USN NUMBER: 1RVU22CSE071
NAME: Joselyn Riana Manoj

Ex No: 7 **Date: 18-09-2024**	**Deep Convolutional Generative Adversarial Network (DCGAN) for MNIST Dataset**

Objective: To implement a Deep Convolutional Generative Adversarial Network (DCGAN) for generating realistic images of handwritten digits using the MNIST dataset. This network will be used to train a generator and discriminator in an adversarial setup.

Descriptions:
Generative Adversarial Networks (GANs) consist of two models trained simultaneously:

1. **Generator**: A neural network that generates fake data (in this case, images).
2. **Discriminator**: A neural network that classifies data as either real (from the dataset) or fake (generated by the generator).

Both models are trained together, with the generator learning to produce realistic images that fool the discriminator, and the discriminator learning to improve its ability to distinguish between real and fake images. The model is trained using the MNIST dataset, which contains 28x28 grayscale images of handwritten digits. The generator takes in random noise as input and generates images, while the discriminator is trained to classify images as real or fake.

Model:

🎬 Dataset **Preparation**: The MNIST dataset is loaded, and the images are reshaped into 28x28 pixels with one channel (grayscale). The pixel values are normalized to the range [-1, 1] to stabilize the training of the GAN.

🎬 Generator **Architecture**: The generator takes a 100-dimensional noise vector as input, which is transformed through a series of layers:

- **Dense Layer**: The noise is passed through a fully connected layer, reshaped into a smaller image.
- **Batch Normalization**: Batch normalization is applied to stabilize the training process.
- **Leaky ReLU**: This activation function helps with the gradient flow, particularly for GANs.
- **Transpose Convolutions**: The image is upsampled to the target size (28x28) using transposed convolution layers, which increase the image dimensions.

The final output layer uses a Tanh activation function, which ensures that the pixel values are in the range [-1, 1], matching the normalization applied to the training data.

USN NUMBER: 1RVU22CSE071
NAME: Joselyn Riana Manoj

🎬 Discriminator **Architecture**: The discriminator is a convolutional neural network that takes in a 28x28 image and determines whether it is real or generated. The network consists of:

- **Convolutional Layers**: These layers extract features from the input image, reducing the spatial dimensions while increasing the depth.
- **Leaky ReLU Activation**: This activation function is used to help prevent the vanishing gradient problem, allowing the model to learn more effectively.
- **Dropout**: Dropout is applied to prevent overfitting, ensuring that the discriminator generalizes well to unseen data.
- **Dense Layer**: Finally, the output is a single unit representing the probability that the input image is real.

🎬 Adversarial **Training**: The training involves two main processes:

- **Generator Training**: The generator creates fake images from random noise and attempts to fool the discriminator into classifying them as real.
- **Discriminator Training**: The discriminator is trained to classify real images from the MNIST dataset and fake images generated by the generator. It outputs a probability score that indicates whether an image is real or fake.

During each iteration, both the generator and discriminator are updated. The generator improves by receiving feedback from the discriminator, while the discriminator improves by learning to classify more effectively.

Code Explanation Line by Line

- **tensorflow**: TensorFlow is the core deep learning framework used.
- **numpy**: NumPy is used for numerical operations on arrays.
- **matplotlib.pyplot**: Used for plotting and visualizing images.
- **tensorflow.keras.layers**: A collection of pre-built neural network layers.
- **IPython.display**: Used for updating image displays during training.
- **tf.keras.datasets.mnist.load_data()**: Loads the MNIST dataset of handwritten digits.
- **train_images.reshape()**: Reshapes the training images into a 4D tensor of shape (number_of_images, 28, 28, 1) to include a channel dimension.
- **astype('float32')**: Converts the pixel values to float32 format.
- **(train_images - 127.5) / 127.5**: Normalizes pixel values to the range [-1, 1] from [0, 255], which helps the model converge faster.
- **BUFFER_SIZE = 60000**: Sets the buffer size for shuffling the dataset.
- **BATCH_SIZE = 256**: Defines the size of each batch for training.
- **from_tensor_slices()**: Converts the dataset into a format suitable for TensorFlow.
- **shuffle(BUFFER_SIZE)**: Shuffles the data to ensure randomness during training.
- **batch(BATCH_SIZE)**: Batches the data for efficient processing.
- **tf.keras.Sequential()**: Initializes a sequential model, where layers are added one after another.

