Project Report - ECE 176
Reimplementation of Deep Convolutional Generative Adversarial Networks

Xiaojie Chen
UCSD ECE
A17015417

Congge Xu
UCSD ECE
A17402388

Abstract

In this project report, we aim to explore the efficacy of DCGANs in generating high-quality images, focusing on the Stanford Dog Dataset. It introduces the background, tentative methods, the dataset we plan to use and the results of our experiments using different hyper-parameters.

1 Introduction

With the remarkable success achieved through the combination of CNNs and supervised learning, attention has shifted towards exploring how CNNs can be effectively integrated with unsupervised learning to advance its development. As GANs frequently find application in unsupervised learning, a challenge arises due to the instability of training, often resulting in the generation of nonsensical output by the generator. Recognizing this issue, a new architecture called Deep Convolution GANs (DCGAN)[1] has emerged, combining the structures of CNNs and GANs to address the instability of training in various settings. In our project, we aim to reimplement this DCGAN model and optimize hyperparameters through different experiments.

By training the model, we hope to generate dog images with 120 breeds and the generated dog images should be recognizable and identifiable as dogs. Our motivation for this project is that as the ability of DCGANs to produce visually appealing and creative content has implications entertainment, being able to generate cute dog pictures will simply make us (dog persons) feel happy:) Our understanding to this reimplementation is that hyperparameter tuning and subtle algorithm changes are necessary to make to adjust for different dataset. We also played around with optimizers to see how it affects the performance.

We took advantage of Pytorch tutorial source code for DCGAN model and changed it based on the performance using dog dataset.

For the result, we find that the fake images are having the shape of dog in general with different color, which indicates possibly different breed. However, it does not have a very recognizable dog face. Sometimes dog eyes, nose, and mouth are missing.

2 Related Work

With the rapid evolution of Generative Adversarial Networks (GANs), several notable models have emerged apart from DCGAN, including Conditional Generative Adversarial Nets (CGAN) and Cycle-Consistent Adversarial Networks (CycleGAN).

Conditional Generative Adversarial Nets (CGAN)[2] Conditional Generative Adversarial Nets (CGANs) extend the traditional GAN structure by adding a condition, where both the generator and discriminator are conditioned on additional information such as class labels or data from other

modalities. This conditioning allows CGANs to generate targeted outputs, offering more control over the generation process compared to standard GANs, which generate data from latent space without explicit control over the types of generated content. By incorporating this additional information, CGANs can produce more diverse and relevant outputs, making them particularly useful for tasks requiring specified characteristics, like generating images of a particular class. This flexibility and specificity set CGANs apart, enhancing their applicability across a wider range of generative tasks.

Cycle-Consistent Adversarial Networks (CycleGAN)[3]

Cycle-Consistent Adversarial Networks (CycleGAN) are a transformative advancement in the field of image-to-image translation, enabling the conversion of images from one domain to another without the need for paired examples. Unlike traditional GANs that learn to generate data from a random noise vector, CycleGAN incorporates a cycle consistency loss to ensure that the original image can be reconstructed after being translated to a new domain and back. This innovative approach allows for impressive applications such as style transfer, season transformation, and photo enhancement without direct correspondence between source and target domain images.

3 Method

3.1 Structure of Network

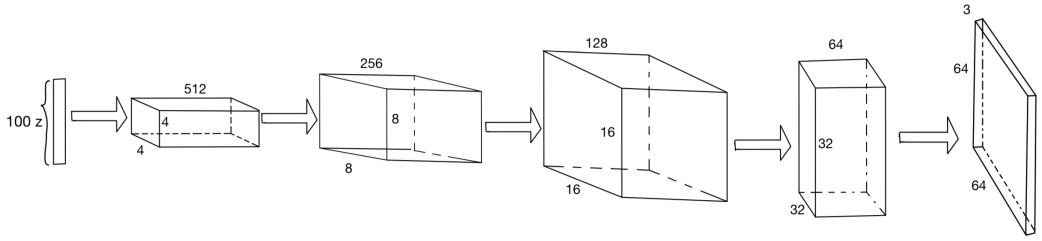


Figure 1: DCGAN Generator Layers Visualization

The first step is to define the structure of the generator. It is designed to map the input noise vector to a data-space representation—in this case, a 3D image tensor representing an image. The network comprises a series of transposed convolutional layers.

