NEW YORK INSTITUTE OF TECHNOLOGY
Deep Learning (Spring 2024)
Project Assignment 2
Ramis Rawnak, ID: 1319750

Data: The MNIST is a database of handwritten digits (0-9), containing a training set of 60,000 examples (images), and a test set of 10,000 examples..

Classification Task: There is no specific classification task required for this project. However, we can discuss the discriminatory ability to draw classification between fake and real images.

Metrics: The metrics used to evaluate the performance of the DCGAN model were Generator Loss, Discriminator Loss, Image Quality and the time taken to train.

Task: With the given MNIST dataset, implement a deep convolutional Generative Adversarial Network (DCGAN). Use TensorFlow + Keras to implement the DCGAN to generate synthetic handwritten digits data using the dataset.

PART A: Implement the Baseline model and discuss the impact of the number of epochs on the quality of the generated images (E.g. Compare sample images generated after 10 epochs, 50 epochs, and 100 epochs)

The main objective of this project was to implement a Deep Convolutional Generative Adversarial Network (DCGAN) utilizing TensorFlow and Keras. The DCGAN model was trained on the MNIST dataset to generate a set of handwritten digit images in PNG format. This report discusses the process of generating these images and their quality with the number of epochs assigned during time of training.

```
[ ] (train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()
```

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

```
[ ] 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
```

```
[ ] # Batch and shuffle the data
    train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

At first, the dataset was loaded and with a buffer_size of 6000 and a batch_size of 256, the dataset was trained on tensor flow.

```python
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)))
    assert model.output_shape == (None, 7, 7, 256)  # Note: None is the batch size

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    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'))
    assert model.output_shape == (None, 28, 28, 1)

    return model
```

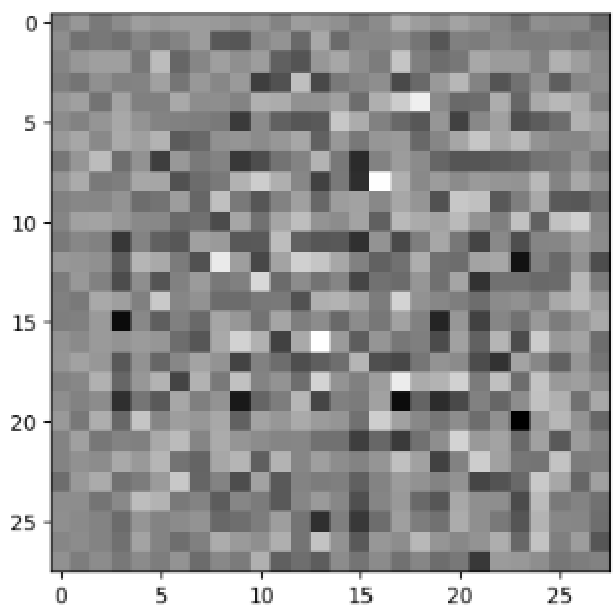
A generator function was created to build the generator model. This function is called to produce realistic images using random noises.

```python
[ ] generator = make_generator_model()

    noise = tf.random.normal([1, 100])
    generated_image = generator(noise, training=False)

    plt.imshow(generated_image[0, :, :, 0], cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x79f563651240>
```



A sample fake image was created with noises assigned to check the functionality of the generator function. As shown above with a distorted image.

```
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
```

```
discriminator = make_discriminator_model()
decision = discriminator(generated_image)
print (decision)
```

```
tf.Tensor([[0.00024603]], shape=(1, 1), dtype=float32)
```

```
# This method returns a helper function to compute cross entropy loss
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)

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

#Mouting the drive to load a simple dataset stored on the google drive
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

```
#Checkpoints

checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer, discriminator_optimizer=discriminator_optimizer, generator=generator, discriminator=discriminator)
```

