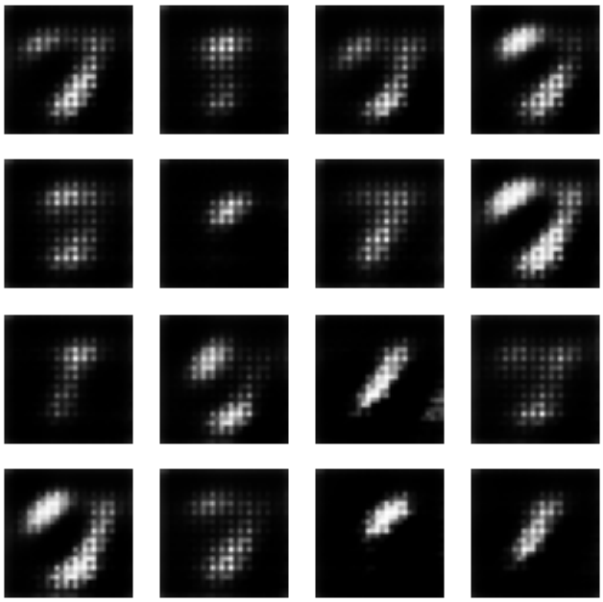
[ ]:
```
import tensorflow as tf
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import numpy as np

# Load and preprocess the MNIST dataset
(train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).
 ↪astype('float32')
train_images = (train_images - 127.5) / 127.5  # Normalize the images to [-1, 1]
BUFFER_SIZE = 60000
BATCH_SIZE = 256
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).
 ↪shuffle(BUFFER_SIZE).batch(BATCH_SIZE)

# Enhanced Generator model
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    model.add(layers.Conv2DTranspose(256, (5, 5), strides=(1, 1),␣
 ↪padding='same', use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(2, 2),␣
 ↪padding='same', use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2),␣
 ↪padding='same', use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(32, (5, 5), strides=(1, 1),␣
 ↪padding='same', use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
```

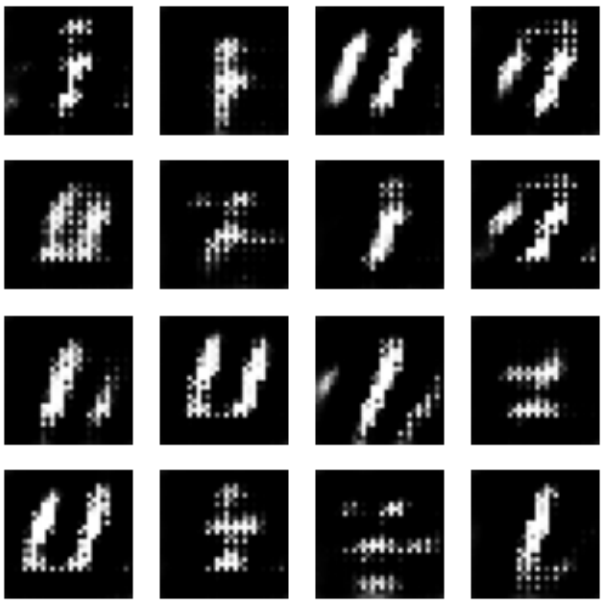
```
    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(1, 1), padding='same',␣
 ↪use_bias=False, activation='tanh'))
    return model

# Enhanced Discriminator model
def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',␣
 ↪input_shape=[28, 28, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(256, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))
    return model

# Define the generator and discriminator
generator = make_generator_model()
discriminator = make_discriminator_model()

# Define the loss and optimizers
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

# Training loop
EPOCHS = 50
noise_dim = 100
num_examples_to_generate = 16
```

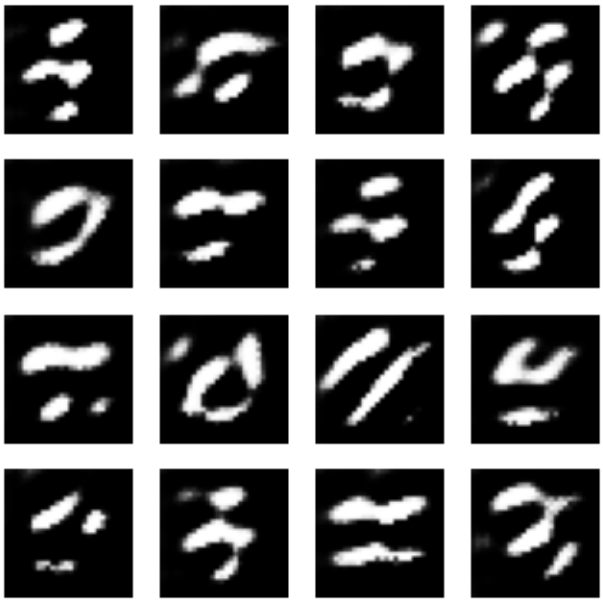
```
# We will reuse this seed overtime (so it's easier) to visualize progress in␣
 ↪the animated GIF)
seed = tf.random.normal([num_examples_to_generate, noise_dim])

@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.
 ↪trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.
 ↪trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.
 ↪trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,␣
 ↪discriminator.trainable_variables))

    return gen_loss, disc_loss

def train(dataset, epochs):
    gen_losses = []
    disc_losses = []
    for epoch in range(epochs):
        epoch_gen_loss = 0
        epoch_disc_loss = 0
        for image_batch in dataset:
            gen_loss, disc_loss = train_step(image_batch)
            epoch_gen_loss += gen_loss
            epoch_disc_loss += disc_loss

        # Calculate average loss for the epoch
        num_batches = len(dataset)
        avg_gen_loss = epoch_gen_loss / num_batches
        avg_disc_loss = epoch_disc_loss / num_batches

        gen_losses.append(avg_gen_loss)
```

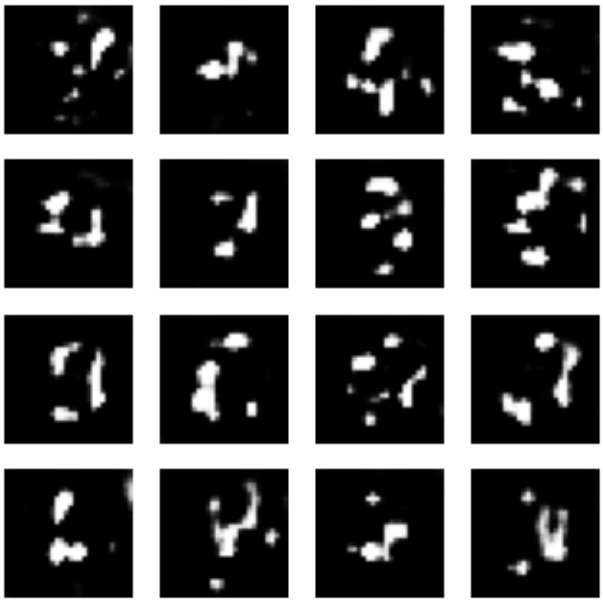
```
        disc_losses.append(avg_disc_loss)

        # Produce images for the GIF as we go
        generate_and_save_images(generator, epoch + 1, seed)

    # Generate after the final epoch
    generate_and_save_images(generator, epochs, seed)

    # Plot the loss curves
    plt.figure(figsize=(10, 5))
    plt.plot(range(epochs), gen_losses, label='Generator Loss')
    plt.plot(range(epochs), disc_losses, label='Discriminator Loss')
    plt.title('Loss Curves')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

def generate_and_save_images(model, epoch, test_input):
    # Notice `training` is set to False.
    # This is so all layers run in inference mode (batchnorm).
    predictions = model(test_input, training=False)

    fig = plt.figure(figsize=(4, 4))

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i + 1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
        plt.axis('off')

    plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
    plt.show()

# Train the model
train(train_dataset, EPOCHS)
```