- **Input**: The input to the Generator is a noise vector z of dimension zdim, with shape **N x** z_dim **x 1 x 1**, where N is the batch size.
- **First Layer**: A transposed convolutional block that takes the noise vector z as input and transforms it into a 4x4 feature map with $features\_g$ * 8 channels, where $features\_g$ is a base number of features that scales the number of channels in each subsequent layer. The layer uses a stride of 1 and padding of 0, designed to transform the input latent vector z into a spatial format, specifically into a feature map of spatial dimension 4x4. It is then followed by **ReLU** activation.
- **Intermediate Layers**: These layers consist of a series of blocks designed to further upscale the spatial dimensions of the feature maps. Each block:
  - Doubles the height and width of the feature maps (from 4x4 to 8x8, 16x16, and then to 32x32) while halving the number of channels (from $features\_d$ *8 to $features\_d$), maintaining a balance between the spatial resolution and the depth of the feature representation
  - Applies BatchNorm to stabilize training and normalize the inputs of each layer
  - Utilizes ReLU activation
- **Last Layer**: The last transposed convolutional layer converts the feature maps to the desired number of output channels, $channels\_img$, which corresponds to the number of channels in the real images (3 for RGB images in our dataset). This layer uses a kernel size of 4, a

stride of 2, and padding of 1 to upscale the spatial dimensions to the final image size, which is 64x64 pixels.

- **Activation Function**: The ReLU activation function is used in all layers except for the last. The Tanh activation function in the last layer maps the output to the range between -1 and 1.
- **BatchNorm**: It is employed across all network layers except for the last layer, enhancing model stability by normalizing layer inputs.

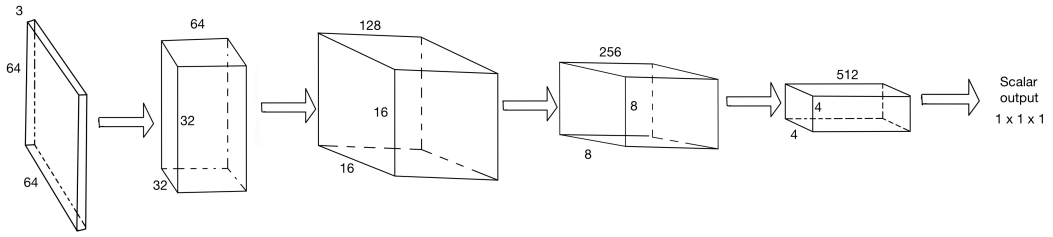


Figure 2: DCGAN Discriminator Layers Visualization

Then, we define the structure of our discriminator, which is pretty similar to the structure of the generator.

- **Input**: The input to the Discriminator is an image as input with dimensions **N x** $channels\_img$ **x 64 x 64**, where **N** is the batch size and $channels\_img$ represents the number of channels in the input images (3 for RGB images in our dataset).
- **First Layer**: A convolutional layer that processes the input image with kernel size of 4, stride of 2, and padding of 1. This layer reduces the spatial dimension by half while increasing the depth to $features\_d$ channels. It is followed by **LeakyReLU** activation function.
- **Intermediate Layers**: These layers consist of a series of blocks designed to further downsample the image while increasing the feature depth. Each block:
  - Doubles the number of channels from its input to output (from $features\_d$ to $features\_d$*8 through successive layers)
  - Reduces the spatial dimensions by half (using stride 2)
  - Applies BatchNorm to stabilize training and normalize the inputs of each layer
  - Utilizes LeakyReLU activation
- **Last Layer**: A convolutional layer that reduces the spatial dimension to 1x1 and outputs a single value per image in the batch. This layer does not use BatchNorm or LeakyReLU but concludes with a Sigmoid activation to output a probability between 0 and 1, indicating whether the input image is real or fake.
- **Activation Function**: The LeakyReLU activation function is used in all layers except for the last, promoting gradient flow during training. The Sigmoid activation in the final layer maps the output to a probability, aiding in the binary classification of images as real or generated.
- **BatchNorm**: It is employed across all network layers except for the first and last layer, enhancing model stability by normalizing layer inputs.

3.2 Train & Test Algorithm

The process to train DCGAN is basically the same as training a Generative Adversarial Network (GAN). It involves training two networks, a generator and a discriminator, where the generator tries to create data that looks like the real data, while the discriminator tries to distinguish between real and generated data.

- **Training Loop**: The outer loop iterates through each epoch and the inner loop iterates over the dataloader, which provides batches of real images.

- **Discriminator Training**: We firstly train the discriminator for each batch. Firstly, we zero out its gradients using $disc.zero\_grad()$. Then we compute the **discriminator's loss for the real images**, encouraging it to output 1s (indicating **real**). After that, we use our generator to generate fake images, which are then used to train the discriminator. It is achieved by computing the **discriminator's loss for these fake images**, encouraging it to output 0s (indicating **fake**). Lastly, we calculate the total discriminator loss as the sum of these two losses, backpropagate it, and update the discriminator's weights.

