Generative Adversarial Networks (GANs) Using MNIST Dataset

Introduction to GANs

Generative Adversarial Networks (GANs) are a class of deep learning models introduced by Ian Goodfellow in 2014. GANs consist of two neural networks, the Generator and the Discriminator, that compete against each other in a zero-sum game framework. The Generator aims to create realistic data (such as images) from random noise, while the Discriminator's goal is to differentiate between real and fake data. Over successive training iterations, the Generator improves its ability to produce realistic data, while the Discriminator becomes better at distinguishing fake data from real.

Key advantages of GANs include their ability to generate high-quality synthetic data and their applications in fields such as image synthesis, data augmentation, and unsupervised learning.

Dataset Description and Preprocessing Steps

Dataset Description

The MNIST dataset consists of 70,000 grayscale images of handwritten digits (0-9), where each image has dimensions of 28x28 pixels. This dataset is widely used for benchmarking machine learning models and serves as an excellent choice for training a GAN to generate digit-like images.

Preprocessing Steps

1. **Normalization:** The pixel values of the MNIST images, originally in the range [0, 255], are normalized to the range [-1, 1]. This normalization ensures faster convergence and stable training of the GAN because the Generator's output uses the `tanh` activation function, which outputs values in this range.
2. **Adding Channel Dimension:** Each image is reshaped to include a channel dimension to match the expected input shape for convolutional layers in the models (28x28x1).
3. **Batching and Shuffling:** The dataset is divided into batches of size 128 and shuffled to ensure the model sees diverse examples during training.

```
# Load MNIST dataset
(x_train, _), (_, _) = mnist.load_data()
# Normalize data to range [-1, 1]
x_train = x_train.astype("float32") / 127.5 - 1.0
# Add channel dimension
x_train = np.expand_dims(x_train, axis=-1)
# Create batches
batch_size = 128
```

```
dataset =
tf.data.Dataset.from_tensor_slices(x_train).shuffle(60000).batch(batc
h_size)
```

Detailed Explanation of Model Architectures

Generator Architecture

The Generator takes a random noise vector as input and transforms it into a 28x28 grayscale image using a series of fully connected and convolutional layers.

Layers in the Generator:

1. **Input Layer:** A dense layer maps the 100-dimensional noise vector to 256 units.
2. **Hidden Layers:**

- Three fully connected layers progressively increase the feature size (256 -> 512 -> 1024).
- Batch normalization layers stabilize training by normalizing activations.
- LeakyReLU activation introduces non-linearity and avoids dying neurons.

3. **Output Layer:** A dense layer reshapes the output to 28x28x1 using a `tanh` activation to match the pixel range of the MNIST images.

```
def build_generator():
    model = tf.keras.Sequential([
        layers.Dense(256, input_shape=(100,)),
        layers.LeakyReLU(alpha=0.2),
        layers.BatchNormalization(momentum=0.8),
        layers.Dense(512),
        layers.LeakyReLU(alpha=0.2),
        layers.BatchNormalization(momentum=0.8),
        layers.Dense(1024),
        layers.LeakyReLU(alpha=0.2),
        layers.BatchNormalization(momentum=0.8),
        layers.Dense(28 * 28 * 1, activation='tanh'),
        layers.Reshape((28, 28, 1))
    ])
    return model
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_7 (Dense)              (None, 256)               25856
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 256)               0
_________________________________________________________________
batch_normalization_3 (Batch (None, 256)               1024
_________________________________________________________________
dense_8 (Dense)              (None, 512)               131584
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 512)               0
_________________________________________________________________
batch_normalization_4 (Batch (None, 512)               2048
_________________________________________________________________
dense_9 (Dense)              (None, 1024)              525312
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 1024)              0
_________________________________________________________________
batch_normalization_5 (Batch (None, 1024)              4096
_________________________________________________________________
dense_10 (Dense)             (None, 784)               803600
_________________________________________________________________
reshape_1 (Reshape)          (None, 28, 28, 1)         0
...
Total params: 1,493,520
Trainable params: 1,489,936
Non-trainable params: 3,584
```

Discriminator Architecture

The Discriminator is a binary classifier that determines whether an input image is real or fake.

Layers in the Discriminator:

1. **Input Layer:** Flattens the 28x28x1 image into a 784-dimensional vector.
2. **Hidden Layers:**

- Two dense layers progressively reduce the feature size (512 -> 256).
- LeakyReLU activation introduces non-linearity.
- Dropout layers (rate = 0.3) help prevent overfitting.