Both the generator and discriminator functions were trained at the same time using Adam optimizer. The purpose of the generator was to generate fake images that would deceive the discriminator, while the discriminator's purpose is to accurately differentiate between the real and fake images. This opposing process creates an observation loop and as time proceeds, the generator learns to advance its ability to produce realistic images while the discriminator advances in detecting fake and real images.

```
[ ]  #Training Loop Starts Here

     #Define the no. of Epochs for training
     EPOCHS = [10, 50, 100]
     noise_dim = 100
     num_examples_to_generate = 16

     # You 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])

     # Notice the use of `tf.function`
     # This annotation causes the function to be "compiled".
```

```
@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))
```

```
  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')

  os.makedirs(save_dir, exist_ok=True)

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


def train(dataset, epochs_list):
  for num_epochs in epochs_list:
      for epoch in range(num_epochs):
          start = time.time()

          for image_batch in dataset:
              train_step(image_batch)

    # Produce images for the GIF as you go
          display.clear_output(wait=True)
    #generate_and_save_images(generator, epoch + 1, seed)


    # Save the model every 15 epochs
          if (epoch + 1) % 15 == 0:
              checkpoint.save(file_prefix=checkpoint_prefix)
     # Save the model after 10, 50, 100 epochs
          if (epoch + 1) == 10 or epoch + 1 == 50 or epoch + 1 == 100:
               generate_and_save_images(generator, epoch + 1, seed)

          print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

 # Generate after the final epoch
  display.clear_output(wait=True)
  generate_and_save_images(generator,
                           sum(epochs_list),
                           seed)
  #Save for each set of epochs
  checkpoint.save(file_prefix=checkpoint_prefix)

train(train_dataset, EPOCHS)
```

With epochs set at 10, 50 and 100 and a noise set to 100, the model was trained to produce 3 PNG images. These images (shown below) were then compared to see the quality with each changing epoch.