- **Generator Training**: After training the discriminator, we start the training of generator by zeroing out its gradients, generating fake images, and passing these through the discriminator again as the gradients of discriminator has just been updated. The **generator's loss** is calculated based on the discriminator's predictions for the fake images, with the goal of confusing the discriminator with fake images and having it predict 1s (indicating **real**). Lastly, this loss is backpropagated and the generator's weights are update.

The primary goal of our testing methodology is to evaluate the performance and progression of our generator.

- **Fixed Noise Vector**: We employ a fixed batch of noise vector generated from a standard normal (Gaussian) distribution with a mean of 0 and a standard deviation of 1. The fixed input ensures that we are able to visualize the evolution of generated images and assess qualitative improvements in the generator's output.

- **Loss Function**: We utilize the **Binary Cross Entropy Loss** (BCELoss) as our criterion for both the generator and discriminator. This loss function measures how well the generator can fool the discriminator and how accurately the discriminator can distinguish real images from fake ones.

- **Real and Fake Labels Convention**: For training consistency and clarity in performance evaluation, we establish a convention for labeling real and fake images. Real images are labeled as 1, and fake images generated by the network are labeled as 0.

- **Performance Metrics**: **Discriminator Loss on Real Images** measures how well the discriminator recognizes real images. **Discriminator Loss on Fake Images** measures the discriminator's ability to identify fake images. **Generator Loss** measures how effectively the generator has fooled the discriminator into classifying fake images as real.

- **Visual Evaluation**: At specific intervals (every 500 iterations and at the end of the last epoch), we generate images using the fixed noise vector and the current state of the generator. These images are then saved for later visualization.

3.3 Strengths compared to previous work and Why

Comparing to previous variants of GANs like CGAN and CycleGAN as I mentioned above, DCGAN has the following strengths:

- **Improved Training Stability**: The architectural choices in DCGANs, such as batch normalization and the elimination of fully connected layers, address common training issues in GANs like mode collapse. Also, DCGANs introduces architectural innovations, such as using strided convolutions in the discriminator and fractional-strided convolutions in the generator. Both of these allow DCGANs to generate higher quality images with more detailed textures compared to the other variants of GANs.

- **Broad Applicability**: While CGAN focuses on conditional generation and CycleGAN on domain transfer without paired data, DCGAN serves as a universal model for high-quality image generation across various domains.

4 Experiments & Results

4.1 Dataset

We use the Stanford Dog Dataset that has 20580 images of 120 dog breeds in 120 folders. All images are in ".jpg" format. For each breed, there are about 150 images in the subforlder.

This dataset is particularly well-suited for our task due to its variety in breed-specific features and substantial volume of images, which allows for robust model training and evaluation.

To prepare the dataset for use with our Deep Convolutional Generative Adversarial Network (DCGAN), we utilized PyTorch's DataLoader class. The images have been pre-arranged in folders corresponding to the respective dog breeds. The dataset is processed through a series of transformations to standardize the images. These transformations include:

- **Resizing**: Each image is resized to the image size of 64 x 64, which matches the input size expected by the neural network.
- **Cropping**: After resizing, the images are center-cropped to the target size, ensuring that the central region of the image is used for training the DCGAN.
- **ToTensor Conversion**: The images are converted to PyTorch tensors, enabling them to be processed by the neural network.
- **Normalization**: The image tensors are normalized using a mean and standard deviation of [0.5, 0.5, 0.5] for each color channel, which scales the pixel values to a range of [-1, 1].

We specify our batch size used in training as 256.