3. **Output Layer:** A dense layer with a `sigmoid` activation outputs a probability indicating whether the input is real or fake.

```
def build_discriminator():
```

```
    model = tf.keras.Sequential([
        layers.Flatten(input_shape=(28, 28, 1)),
        layers.Dense(512),
        layers.LeakyReLU(alpha=0.2),
        layers.Dropout(0.3),
        layers.Dense(256),
        layers.LeakyReLU(alpha=0.2),
        layers.Dropout(0.3),
        layers.Dense(1, activation='sigmoid')
    ])
    return model
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_11 (Dense)             (None, 512)               401920
_________________________________________________________________
leaky_re_lu_8 (LeakyReLU)    (None, 512)               0
_________________________________________________________________
dropout (Dropout)            (None, 512)               0
_________________________________________________________________
dense_12 (Dense)             (None, 256)               131328
_________________________________________________________________
leaky_re_lu_9 (LeakyReLU)    (None, 256)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_13 (Dense)             (None, 1)                 257
=================================================================
Total params: 533,505
Trainable params: 533,505
Non-trainable params: 0
```

GAN Architecture

The GAN combines the Generator and Discriminator into a single model. During training, the Discriminator is frozen to ensure gradients flow only through the Generator.

```
def build_gan(generator, discriminator):
    discriminator.trainable = False
    model = tf.keras.Sequential([generator, discriminator])
    return model
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
sequential_3 (Sequential)    (None, 28, 28, 1)         1493520
_________________________________________________________________
sequential_4 (Sequential)    (None, 1)                 533505
=================================================================
Total params: 2,027,025
Trainable params: 1,489,936
Non-trainable params: 537,089
```

Explanation of Loss Functions and Training Strategy

Loss Functions

1. **Discriminator Loss:** Measures how well the Discriminator distinguishes between real and fake images. It is the sum of:

- **Real Loss:** Binary crossentropy loss for real images (label = 1).
- **Fake Loss:** Binary crossentropy loss for fake images (label = 0).

```
real_loss = loss_fn(tf.ones_like(real_output), real_output)
fake_loss = loss_fn(tf.zeros_like(fake_output), fake_output)
disc_loss = real_loss + fake_loss
```

2. **Generator Loss:** Measures how well the Generator fools the Discriminator. The goal is to maximize the Discriminator's error on fake images (label = 1).

```
gen_loss = loss_fn(tf.ones_like(fake_output), fake_output)
```

Training Strategy

1. **Training the Discriminator:**

- Use real images from the dataset and fake images generated by the Generator.
- Compute real and fake losses and update the Discriminator's weights.

2. **Training the Generator:**

- Generate fake images from random noise.
- Pass these images through the Discriminator and compute the Generator loss.
- Update the Generator's weights to improve its ability to fool the Discriminator.

3. **Iterative Training:**

- Alternate between training the Discriminator and Generator.
- Repeat for multiple epochs until the Generator produces realistic images.

```
@tf.function
def train_step(real_images):
    batch_size = tf.shape(real_images)[0]
    random_noise = tf.random.normal([batch_size, 100])

    # Train Discriminator
    with tf.GradientTape() as disc_tape:
        fake_images = generator(random_noise, training=True)
        real_output = discriminator(real_images, training=True)
        fake_output = discriminator(fake_images, training=True)

        real_loss = loss_fn(tf.ones_like(real_output), real_output)
        fake_loss = loss_fn(tf.zeros_like(fake_output), fake_output)
        disc_loss = real_loss + fake_loss

    disc_grads = disc_tape.gradient(disc_loss,
discriminator.trainable_variables)
    discriminator_optimizer.apply_gradients(zip(disc_grads,
discriminator.trainable_variables))

    # Train Generator
    with tf.GradientTape() as gen_tape:
        generated_images = generator(random_noise, training=True)
        fake_output = discriminator(generated_images, training=True)
        gen_loss = loss_fn(tf.ones_like(fake_output), fake_output)

    gen_grads = gen_tape.gradient(gen_loss,
generator.trainable_variables)
    generator_optimizer.apply_gradients(zip(gen_grads,
generator.trainable_variables))

    return disc_loss, gen_loss
```

Evaluation and Visualizations of Results