|  |  |
| --- | --- |
| **Ex No: 7**  **Date: 18/09/24** | **Generative Adversarial Network Using MNIST Dataset** |

**Objective**:  
To implement a Deep Convolutional Generative Adversarial Network (DCGAN) to generate new images of handwritten digits using the MNIST dataset.

**Description**:  
This exercise implements a DCGAN in three main steps:

1. **Data Preparation**:
   * The MNIST dataset, which consists of 28x28 grayscale images of handwritten digits, is loaded using tf.keras.datasets.
   * These images are normalized and used for training both the generator and discriminator models.
2. **Model Architecture**:
   * **Generator**: The generator starts with a dense layer to expand a noise vector into a 7x7x256 tensor. This is followed by BatchNormalization, LeakyReLU, and Conv2DTranspose layers to upscale the tensor back to the original 28x28x1 image size.
   * **Discriminator**: The discriminator (not shown in the snippets) is likely a series of convolutional layers used to classify the images as real or fake.
3. **Training**:

### 1. Overview of GAN Training:

In a GAN, the **generator** tries to create realistic data (images, in this case) from random noise, while the **discriminator** attempts to distinguish between real data (actual MNIST images) and fake data (generated by the generator). These two models are trained in opposition to one another in what is called an **adversarial process**:

* The generator tries to fool the discriminator by improving its ability to generate realistic images.
* The discriminator tries to get better at detecting fake images from the generator.

This creates a loop where the generator gradually improves at generating realistic images, and the discriminator becomes better at spotting fakes, until an equilibrium is reached.

**2. Detailed Training Steps:**

#### Step 1: Data Preparation:

* **Loading the dataset**: The MNIST dataset is used, which contains 28x28 pixel grayscale images of handwritten digits.

#### Step 2: Generator Training:

The generator starts with random noise (a latent space vector, usually sampled from a normal distribution). During training, it learns to transform this noise into a coherent image that closely resembles the MNIST digits.

The training process for the generator is:

1. **Generate fake images** from random noise using the generator.
2. **Compute the loss** based on how well the discriminator is fooled:
   * If the discriminator classifies the fake image as "real," the generator is rewarded.
   * The generator’s loss is the binary cross-entropy between the discriminator’s predictions (for fake images) and the label "real" (which we want the discriminator to believe).

#### Step 3: Discriminator Training:

The discriminator is a binary classifier that learns to differentiate between real MNIST images and fake images generated by the generator.

The training process for the discriminator is:

1. **Receive real images** from the MNIST dataset.
2. **Receive fake images** generated by the generator.
3. **Classify both sets of images** and compute the loss:
   * The discriminator's goal is to correctly label real images as "real" and fake images as "fake."
   * The discriminator’s loss is computed using binary cross-entropy for both the real and fake images:

#### Step 4: Adversarial Training Loop:

In each training step, both the generator and the discriminator are updated:

1. **Generator Training**:
   * The generator is updated based on how well it fools the discriminator.
   * The generator's weights are optimized using gradients from the loss function computed against the discriminator’s "fake" outputs.

**Discriminator Training**:

* The discriminator is updated based on its ability to correctly classify real and fake images.
* The discriminator’s weights are updated using gradients from the loss function computed on both real and fake inputs.

**Key Training Elements**:

* **Batch training**: For each batch of images, the generator and discriminator are updated in each epoch. The generator creates fake images, which are evaluated by the discriminator along with real images from the dataset.
* **Optimizer**: Adam optimizers are used for both the generator and discriminator to efficiently compute updates to the weights during training.

#### Step 5: Evaluation and Image Generation:

After several training epochs, the generator should improve to the point where it can produce images that are difficult for the discriminator to distinguish from real MNIST digits. The images are sampled and displayed at regular intervals during training to monitor progress:

### 3. Loss Functions:

Both the generator and discriminator use **binary cross-entropy loss**:

* For the generator, the loss function measures how well it can fool the discriminator into thinking generated images are real.
* For the discriminator, the loss function measures its accuracy in distinguishing real images from generated ones.

### 4. Training Loop Summary:

The training loop alternates between:

* Training the discriminator to differentiate between real and fake images.
* Training the generator to fool the discriminator by producing increasingly realistic images.

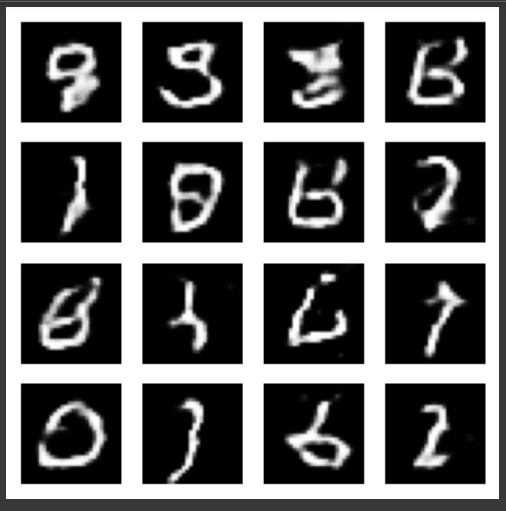
### 5. Final Results:

* Over time, the generator improves, and the generated images become more realistic. The discriminator, in turn, becomes better at distinguishing real from fake, until it becomes increasingly difficult for the generator to fool the discriminator.
* The results are visualized by generating images using the generator and plotting them after different epochs.

**Model**:

* **Generator**:
  + Input: Random noise (100-dimensional vector).
  + Layers: Dense layer (7x7x256), Conv2DTranspose layers to scale up the image size, and BatchNormalization and LeakyReLU activations.
  + Output: 28x28x1 (Grayscale image).

**Result and Analysis**:

* The DCGAN model successfully generated images that closely resemble handwritten digits from the MNIST dataset.
* The generated images improve in quality over training epochs as the generator learns to mimic the distribution of the real dataset.****

**GitHub Link:**