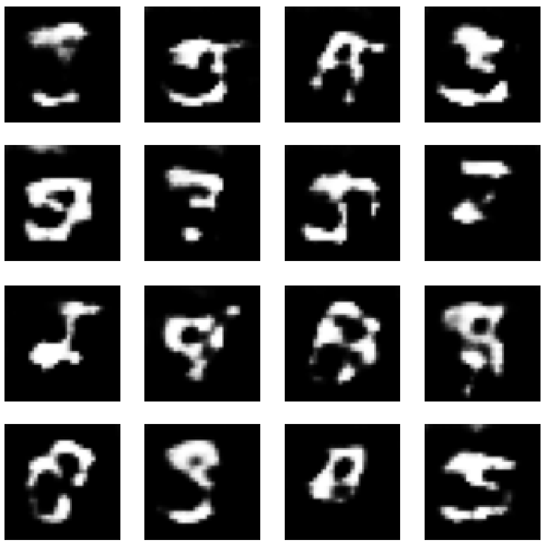


Image at Epoch 10

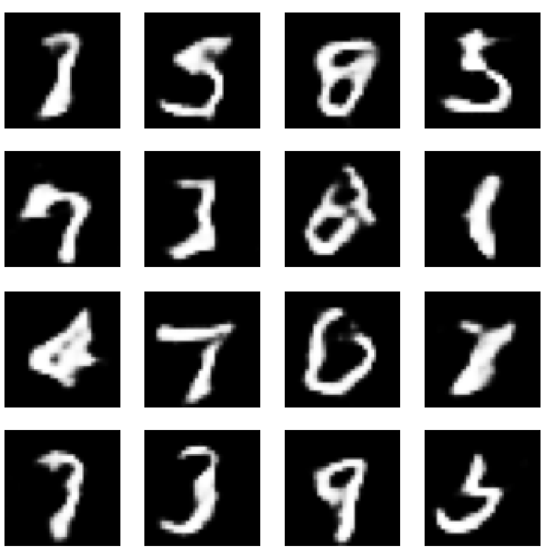


Image at Epoch 50

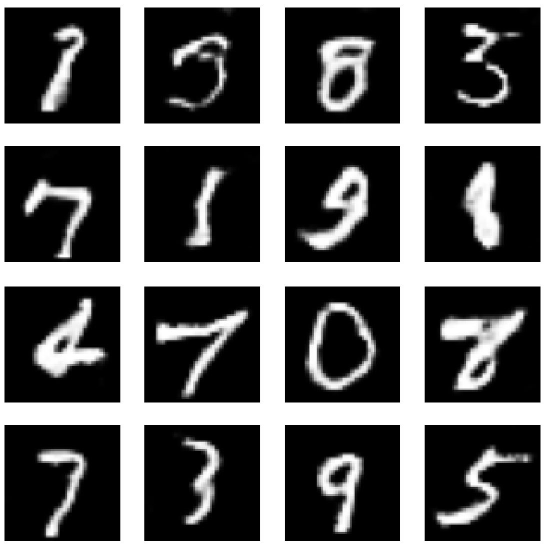


Image at Epoch 100

After generating the images for 10, 50, 100 epochs, it was observed the quality of the images increased with increasing epochs. As you can see the image produced after 10 epochs is unclear and blurry, at epoch 50 it gets slightly better but at epochs 100, the images are better to read and portray sharper details.

We can conclude that the model of DCGAN on the MNIST dataset has successfully generated the handwritten numbers' images. After an analogy of the images produced, we can speculate that with an increased number of epochs, the quality of the generated images improves. For future implementation, exploration of fine-tuning of the model parameters could lead to produce enhanced quality of the images

PART B: Use ReLU activation (instead of LeakyReLU in the Baseline) in the generator for all layers except for the output, which uses a Tanh activation. Report the difference in the quality of images and training time at 50 epochs compared with that of the Baseline.

```python
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.ReLU())

    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256)  # Note: None is the batch size

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.ReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.ReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model
```

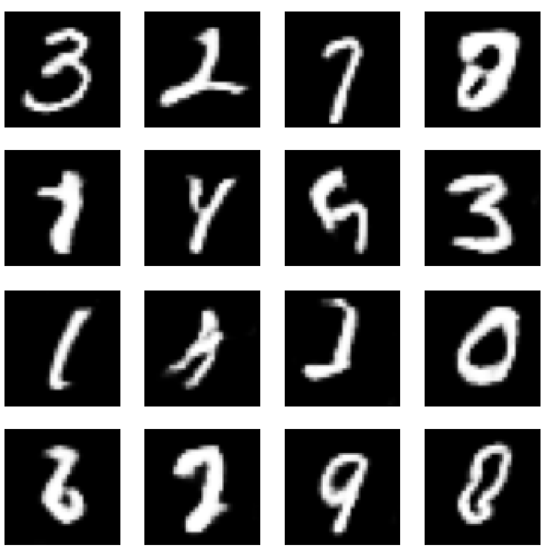


Image at Epoch 50 with ReLU

As per the requirement for Part B, the epochs were set to 50 and ReLU activation was set to ReLU. ReLU takes faster time to produce the image. It only took a few minutes to produce this image whereas the images produced in Part A took at least half an hour to complete. Also if we compare this image with 50 epochs with our image with 50 epochs from part A, we can draw a conclusion that with ReLU not only the image produced is time efficient but the quality also improved. The image produced in part B has sharper and clearer details with less distortion.