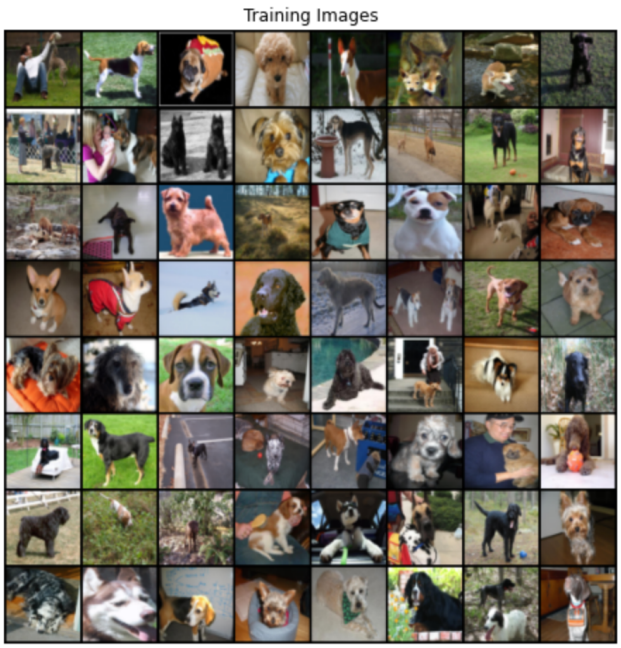


Figure 3: Visualization of 64 Training Images

4.2 Results

During the training loop, we print out loss of discriminator and generator that are denoted as $loss_D = log(D(x)) + (1 - D(G(z)))$ and $loss_G = log(D(G(z)))$

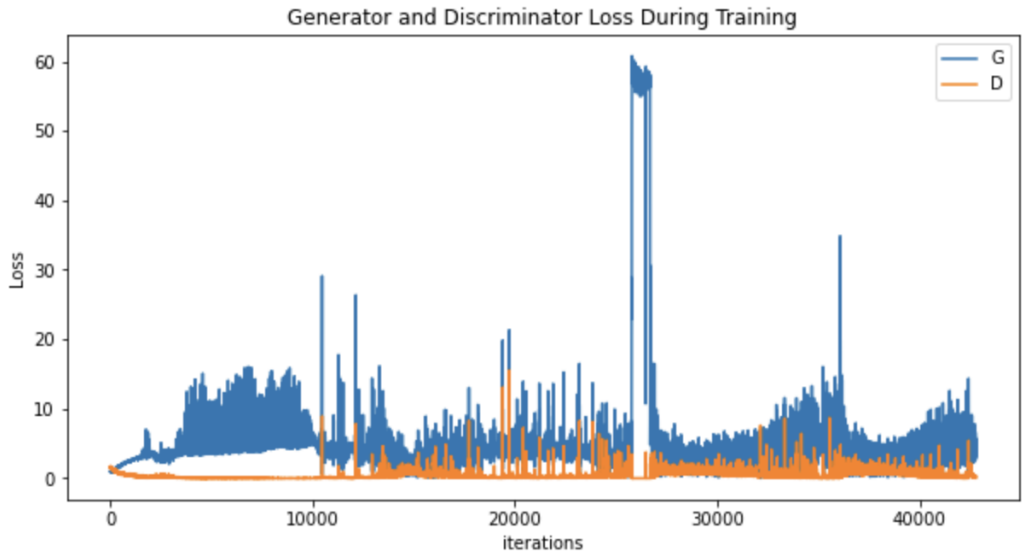
We define $D(x)$ as the average output (across the batch) of the discriminator for the all real batch.

$D(G(z))$ is defined as the average discriminator outputs for the all fake batch.

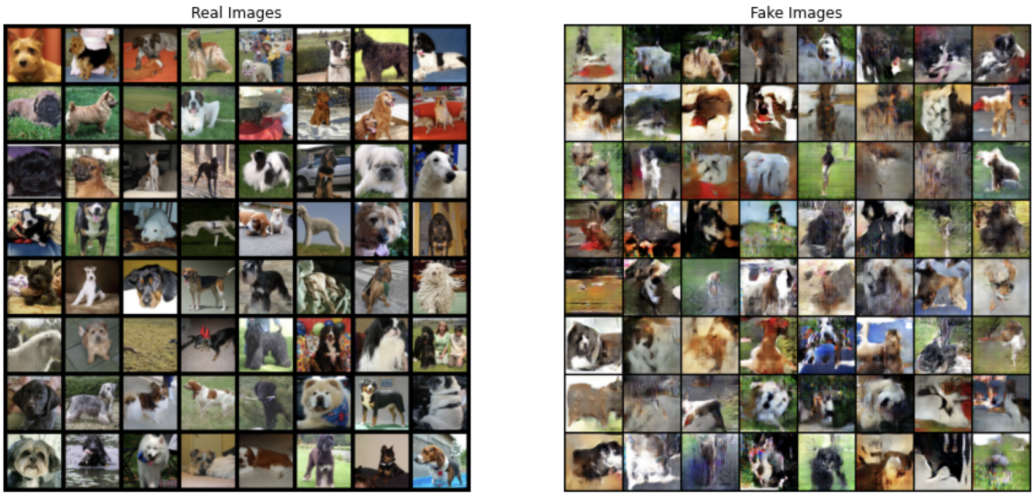
The last epoch gives the following result:

```
[79/80][40/81]  Loss_D: 0.0449  Loss_G: 4.9802  D(x): 0.9918    D(G(z)): 0.0327 / 0.0184
[79/80][80/81]  Loss_D: 0.3036  Loss_G: 3.0305  D(x): 0.7784    D(G(z)): 0.0206 / 0.1191
```

Here is a plot for the generator and discriminator loss over training iterations:



Here is a comparison between a group of real images and fake images:



For the result, we find that the fake images are having the shape of dog in general with different color, which indicates possibly different breed. However, it does not have a very recognizable dog face. Sometimes dog eyes, nose, and mouth are missing.

