Generative Models: Conditional Generative Adversarial Networks (cGAN)

Overview

Conditional Generative Adversarial Networks (cGANs) are a variant of the original GANs (Generative Adversarial Networks), where both the generator and the discriminator receive additional information, called "conditions," along with the usual random noise (for the generator) or real/fake labels (for the discriminator). This allows the model to generate data conditioned on certain inputs, making it useful for tasks such as image generation conditioned on labels (e.g., generating images of digits conditioned on their label).

Key Features of cGAN:

- **Generator**: Takes random noise and a condition (e.g., class label) to produce an output (e.g., an image).
- **Discriminator**: Receives an image and its condition, and predicts whether the image is real or generated.
- **Conditioning**: The conditioning allows the model to learn specific data distributions for different categories, enabling more control over the generated outputs.

Why Use cGAN?

cGANs are used in scenarios where you want to generate samples that adhere to specific conditions:

- **Image Generation**: Generate images conditioned on labels (e.g., generating different digits in MNIST).
- **Image-to-Image Translation**: Generate images from sketches, or convert daytime images to nighttime images, etc.
- **Text-to-Image**: Generate images from textual descriptions.
- **Data Augmentation**: Generate more labeled data for training classifiers.

Prerequisites

- **Python 3.x**: Ensure Python 3.x is installed.

- **PyTorch**: Install PyTorch for building the GAN model.

  ```
  pip install torch torchvision
  ```

Code Description

1. Generator Model

```
 import torch
import torch.nn as nn

class Generator(nn.Module):
    def __init__(self, z_dim, condition_dim, img_dim):
        super(Generator, self).__init__()
        self.fc = nn.Sequential(
```

```
            nn.Linear(z_dim + condition_dim, 128),
            nn.ReLU(True),
            nn.Linear(128, img_dim),
            nn.Tanh()
        )

    def forward(self, z, condition):
        x = torch.cat([z, condition], dim=1)  # Concatenate noise z and condition
        return self.fc(x)
```

Explanation:

- **Generator Input**: The generator receives two inputs: random noise vector `z` and a `condition` (e.g., class label).
- **Fully Connected Layer**: A fully connected network maps the concatenated input (noise + condition) to the output image dimensions.
- **Activation Functions**: ReLU and Tanh are used for the hidden layers and the output, respectively, where Tanh ensures that the output is scaled to [-1, 1] (common for image generation).

---

2. Discriminator Model

```
class Discriminator(nn.Module):
    def __init__(self, img_dim, condition_dim):
        super(Discriminator, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(img_dim + condition_dim, 128),  # Concatenate image and condition
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(128, 1),
            nn.Sigmoid()
        )

    def forward(self, img, condition):
        x = torch.cat([img, condition], dim=1)  # Concatenate image and condition
        return self.fc(x)
```

Explanation:

- **Discriminator Input**: The discriminator receives the image and the condition (e.g., label) as input.
- **Fully Connected Layer**: The image and condition are concatenated and passed through a fully connected network.
- **Leaky ReLU and Sigmoid**: Leaky ReLU is used for hidden layers, and Sigmoid ensures the output is a probability (real or fake).

---

3. Model Initialization

```
 z_dim = 100  # Dimensionality of the random noise vector
condition_dim = 10  # Example: number of conditions (e.g., label)
img_dim = 784  # Example: flattened image of size 28x28 (MNIST dataset)

# Create the generator and discriminator
generator = Generator(z_dim, condition_dim, img_dim)
discriminator = Discriminator(img_dim, condition_dim)
```

Explanation:

- **Dimensions**:
    - `z_dim` is the dimensionality of the noise vector.

    - `condition_dim` represents the number of possible conditions (e.g., for MNIST, this could be 10, corresponding to the 10 digits).
    - `img_dim` is the dimensionality of the image (e.g., 28x28 pixels flattened into a 784-dimensional vector for MNIST).

- **Model Creation**: Both the generator and discriminator are instantiated using the specified dimensions.

---

4. Training Loop (Sketch)

While the full training loop is not included, here is a simplified sketch of the process:

```
 # Hyperparameters
lr = 0.0002
batch_size = 64
epochs = 100
z_dim = 100
condition_dim = 10
img_dim = 784

# Optimizers
optimizer_g = optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.999))
optimizer_d = optim.Adam(discriminator.parameters(), lr=lr, betas=(0.5, 0.999))

# Loss function
criterion = nn.BCELoss()

for epoch in range(epochs):
    for i, (imgs, labels) in enumerate(dataloader):
        # Create labels for real and fake images
        real_labels = torch.ones(batch_size, 1)
        fake_labels = torch.zeros(batch_size, 1)

        # Generate random noise and conditions
        z = torch.randn(batch_size, z_dim)
        conditions = torch.eye(condition_dim)[labels]

        # Create fake images
        fake_imgs = generator(z, conditions)

        # Train discriminator on real and fake images
        optimizer_d.zero_grad()
        real_preds = discriminator(imgs, conditions)
        fake_preds = discriminator(fake_imgs.detach(), conditions)
        loss_d = criterion(real_preds, real_labels) + criterion(fake_preds, fake_labels
        loss_d.backward()
        optimizer_d.step()

        # Train generator
        optimizer_g.zero_grad()
        fake_preds = discriminator(fake_imgs, conditions)
        loss_g = criterion(fake_preds, real_labels)  # Generator tries to fool discrimi
        loss_g.backward()
        optimizer_g.step()

    print(f'Epoch [{epoch+1}/{epochs}], Loss D: {loss_d.item()}, Loss G: {loss_g.item()}
```

Explanation:

- **Training Process**:
    - The generator produces fake images based on random noise and conditions.
    - The discriminator is trained to distinguish between real and fake images.
    - The generator is trained to produce images that can fool the discriminator.
- **Loss Functions**: Binary cross-entropy (BCE) is used to train both the discriminator and generator.

Expected Outputs

- **Generator Output**: The generator will produce images based on the input noise and conditions.
- **Discriminator Output**: The discriminator will output the probability that an image is real or fake.

Use Cases

- **Image Generation**: cGANs can generate realistic images conditioned on certain labels (e.g., digits, objects, scenes).
- **Style Transfer**: cGANs can generate images conditioned on specific styles or attributes.
- **Text-to-Image Synthesis**: cGANs can generate images from textual descriptions by conditioning on text embeddings.

Future Enhancements

1. **Improved Architectures**: Experiment with more complex architectures like Wasserstein GANs (WGANs) or Progressive GANs for better performance.
2. **Conditioning on Multiple Inputs**: Explore conditioning on more complex or multi-dimensional conditions like text or bounding boxes.
3. **Higher-Resolution Images**: Use more advanced techniques for generating higher-resolution images, such as multi-scale architectures.

References

- Goodfellow, I., et al. (2014). *Generative Adversarial Nets*. Link
- Mirza, M., & Osindero, S. (2014). *Conditional Generative Adversarial Nets*. Link