Part C: Change the following hyperparameters: (1) the dimensionality of the noise vector, (2) the batch size (i.e., the number of images per forward/backward pass), (3) the learning rates, and (4) the momentum terms. Report the difference in the quality of images and training time at 50 epochs compared with that of the Baseline. Justify your choices of hyperparameter values.

```
[ ] BUFFER_SIZE = 60000
    BASELINE_BATCH_SIZE = 256
    NEW_BATCH_SIZE = 128
    NOISE_DIM = 100
    NEW_NOISE_DIM = 50
    EPOCHS = [50]
    BASELINE_LR = 1e-4
    NEW_LR = 1e-3
```

```
[ ] # Baseline model hyperparameters
    BASELINE_PARAMS = {
        "noise_dim": NOISE_DIM,
        "batch_size": BASELINE_BATCH_SIZE,
        "learning_rate": BASELINE_LR
    }

    # New hyperparameters
    NEW_PARAMS = {
        "noise_dim": NEW_NOISE_DIM,
        "batch_size": NEW_BATCH_SIZE,
        "learning_rate": NEW_LR
    }
```

```python
# Define discriminator and generator
generator_baseline = make_generator_model(BASELINE_PARAMS["noise_dim"])
generator_new = make_generator_model(NEW_PARAMS["noise_dim"])
discriminator = make_discriminator_model()
# Define optimizers
generator_optimizer_baseline = tf.keras.optimizers.Adam(BASELINE_PARAMS["learning_rate"])
generator_optimizer_new = tf.keras.optimizers.Adam(NEW_PARAMS["learning_rate"])
discriminator_optimizer = tf.keras.optimizers.Adam(BASELINE_PARAMS["learning_rate"])  # Sa
```

```python
@tf.function
def train_step(images, generator, discriminator_optimizer, generator_optimizer):
    baseline_noise_dim = BASELINE_PARAMS["noise_dim"]
    new_noise_dim = NEW_PARAMS["noise_dim"]

    #Generate noise tensors using the noise dimensions provided
    baseline_noise= tf.random.normal([BASELINE_BATCH_SIZE, baseline_noise_dim])
    new_noise= tf.random.normal([NEW_BATCH_SIZE, new_noise_dim])

    #Select appropriate noise tensor
    noise = baseline_noise if generator == generator_baseline else new_noise

    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))
```

```python
#Define a checkpoint
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
# checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer_baseline, discriminator_optimizer=discriminator_optimizer, generator=generator_baseline,
#                                  discriminator=discriminator)
checkpoint = tf.train.Checkpoint()
```

```python
def generate_and_save_images(generator, epoch, test_input, save_dir='images/'):
  print(save_dir)
  # Notice `training` is set to False.
  # This is so all layers run in inference mode (batchnorm).
  predictions = generator(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')

  os.makedirs(save_dir, exist_ok=True)

  plt.savefig(os.path.join(save_dir, 'image_at_epoch_{:04d}.png'.format(epoch)))  # Save images in the specified directory
  plt.show()
```

```
# Generate a random seed
seed_baseline = tf.random.normal([16, BASELINE_PARAMS["noise_dim"]])  # You can adjust the first dimension (16) as needed
seed_new = tf.random.normal([16, NEW_PARAMS["noise_dim"]])
```

```
# Function to train the model
def train(dataset, epochs, generator, discriminator, generator_optimizer, discriminator_optimizer, seed):
    for num_epochs in epochs:
        for epoch in range(num_epochs):
            start = time.time()

            for image_batch in dataset:
                baseline_noise= tf.random.normal([BASELINE_BATCH_SIZE, BASELINE_PARAMS["noise_dim"]])
                new_noise= tf.random.normal([NEW_BATCH_SIZE, NEW_PARAMS["noise_dim"]])
                noise = baseline_noise if generator == generator_baseline else new_noise

                train_step(image_batch, generator, discriminator_optimizer, generator_optimizer)
                # Print generator and noise information
                # print("Generator:", generator)
                # print("Noise shape:", noise.shape)

            # Update and display progress
            display.clear_output(wait=True)
            print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

            # Save the model every 15 epochs
            if (epoch + 1) % 15 == 0:
                checkpoint.save(file_prefix=checkpoint_prefix)
            # Save the model after 50 epochs
            if (epoch + 1) == 50:
                generate_and_save_images(generator, epoch + 1, seed)
             # Print generator and noise information
            print("Generator:", generator)
            print("Noise shape:", noise.shape)

  # Generate after the final epoch
    display.clear_output(wait=True)
    generate_and_save_images(generator,
                             sum(epochs),
                             seed)
  #Save for each set of epochs
    checkpoint.save(file_prefix=checkpoint_prefix)

#Create a TensorFlow dataset
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BASELINE_BATCH_SIZE)

#train with baseline hyperparameters
train(train_dataset, [50], generator_baseline, discriminator, generator_optimizer_baseline, discriminator_optimizer, seed_baseline)
#train with new hyperparameters
train(train_dataset, [50], generator_new, discriminator, generator_optimizer_new, discriminator_optimizer, seed_new)
```