- **layers.Dense(7*7*256, use_bias=False, input_shape=(100,))**: First, a dense (fully connected) layer is created. The output dimension is 7*7*256, which will be reshaped later into a 7x7x256 image. The input is a 100-dimensional noise vector (random noise).
- **layers.BatchNormalization()**: Normalizes the activations, stabilizing and accelerating training.
- **layers.LeakyReLU()**: Leaky ReLU is used as the activation function, helping prevent vanishing gradients.
- **layers.Reshape((7, 7, 256))**: Reshapes the output from the dense layer into a 7x7x256 image.
- **layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False)**: This is a transposed convolutional layer (also known as deconvolution) used to upsample the image. It uses 128 filters of size 5x5.
- **layers.BatchNormalization()**: Again, batch normalization is applied to stabilize training.
- **layers.LeakyReLU()**: Activation function to add non-linearity.
- **layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False)**: Another transposed convolutional layer with 64 filters and a stride of 2x2, effectively doubling the image size from 7x7 to 14x14.
- **layers.BatchNormalization()**: Further batch normalization.
- **layers.LeakyReLU()**: Activation function.
- **layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh')**: The final transposed convolutional layer produces a 28x28x1 output (an image). The tanh activation function is used to output values in the range [-1, 1], which matches the range of the normalized training images.
- **make_generator_model()**: Initializes the generator model.
- **tf.random.normal([1, 100])**: Generates a batch of random noise vectors, each of length 100.
- **generator(noise, training=False)**: Passes the noise through the generator to produce a fake image.
- **plt.imshow()**: Displays the generated image using matplotlib.
- **layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape=[28, 28, 1])**: The discriminator starts with a convolutional layer that takes a 28x28x1 image as input and applies 64 filters of size 5x5. The strides of 2x2 reduce the image size by half (to 14x14).
- **layers.LeakyReLU()**: Leaky ReLU activation function.
- **layers.Dropout(0.3)**: A dropout layer is applied to prevent overfitting. 30% of neurons are randomly turned off during training.
- **layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same')**: Another convolutional layer with 128 filters, reducing the image size further to 7x7.
- **layers.LeakyReLU()**: Leaky ReLU activation.

- **layers.Dropout(0.3)**: Dropout layer to prevent overfitting.
- **layers.Flatten()**: Flattens the image into a 1D vector for the final classification layer.
- **layers.Dense(1)**: A dense layer with one output node. This produces a single value representing whether the image is real or fake (discriminator output).
- **make_discriminator_model()**: Initializes the discriminator model.
- **discriminator(generated_image)**: Tests the discriminator by feeding it the generated image.
- **print(decision)**: Prints the discriminator's decision, a value that indicates how real or fake the image is.
- **tf.keras.losses.BinaryCrossentropy(from_logits=True)**: Defines the binary cross-entropy loss function, which compares the output of the discriminator to the true labels (real or fake). from_logits=True means that the discriminator outputs unscaled values (logits).
- **generator_loss(fake_output)**: Calculates the loss for the generator. It compares the discriminator's classification of fake images with a target of 1 (real). The generator's goal is to fool the discriminator into thinking the images are real.
- **discriminator_loss(real_output, fake_output)**: Calculates the loss for the discriminator by comparing its classification of real images to 1 (real) and fake images to 0 (fake). The total loss is the sum of these two losses.

Results
By training for multiple epochs, the generator gradually improves, producing more realistic images as the discriminator becomes more effective at identifying real versus fake images. Early in training, the generated images are blurry and unrealistic, but as training progresses, the generator learns to produce clearer and more accurate representations of handwritten digits.

Conclusion
The DCGAN implementation successfully demonstrates the adversarial training process between a generator and discriminator. Over time, the generator learns to produce realistic images, and the discriminator becomes adept at distinguishing between real and fake images. The balance between the two models leads to the generation of visually plausible digit images from random noise.

GitHub Link:
https://github.com/joselynrianaaa/DeepLearning_Labs/blob/main/JoselynRiana_dcgan_DISTRI.ipynb