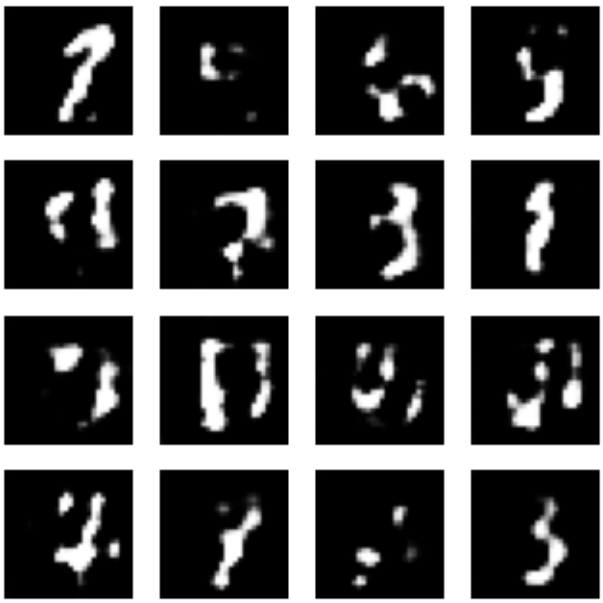


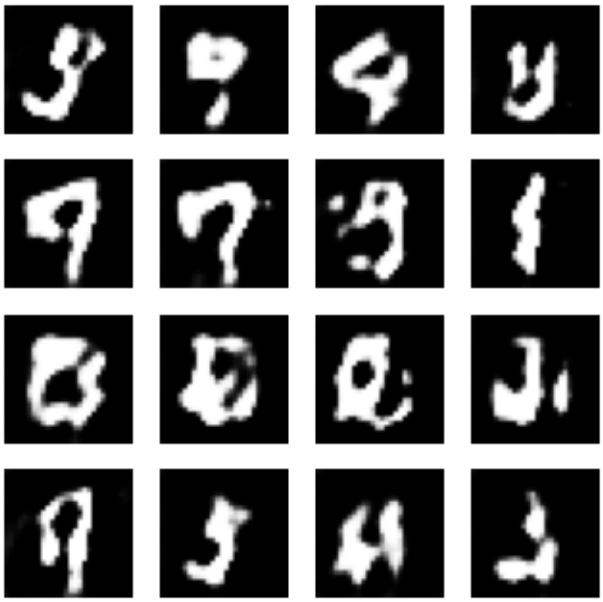


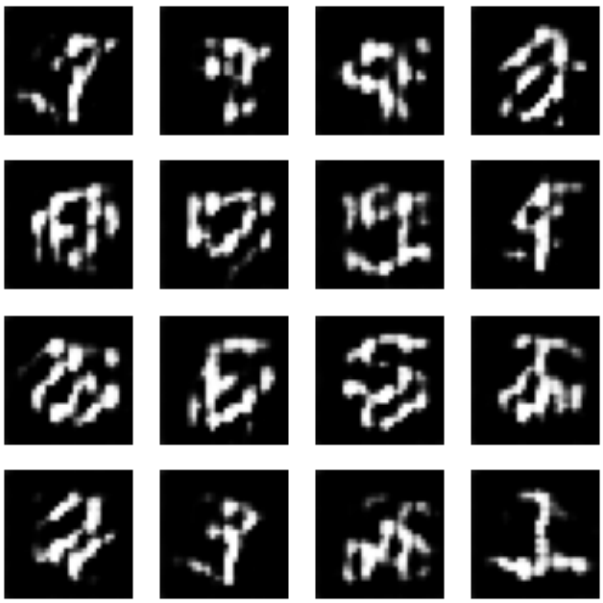


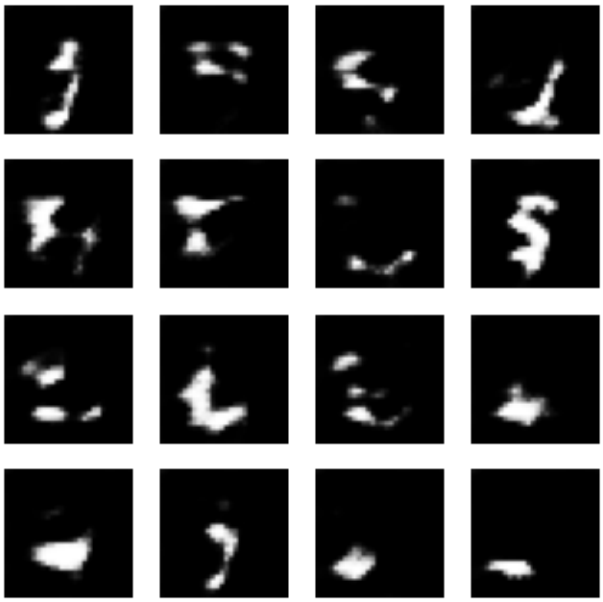


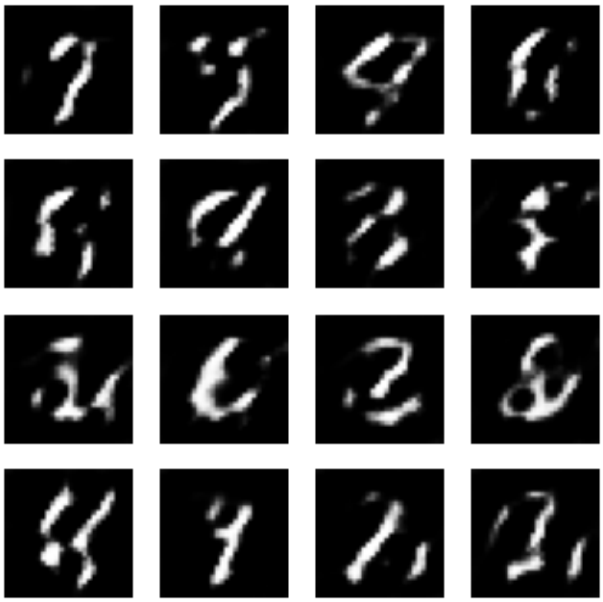


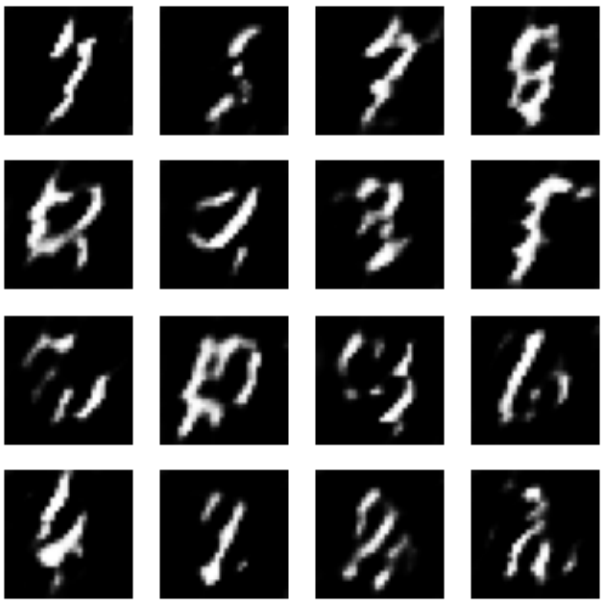


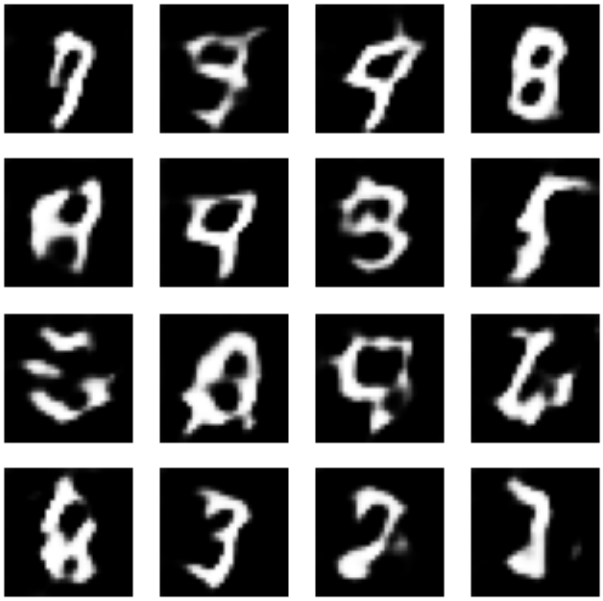


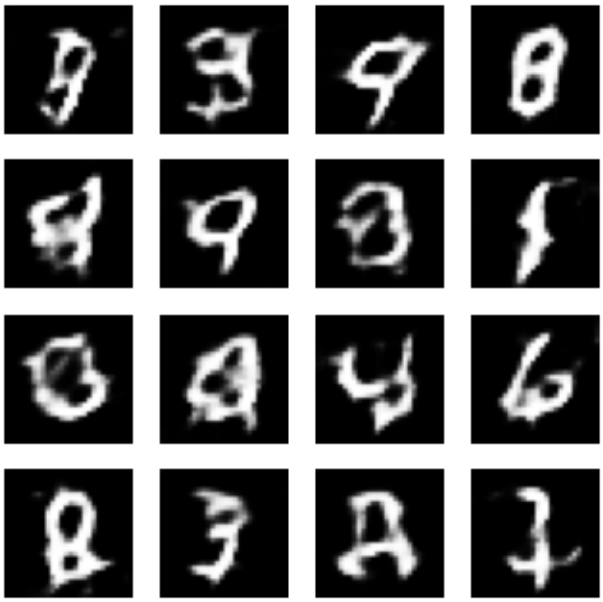


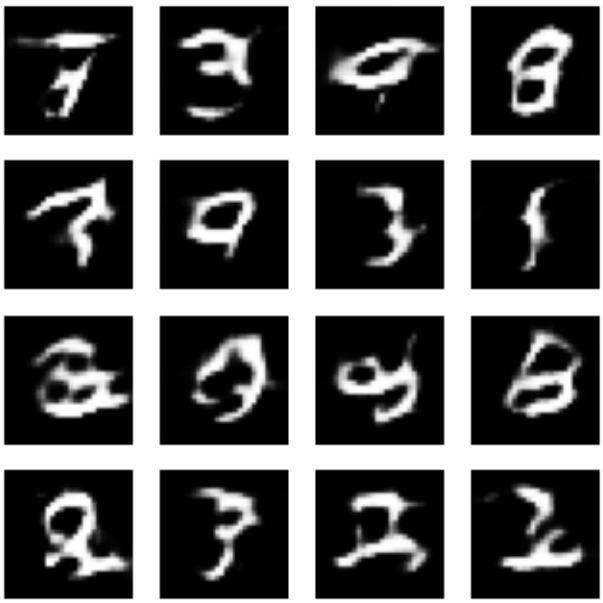


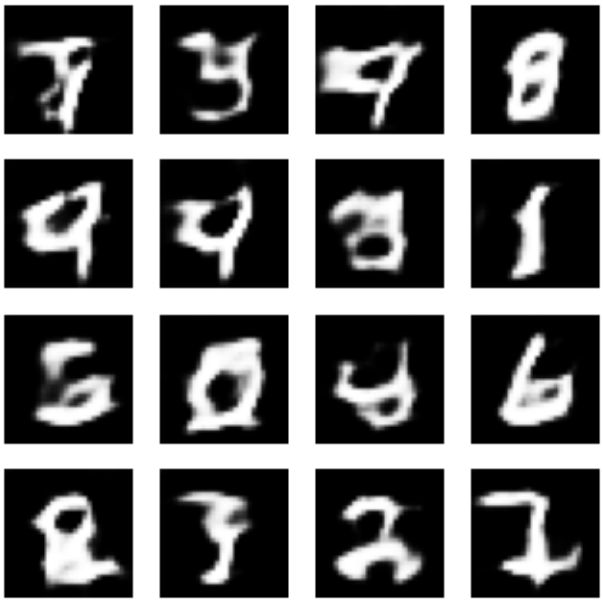


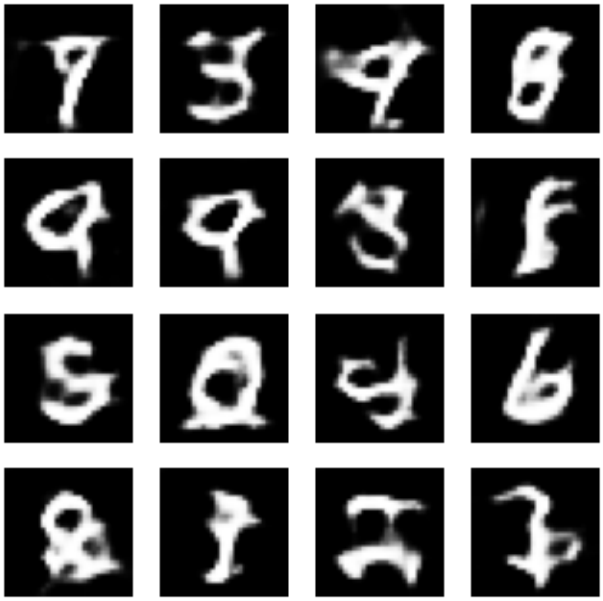


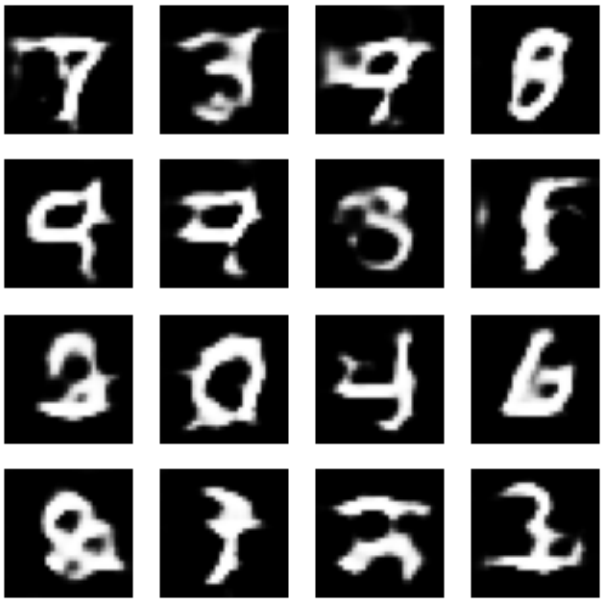


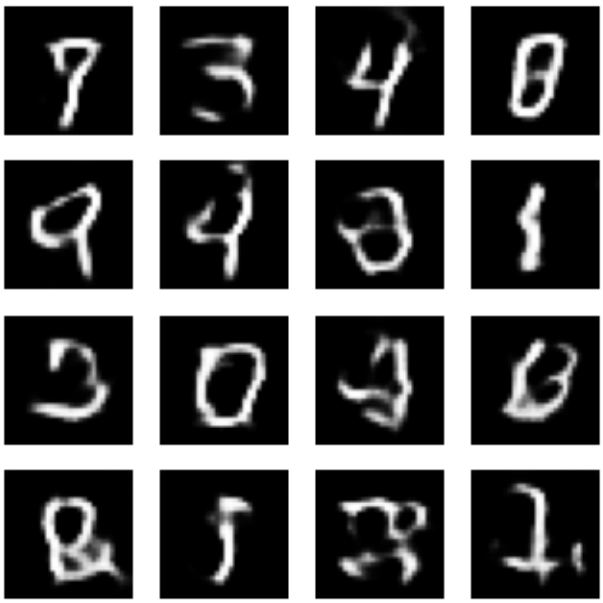


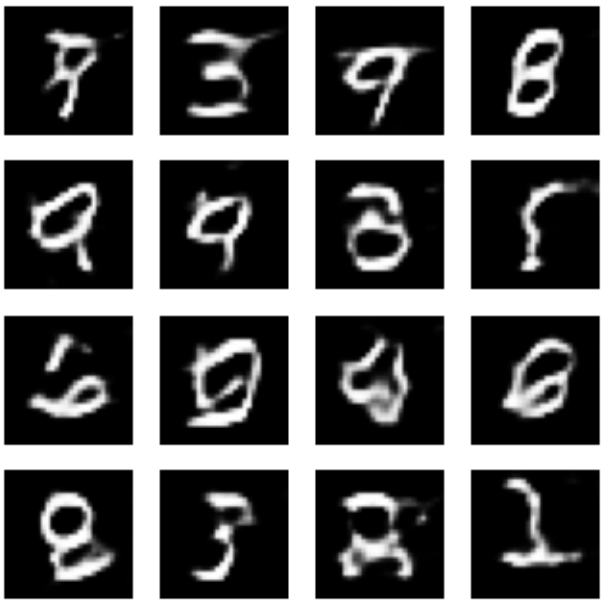


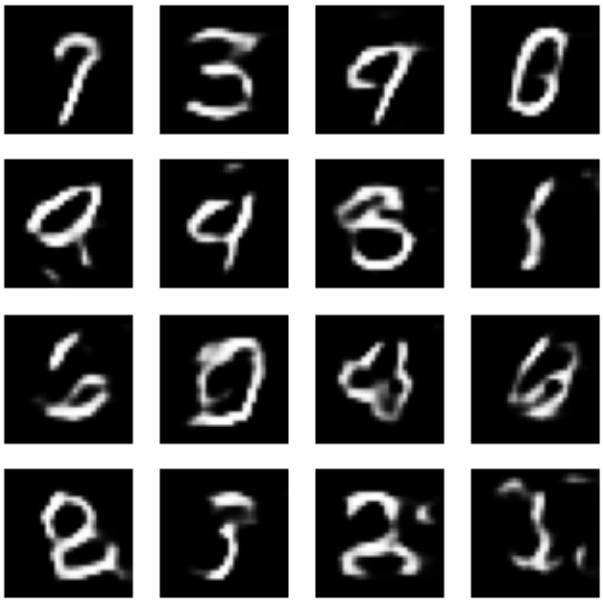


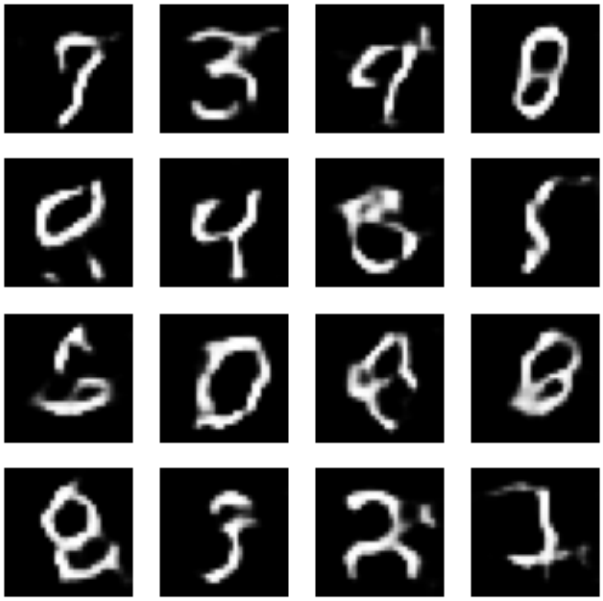


