GENERATIVE ADVERSARIAL NETWORK(GAN) (Report)

The code is implementation the Deep Convolutional Generative Adversarial Network (DCGAN) architecture using the PyTorch framework. DCGAN is composed of two neural networks, a Discriminator and a Generator, that are trained together in a minimax game.

1. **Discriminator (class Discriminator)**: This is a convolutional neural network that takes an image as input and outputs a probability indicating whether the input image is real or fake. The _block function is a helper function that creates a sequential block with a Conv2d layer, followed by BatchNorm2d and LeakyReLU activation. The Discriminator has several of these blocks, with an increasing number of feature maps.
2. **Generator (class Generator):** This is a convolutional neural network that takes a random noise vector as input and generates an image. The _block function is a helper function that creates a sequential block with a ConvTranspose2d layer, followed by BatchNorm2d and ReLU activation. The Generator has several of these blocks, with a decreasing number of feature maps. Finally, a ConvTranspose2d layer is used to generate an image with the required number of channels, and a Tanh activation is applied to normalize the output.
3. **initialize_weights function**: This function initializes the weights of the model according to the DCGAN paper. It sets the weights of Conv2d, ConvTranspose2d, and BatchNorm2d layers to have a mean of 0 and a standard deviation of 0.02.
4. **test function** (commented out): This is a simple test function that creates instances of the Discriminator and Generator classes, then checks if their output shapes are as expected. If the tests pass, it prints "Success, tests passed!". The code provided is a complete implementation of the DCGAN architecture, and it can be used as a starting point to train the models on a dataset of images.

(a)**Training DCGAN**
Hyperparameters: learning rate(0.0002), batch size(128), image size(64x64), number of features in the generator(64) and discriminator(64), number of channels in the input image(3), and noise dimensions(100).

Data augmentation and dataset loading: The code uses the PyTorch transforms module to apply transformations like resizing and normalization on the input images. It then loads the dataset, filters it to only include dog images, and creates a DataLoader with the specified batch size.

Optimizers and loss function: Adam optimizers are created for the Generator and Discriminator with the specified learning rate and betas. The Binary Cross Entropy (BCE) loss function is used for training.

Training: The code trains the model for the specified number of epochs. During each epoch, the Generator and Discriminator are trained using the minimax loss functions. The Discriminator's accuracy and loss values are calculated and printed. At the end of each epoch, generated images are saved, and the average loss values are printed.

Saving the trained weights: After training, the weights of the Generator model are saved.

The DCGAN training is progressing through multiple epochs. An overview of the training progress and describe the key observations:
During the training process, several metrics are tracked to assess the performance of both the generator (G) and the discriminator (D). The metrics include average loss for the generator (Avg Loss G), average loss for the discriminator (Avg Loss D), and average accuracy for the discriminator (Avg Acc D).

From Epoch 1 to Epoch 268, the following trends and observations can be made:

1. The average loss for the generator (Avg Loss G) generally increases at the beginning of the training, reaching its peak around Epoch 3, and then decreases before stabilizing and increasing again from Epoch 191 onwards.
2. The average loss for the discriminator (Avg Loss D) starts relatively high and then decreases over time, suggesting that the discriminator is getting better at distinguishing between real and generated images.
3. The average accuracy for the discriminator (Avg Acc D) starts high and gradually decreases before stabilizing and increasing again from Epoch 191 onwards. This indicates that the discriminator's performance in classifying real and generated images is improving after Epoch 191.
4. Training time per epoch seems to be relatively stable, with some variations between epochs. This could be due to fluctuations in the system's resource utilization or other factors.

In summary, the training progress of the DCGAN shows that both the generator and the discriminator are learning and improving their performance over time. The generator's loss initially decreases, then stabilizes and starts increasing again, while the discriminator's loss and accuracy show a general trend of improvement throughout the training process.

(b) **Hyper parameters Tuning:**

We have performed Tuning of learning rate, Optimizers and number of Convolution layers.
The output and images generated can be seen in the notebook with graphs.
And then decide which parameters are producing better results.

(c) **Modifying the DCGAN architecture by implementing and adding the mapping network from z-space to w-space from StyleGAN2.**

Implementation of a generative model with adaptive instance normalization (AdaIN). The code consists of four classes: MappingNetwork, AdaIN, GeneratorWithMapping, and Discriminator.

1. **MappingNetwork:** This class represents a feed-forward neural network with a given number of layers (default is 4). Each layer consists of a linear transformation followed by a Leaky ReLU activation function with a slope of 0.2. The purpose of this network is to map the input noise to a latent space that can be used in the generator.
2. **AdaIN:** The AdaIN class is a custom normalization layer that applies adaptive instance normalization to the input feature maps. It takes an input tensor x and applies instance normalization (without affine parameters) followed by element-wise scaling and shifting using the provided scale and shift tensors.
3. **GeneratorWithMapping:** This class integrates the MappingNetwork and the generator network, which is responsible for generating images based on the input noise. The generator network is not explicitly defined in the code provided, but it is assumed to be a class named Generator (in original DCGAN architecture) that takes the noise dimension, image channels, and a feature multiplier as input arguments. The GeneratorWithMapping class also constructs a list of AdaIN layers that are applied to the generator's output at each BatchNorm2d layer.
4. **Discriminator:** This class implements the discriminator network, which is responsible for classifying whether an input image is real or generated. The discriminator consists of a series of convolutional layers, each followed by instance normalization (with affine parameters) and a LeakyReLU activation function. The output of the network is a single value between 0 and 1, representing the probability that the input image is real. The generative model can be used in a generative adversarial network (GAN) setting, where the generator and discriminator are trained together in a minimax game to generate realistic images.

