Report on CV Assignment 2

Owais Aijaz

December 1, 2024

1 Introduction

The objective of this project was to implement a Deep Convolutional Generative Adversarial Network (DCGAN) for generating images. The DCGAN was trained on the Stanford Cars dataset to generate car images. This report covers the various phases of the project, challenges faced, and insights gained during the process.

2 References

The following references were crucial to the implementation and understanding of DCGANs:

- **What are Generative Adversarial Networks (GANs)?** - `https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/`
- **DCGAN Tutorial (PyTorch)** - `https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html`
- **StyleGAN2 (PyTorch)** - `https://github.com/lucidrains/stylegan2-pytorch`

3 Setup and Initial Implementation

3.1 Environment Setup

To set up the environment for DCGAN training, the following libraries were imported:

```
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torch.utils.data
```

Additionally, other dependencies like `kagglehub` were used to download datasets:

```
import kagglehub
path = kagglehub.dataset_download("jessicali9530/stanford-cars-dataset")
```

The `stanford-cars-dataset` was downloaded from Kaggle, and the images were resized to 64x64 for input into the DCGAN.

3.2 Dataset Preprocessing

The Stanford Cars dataset was preprocessed by resizing all images to 64x64 and normalizing the pixel values. The dataset was loaded using PyTorch's `ImageFolder` and transformed using the following sequence:

```
transform=transforms.Compose([
    transforms.Resize(64),
    transforms.CenterCrop(64),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])
```

This transformation sequence ensured that the images were appropriately scaled for DCGAN.

3.3 Dataloader Creation

The dataset was then passed to a DataLoader to handle batching and shuffling:

```
dataloader = torch.utils.data.DataLoader(dataset, batch_size=128, shuffle=True, num_
```

This dataloader was used to efficiently manage the training process.

4 Model Implementation

4.1 DCGAN Architecture

The DCGAN architecture consists of two primary components:

- **Generator**: A neural network that takes random noise as input and generates images.
- **Discriminator**: A neural network that attempts to distinguish between real and generated images.

The architecture was defined using PyTorch's `nn.Module`. The generator uses transposed convolutions to upsample the input noise into an image, while the discriminator uses regular convolutions to classify the input as real or fake.

4.2 Training Setup

The models were trained using the following optimization settings:

- Batchsize = 128
- Epochs = 50
- Latent Vector = 100
- Optimizer: Adam
- Learning Rate: 0.0002
- Beta1: 0.5
- Loss Function: Binary Cross-Entropy Loss

5 Challenges and Observations

5.1 What Worked

The overall training pipeline, including data loading, model definition, and training loop, functioned as expected. The generator was able to generate images that resembled cars after a few epochs of training. The discriminator was able to distinguish real and fake images with decent accuracy.

5.2 What Didn't Work

One issue encountered during training was overfitting, where the discriminator became too powerful compared to the generator. This made it difficult for the generator to improve, as the discriminator could easily classify its outputs as fake. Another issue was slow training times due to the large size of the dataset and the complexity of the model.

5.3 What I Learned

- The importance of balancing the generator and discriminator during training. The adversarial nature of GANs requires that both networks improve simultaneously for effective results.
- Fine-tuning the hyperparameters, such as learning rate and batch size, is crucial for efficient training.
- Using a pretrained model (like StyleGAN) can speed up the process and improve the quality of generated images.

5.4 What Extra Things I Tried

To improve training, I experimented with the following:

- **Adding Dropout Layers**: To regularize the models and avoid overfitting.
- **Changing Optimizer Parameters**: Testing different values for Adam's `beta1` parameter.
- **Different Discriminator and Generator Architectures**: I tried different combination of layers in both, experimenting with multiple activation functions and spectral normalization, etc.

6 Experimenting with VGG-16 as the Discriminator

In addition to the original DCGAN setup, I also experimented with replacing the DCGAN discriminator with a VGG-16 network. The idea was to leverage the pre-trained VGG-16, which is a powerful model for image recognition tasks, to improve the discriminator's ability to distinguish between real and generated images.

The results were significantly better than the original DCGAN model in terms of both loss values and image quality. Specifically, the discriminator loss ('Loss_D') and the generator loss ('Loss_G') were much lower when VGG-16 was used as the discriminator, and the generated images appeared more realistic.

6.1 Results and Comparison

When using the original DCGAN architecture, the losses fluctuated significantly over training, and the generated images were often blurry and unrealistic. However, when using VGG-16 as the discriminator, the training stabilized much faster, and the generator started producing sharper and more detailed images. The discriminator's performance was also improved, as expected due to VGG-16's deep convolutional layers and pre-trained weights.

Summary of Results:

- **DCGAN Discriminator:** The losses were unstable, with large fluctuations. The generated images were less detailed.
- **VGG-16 Discriminator:** Much better stability in losses. The generated images were sharper and more realistic.

7 Next Steps and Future Work

7.1 Improvement Strategies

To proceed with this project, the following steps can be taken:

- Experiment with other architectures, such as Wasserstein GANs (WGANs), to mitigate mode collapse and improve training stability.
- Implement more advanced techniques like progressive growing of GANs for better image quality.
- Fine-tune the hyperparameters further, especially the learning rate and batch size, to achieve better convergence.

7.2 Further Exploration

Additionally, exploring conditional GANs (cGANs) to generate car images based on specific attributes (e.g., car color, model) could provide more useful and diverse results.

8 Conclusion

The project successfully implemented a basic DCGAN model capable of generating car images. While challenges related to model convergence and overfitting were faced, valuable lessons were learned, including the importance of hyperparameter tuning and balancing the generator and discriminator during training. With further experimentation, the model's performance can be improved.