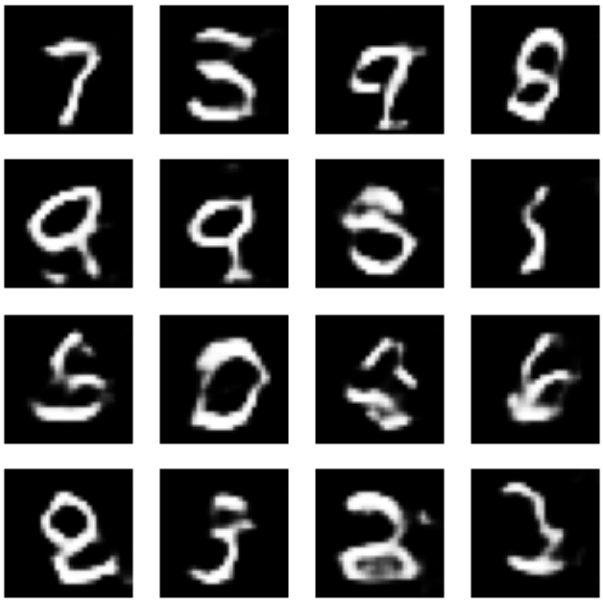


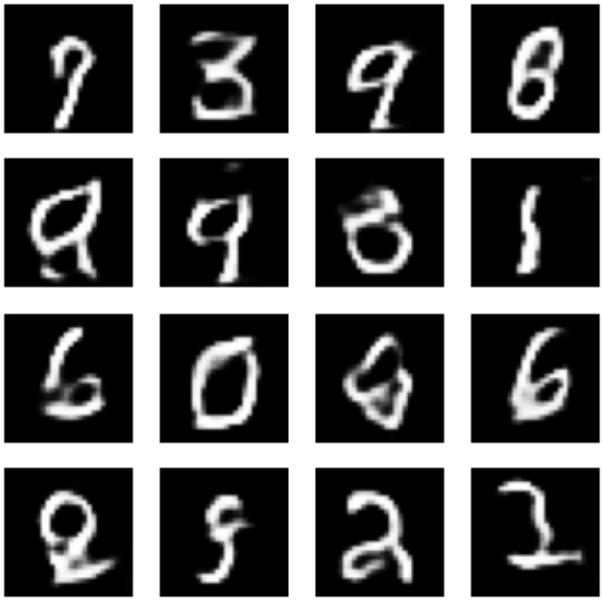


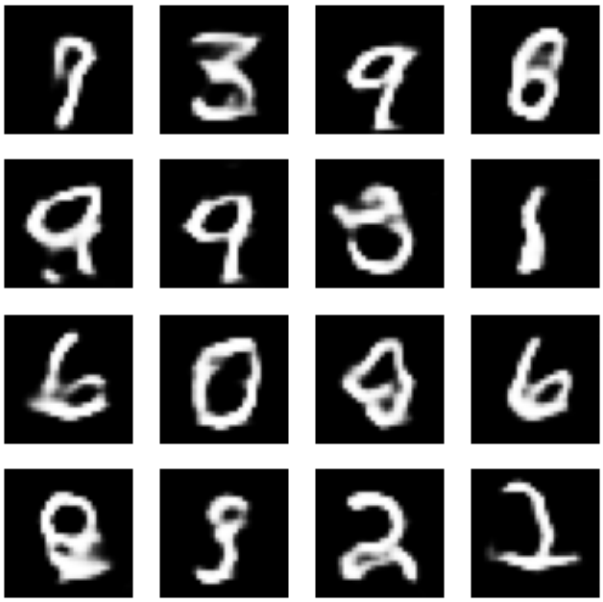


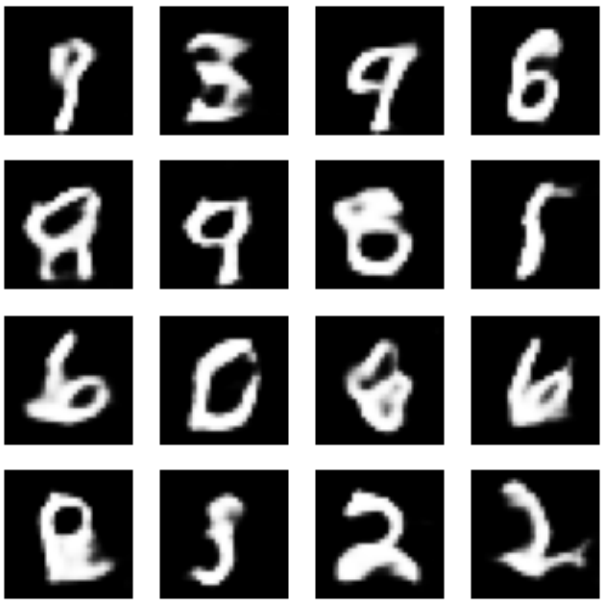


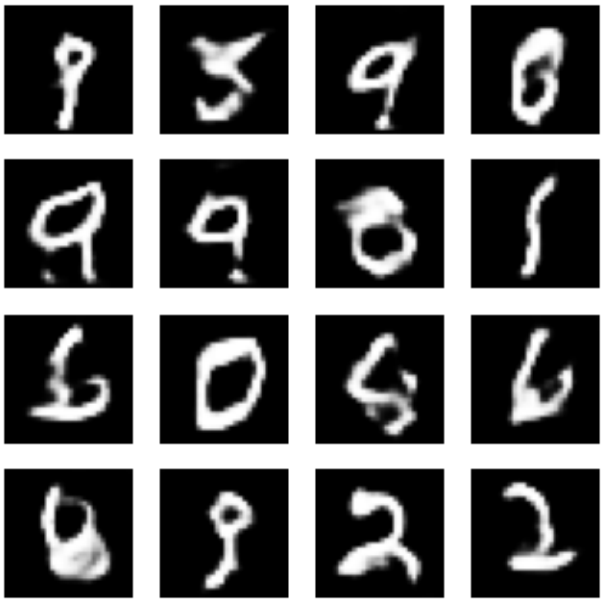


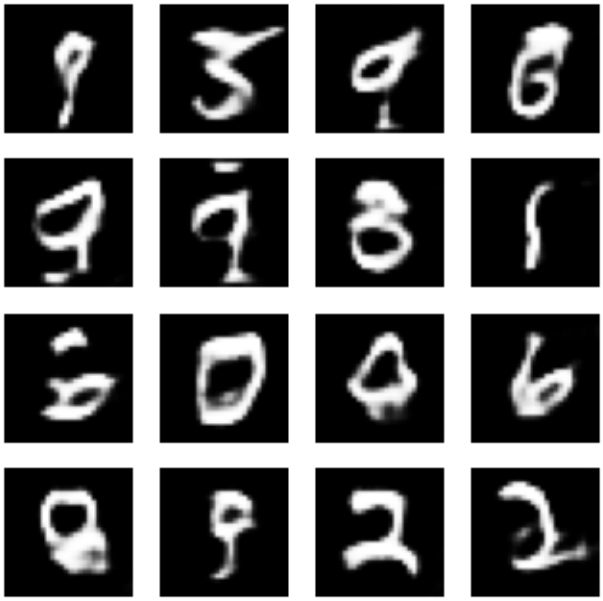


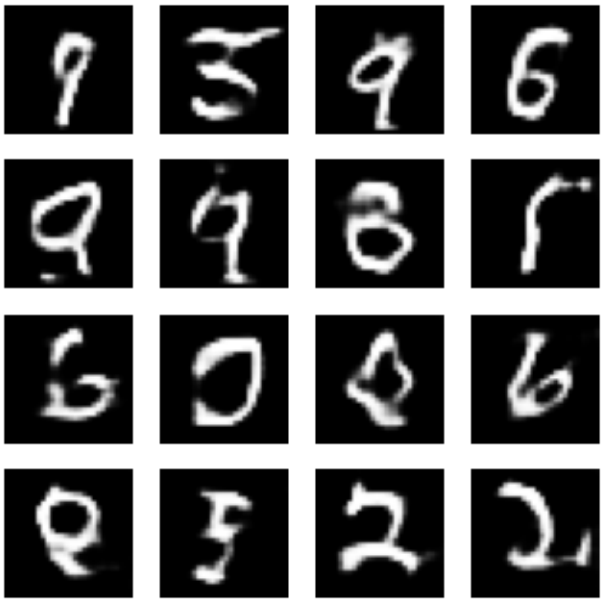


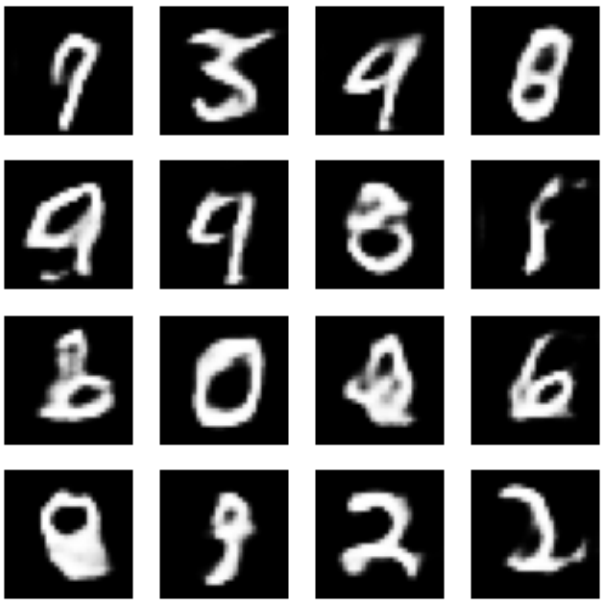


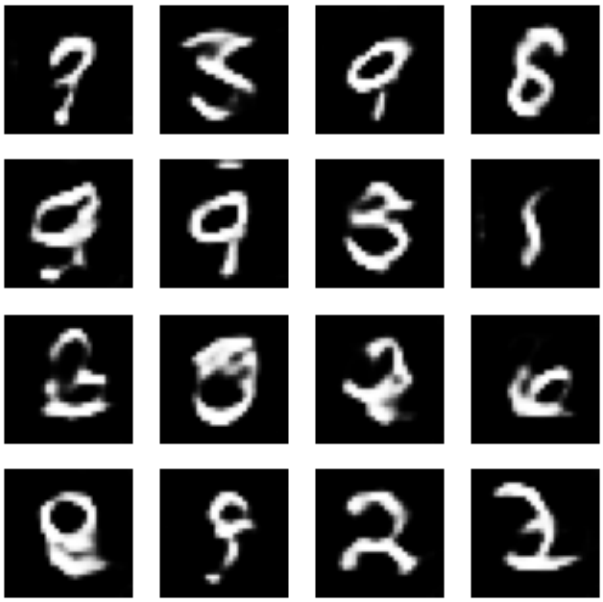


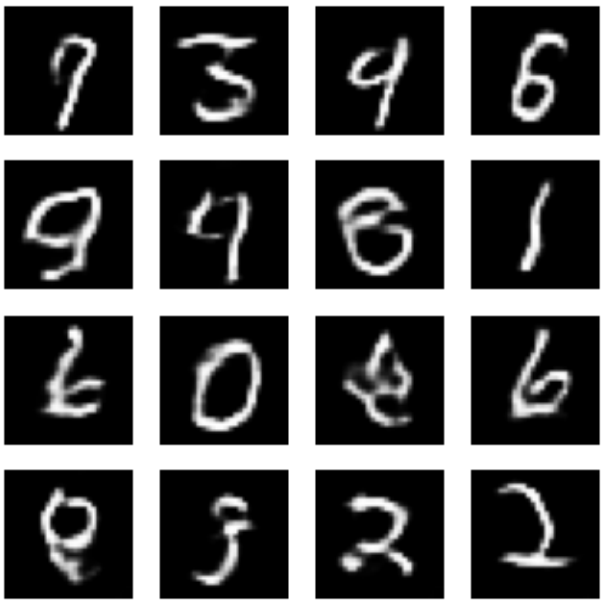


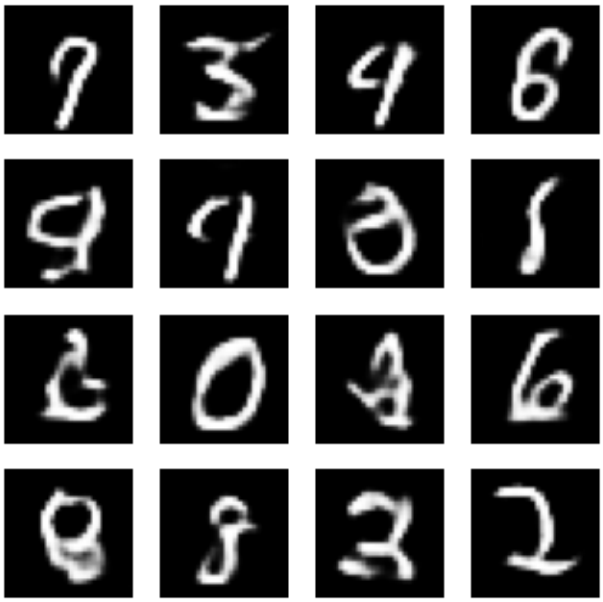


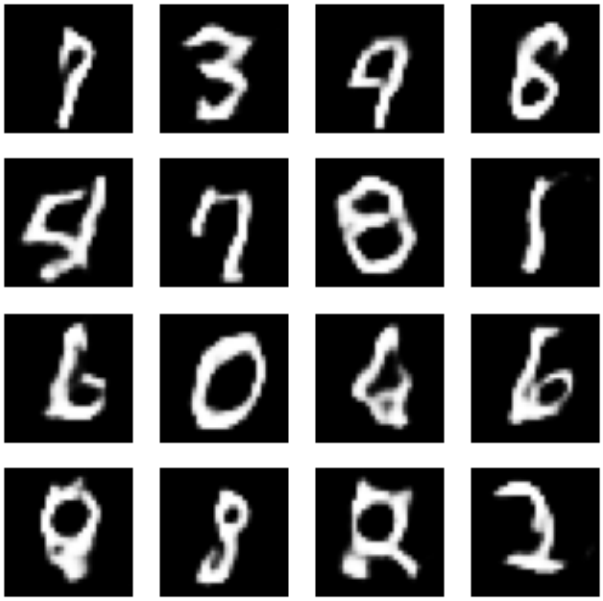


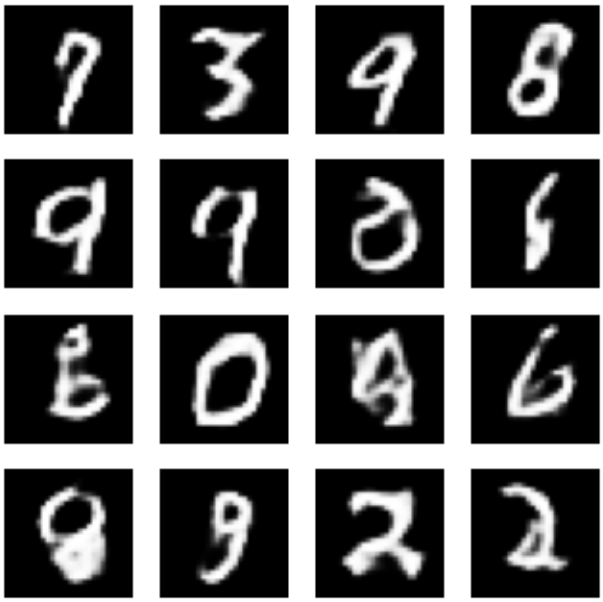


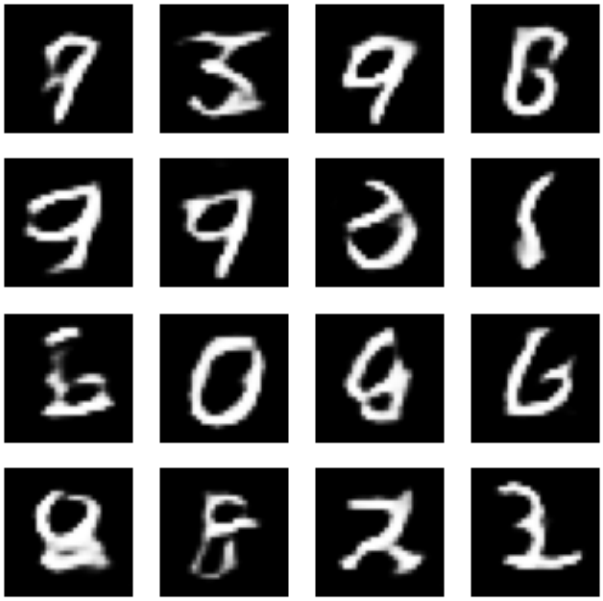


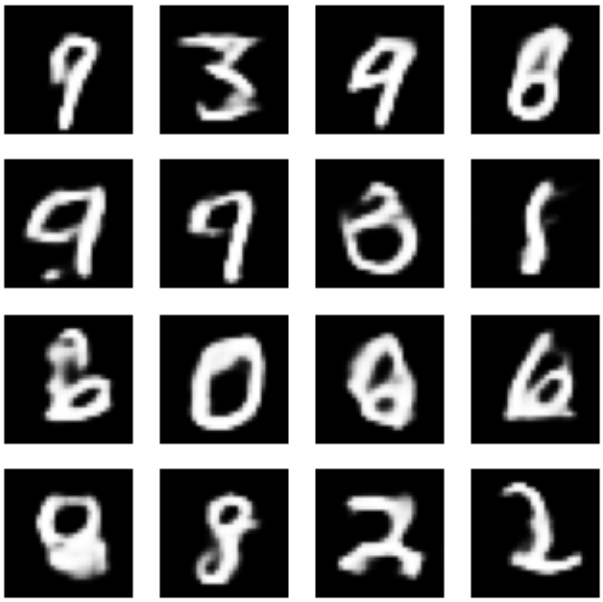