As per the requirement of Part C, the following hyperparameters were modified:

- **Noise vector:** The baseline of the noise vector of 100 was reduced to 50. Reduced noise vector meant simplifying the generator's task as it led to the reduction of complexity of the latent space.
- **Batch size:** The batch size of 256 was set to 128 for the modified batch, as utilizing a smaller batch size expedites the training process and also time.
- **Learning rates:** The baseline learning rate was set to 1e-4 but for the modified learning rate, was set to 1e-3. A higher learning rate facilitates faster updates to the model parameters and thus producing faster results.

- **Momentum:** Momentum remained unchanged to Adam Optimizer. This was not modified because Adam had previously proven effectiveness in the optimization of neural network parameters, which includes the generator and discriminator for GAN Training. Given its track record, it was best to not alter the existing optimization process.

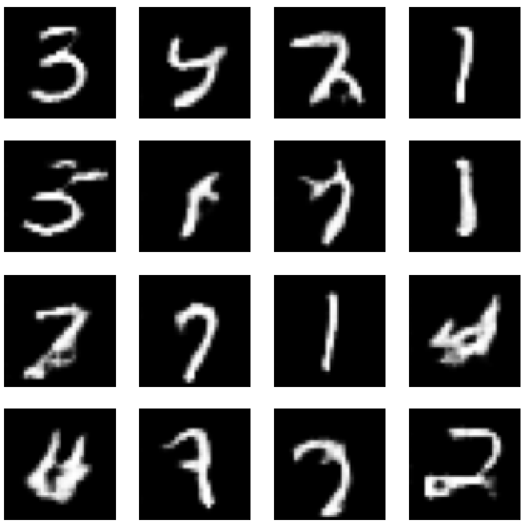


Image at Epoch 50_Baseline

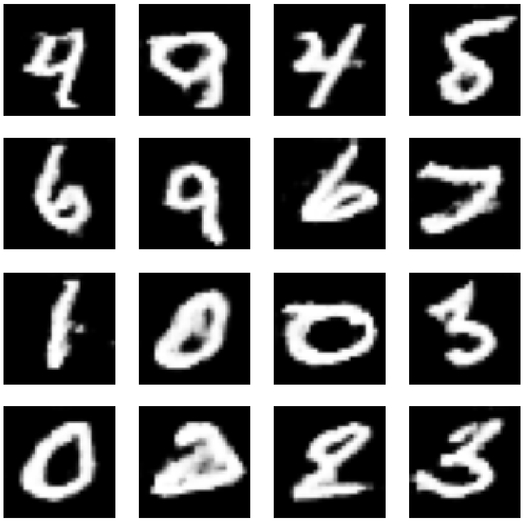


Image at Epoch 50_New

Results: As shown in the images above of **Image at Epoch 50_Baseline** and **Image at Epoch 50_New,** we can observe that with the changed hyperparameter, the numbers in the images tend to be bolder, but most numbers appear fuzzy and lack clarity. Whereas, in the Baseline image, the numbers appear cleaner and sharper making it feasible to interpret.

In addition, decreasing the noise vector dimensionality, led to faster training time. Resulting in enhanced time efficiency compared to the baseline hyperparameters.

References:

1. Deep Convolutional Generative Adversarial Network. Tensorflow [https://www.tensorflow.org/tutorials/generative/dcgan]

2. THE MNIST DATABASE of handwritten digits [http://yann.lecun.com/exdb/mnist/]

3. MNIST database. Wikipedia [https://en.wikipedia.org/wiki/MNIST_database]