4.3 Tuned hyperparameters

In order to reimplement dog dataset on the current DCGANs model. We tune hyperparameters $batchsize$ from default 128 to 256. $NumberofEpochs$ is changed from 5 (from original DCGANs paper) to 80. $DepthofFeatureMapInitiallyforGenerator$ is changed from 128 to 64. Original learning rate for both generator and discriminator is 0.0002. $LearningRateforGenerator$ is changed to 0.00025 and $LearningRateforDiscriminator$ is changed to 0.00033.

Summary of changed hyperparameters:
$batchsize$ = 256
$numepochs$ = 80

$depth$ = 64
$Generatorlr$ = 0.00025
$Discriminatorlr$ = 0.00033

Specific process of tuning:

We started off with the original code from the PyTorch DCGAN tutorial, available at PyTorch DCGAN Faces Tutorial.

1. Here we found the depth size is changed to 64 instead of 128 from original paper. After running depth 128 on dog dataset, the result is collapsing and depth 64 works out with vague generated images.

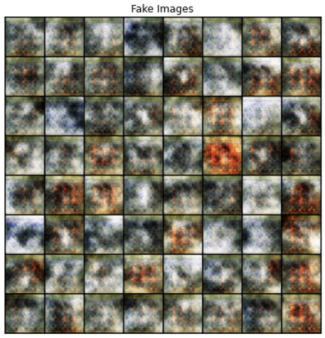
2. We realize our dataset is relatively small comparing to original paper, so $epochs = 5$ is not enough. So we tried $epochs = 50, 80, 100$. Loss has reduced significantly from epochs5 to 50 to 80. From $epochs = 80$ to 100, the loss does not vary much. So we stick with 80 to save computation time.

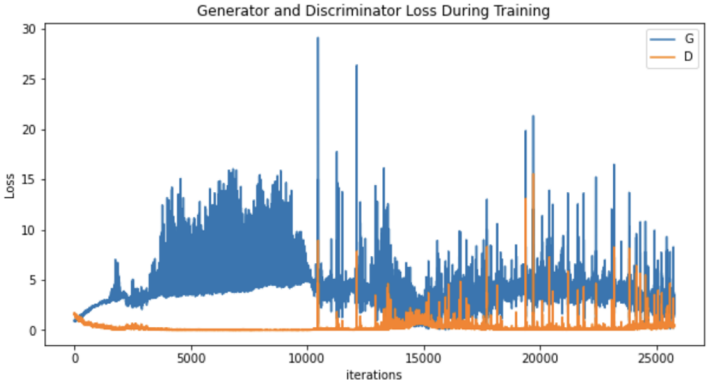
3. The next step, we noticed the slow imbalance loss between generator and discriminator when using $lr = 0.0002$ for both models, while discriminator outcompeted generator. So we make lr for D smaller and G bigger.

4. The next step, we tried to tune $batchsize$. We tried $batchsize$ = 64, 128, 256, 512. Batch size 64 and 512 didn't give a good result from the beginning of training. So we chose $batchsize$ = 256. We also made change to lr as we change batch size.

Moreover, we also tried to use SGD optimizer instead of Adam optimizer in the model with lr 0.00025 and 0.00033 for generator and discriminator respectively. However, the result from plotting the generated images are vague and unrecognizable. So we decide to stick with Adam optimizer.

Here is the result from SGD optimizer with tuned hyperparameters mentioned above:





References

[1] Radford, A., Metz, L., Chintala, S. (2015). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. ArXiv. /abs/1511.06434

[2] Mirza, M., Osindero, S. (2014). Conditional Generative Adversarial Nets. arXiv preprint arXiv:1411.1784. https://doi.org/10.48550/arXiv.1411.1784

[3] Zhu, J.-Y., Park, T., Isola, P., Efros, A. A. (2017). Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. arXiv preprint arXiv:1703.10593. https://doi.org/10.48550/arXiv.1703.10593