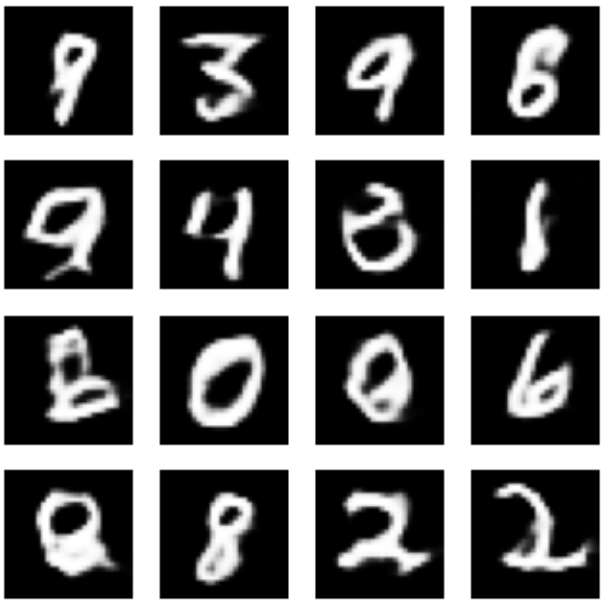


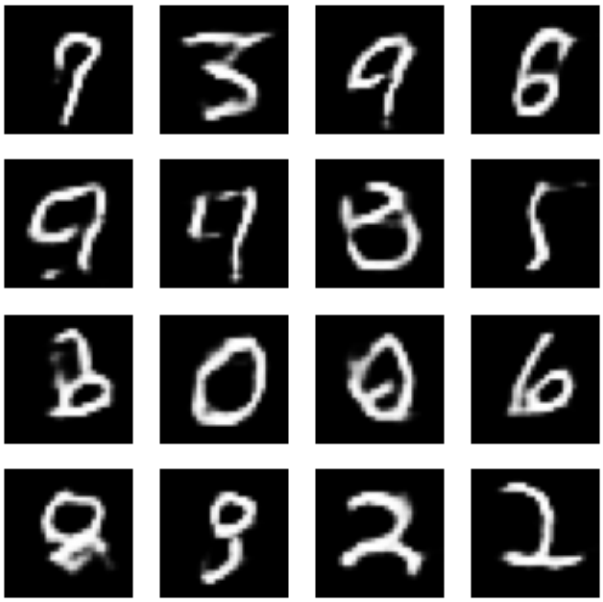


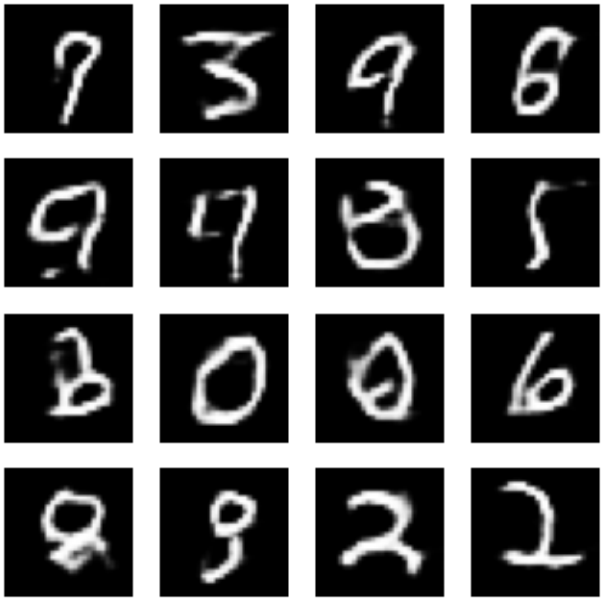


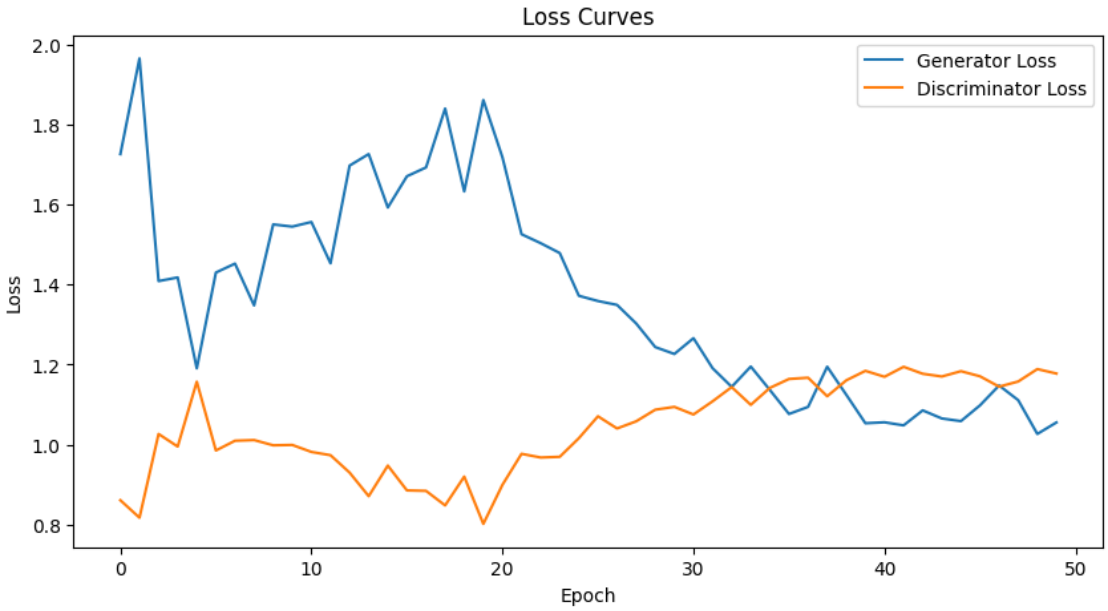








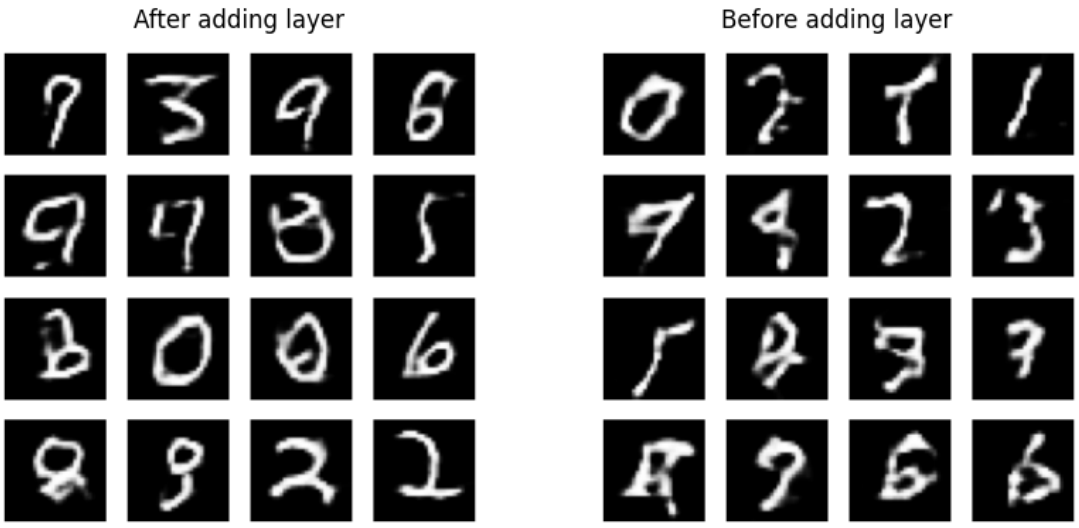



Loss Curves
Generator Loss
Discriminator Loss
Loss
Epoch
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0
10
20
30
40
50

3 Part 3

3.1 Comparison

[ ]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# Load images
image1 = mpimg.imread('with_enhancement.png')
image2 = mpimg.imread('without_enhancement.png')

# Create a figure and set of subplots
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

# Display the first image
axes[0].imshow(image1)