Observation:
By the changing the DCGAN architecure, in the output, generated images have the same distribution and no variance, it is likely that the GAN is experiencing mode collapse. Mode collapse occurs when the generator produces a limited variety of images or even the same image repeatedly. This can happen due to a few reasons:

Imbalance between generator and discriminator: If the discriminator is too strong compared to the generator, the generator might find it difficult to produce diverse samples that can fool the discriminator. In response, it may focus on generating only a few samples that the discriminator has trouble distinguishing.

Vanishing gradients: The generator may struggle to learn if it receives weak gradient signals from the discriminator. This can happen when the discriminator is very confident about its predictions, causing the gradients to become very small and limiting the generator's ability to improve.

Limited mapping network capacity: The mapping network in the GeneratorWithMapping class might be too simple, which could prevent it from learning a more diverse set of transformations to apply to the input noise. This can result in limited diversity in the generated images.

To address these issues and improve the diversity of the generated images, we can try the following:

1. Adjust the architecture or training hyperparameters, such as learning rate or batch size, to ensure a more balanced training process between the generator and discriminator.
2. Use techniques like gradient penalty or spectral normalization to stabilize the training process and prevent vanishing gradients.
3. Increase the capacity of the mapping network by adding more layers or increasing the number of hidden units.
4. Use a different loss function, like Wasserstein loss or hinge loss, which can help stabilize the training process and encourage the generator to produce more diverse images.
5. Experiment with different normalization techniques, like using Batch Normalization instead of Instance Normalization, or vice versa.

As GANs can be sensitive to hyperparameters and architectural choices, so it might require some experimentation to find the best configuration for your specific problem.

(d) **Traverse in the latent space of GAN and report your observations and images generated.**

Observations:

1. Lack of diversity: Since the modified DCGAN is experiencing mode collapse, we observe that the generated images look very similar or even identical.
2. Difficulty in traversing the latent spaces: When mode collapse occurs, traversing both Z-space and W-space may not result in smooth transitions or meaningful interpolations between images.
3. No clear improvement between Z-space and W-space: If the mapping network in the GeneratorWithMapping class does not effectively learn to disentangle the input noise, you may not observe any significant improvement in the generated images when comparing Z-space and W-space.

To address mode collapse, you can try the suggestions mentioned in the above such as adjusting the architecture, training hyperparameters, or using different loss functions and normalization techniques.

(e) **Changes in Architecture: To further improve the quality/diversity of the generated images in the modified DCGAN.**

Along with the mapping network and AdaIN, Here are some changes that can be made to the modified DCGAN architecture to improve the quality and diversity of the generated images:

1. Progressive Growing: Gradually increasing the resolution of the generated images by progressively adding layers to the generator and discriminator can help stabilize training and produce higher-quality images. It is a technique used in StyleGAN where the model is trained incrementally, starting with low-resolution images and then increasing the resolution over time. This is done by gradually adding layers to the generator and discriminator during training.

2.Skip Connections: Skip connections are used to directly connect the output of a layer to the input of another layer further down the network, enabling better information flow and alleviating the vanishing gradient problem.

3.Change activation function in MappingNetwork: Instead of using LeakyReLU with a negative slope of 0.2, we can try using the ReLU activation function. ReLU is computationally efficient and could help the network learn non-linear features better.

4.Increase the depth of MappingNetwork: Increasing the number of layers in the MappingNetwork can help learn more complex mappings from the input noise to the latent space. However, this may also increase the risk of overfitting and make the model more challenging to train.

5.Add residual connections: Add residual connections to the generator to help with gradient flow and stabilize training. This can be achieved by adding a residual block that combines the output of a series of convolutions with the input of the block.

6.Use Spectral Normalization: Apply spectral normalization to the weights of both the generator and discriminator networks. This normalization technique can help stabilize training and improve the quality of the generated images.

7.Replace InstanceNorm2d with BatchNorm2d: In the Discriminator, you can try replacing InstanceNorm2d with BatchNorm2d. Batch normalization helps in training deep networks by reducing the internal covariate shift. It also acts as a regularizer, reducing the need for dropout in some cases.

8.Use different loss functions: Experiment with different loss functions, such as the Wasserstein GAN loss or the hinge loss. These alternative loss functions could lead to better convergence and improved image quality.

9.Add dropout layers: Add dropout layers to the generator and discriminator architectures. Dropout layers can help regularize the network and prevent overfitting, which can lead to better performance in terms of image quality and diversity.

Note that these changes may increase the complexity of the model and training time. The improvements in image quality and diversity may vary depending on the specific problem and dataset.