axes[0].axis('off')  # Hide the axes
axes[0].set_title('After adding layer')

# Display the second image
axes[1].imshow(image2)
axes[1].axis('off')  # Hide the axes
axes[1].set_title('Before adding layer')

# Show the plot
plt.show()
```



3.1.1 Comparison of Non-Enhanced and Enhanced Models

Note: We have compared generated images at epoch 50th for both models #### Similarities:

1. **Basic Structure**: Both models are GANs with a similar fundamental architecture comprising a generator and a discriminator.
2. **Training Procedure**: The training procedure, including loss functions and optimizers, is the same for both models.
3. **Output Format**: Both models generate grayscale images of size 28x28 pixels, which are normalized to the range [-1, 1].

Differences:

1. **Model Complexity**:
   - **Non-Enhanced Model**: The generator and discriminator have fewer layers and parameters.
   - **Enhanced Model**: The generator and discriminator have additional layers, resulting in more parameters and increased complexity.
2. **Loss Curves**:
   - **Non-Enhanced Model**: The generator and discriminator loss curves show a higher and more fluctuating loss, indicating less stable training and higher error.
   - **Enhanced Model**: The loss curves are smoother and lower, indicating more stable training and reduced error.
3. **Image Quality**:
   - **Non-Enhanced Model**: The generated images are more likely to be blurry and contain more artifacts, with some digits not being clearly recognizable.
   - **Enhanced Model**: The generated images are sharper, with clearer and more recognizable digits, showing finer details and fewer artifacts.
4. **Training Speed**:
   - **Non-Enhanced Model**: Training is faster due to fewer parameters and simpler architecture.
   - **Enhanced Model**: Training is slower due to the increased complexity, but the improved results justify the extra computational effort.
5. **Diversity and Consistency**:
   - **Non-Enhanced Model**: The diversity of generated digits is less, and some generated images might look similar or repetitive.
   - **Enhanced Model**: There is greater diversity in the generated digits, and the images are more consistent with the variations found in real MNIST digits.

3.1.2 Conclusion

The enhanced model, with additional layers and increased complexity, results in significantly improved performance. This improvement is reflected in the lower loss, reduced error, and higher quality of generated images. The enhanced model produces clearer, more detailed, and more varied images, demonstrating the benefit of added complexity in deep learning models for tasks like image generation.