Deep Convolutional Generative Adversarial Networks

Po Hsiang Huang
UCSD ECE Department
A14578741

Arden Ma
UCSD ECE Department
A15232741

Abstract

This project seeks to investigate the performance of Deep Convolutional Generative Adversarial Network (DCGAN) introduced by Radford et al. [2016] for the task for generating natural images. We created our own implementation of DCGAN to artificially generate natural images that closely mimic the images found in the training set. We tried modifying our model architecture and hyperparameters to generate the best images. The results are attached and explained in this paper.

1 Introduction

The specific problem that we want to tackle is the challenge of developing a model which learns to artificially produce natural images mimicking the images that they were trained on. This problem of image generation is actually a specific case of a more general problem which can be posed as follows: suppose we have some distribution of data $P(y)$, we want to learn a parametric model $f(x;\theta)$ which approximately maps samples of random data to this target distribution $P(y)$. In our case, we will be utilizing the DCGAN architecture as our parametric model, and we hope to train our model such that it learns to produce image samples belonging to the target distribution of images that we trained the model on.

This ability for a model to produce lifelike images is an exciting feat on its own, and additionally, such models lie at the backbone of many interesting (and sometimes controversial) applications in the world today, with deepfakes being one of the most talked about use cases for such models. In addition, in the machine learning world, the ability to create new images belonging to some target image distribution would serve as a valuable method for augmenting datasets, which could help improve performance of many models.

2 Related Work

This problem of generating natural images is a widely studied problem with many interesting applications. Some common techniques used to solve this problem include learning a latent distribution over the desired image distribution that can be stochastically sampled to generate images, which is done by the popular Variational Autoencoder (VAE) (Kingma and Welling [2014]) model, as well as various autoregressive modeling techniques such as the one employed by Pixel CNN (van den Oord et al. [2016]).

Another common technique, which serves as the foundation for our method, is adversarial training. Adversarial techniques have played a big role in propelling research in this field forward, most notably starting with the introduction of General Adversarial Networks by Goodfellow et al. [2014]. These adversarial models pit two competing models (typically neural networks) against each other in a two-person minimax game, where one model tries to produce data from a target distribution and the other model tries to tell whether or not a sample of data is from the generative model or from the actual data distribution. These ideas will be the focus of our investigation and will be explored further in the following sections.

Some of the most exciting recent work in natural image generation has come out of OpenAI who recently released code and a paper for their DALL-E (Ramesh et al. [2021]) model, which adopts a likelihood maximization approach rather than the adversarial approaches that are currently very popular. The authors of DALL-E try to solve a slightly different task than what DCGAN attempts to solve: the task of generating natural images from a user inputted caption. To do this they use a sized down GPT-3 (Brown et al. [2020]) model along with a VQ-VAE (van den Oord et al. [2018]) autoencoder to autoregressively model this generative task.

3 Method

Our method will mainly revolve around the DCGAN (Radford et al. [2016]) architecture, an extension of General Adversarial Networks (GANs) proposed by Goodfellow et al. [2014] for image data. Specifically, DCGAN builds on top of this already successful generative model framework while incorporating elements from Convolutional Neural Networks (CNNs) which have been effective in boosting Neural Network performance on image tasks. The results in Radford et al. [2016] shows that DCGAN produces a more stable output than GAN and achieves state-of-the-art performance on multiple datasets.

From GANs, DCGAN derives its overall model and training formulation of a adversarial "game" played between a Discriminator Network and Generator Network. Essentially we take turns training one network and freezing the weights of the other network, so that the generator learns to produce "natural" images from the training distribution such that the discriminator learns to tell whether a sample is from the training distribution or was created by the generator which was used to attempt to fool the discriminator. The exact formulation of this adversarial objective as a minimax game is given in Goodfellow et al. [2014].

Then from CNNs, DCGAN takes the idea of transposed convolutions, which are often used in neural networks for semantic segmentation. In DCGAN, transposed convolutions are used to upsample the feature representations/maps of the generated image, to go from random noise to a full image. These transposed convolutions were inspired by the convolutional layers made famous by CNNs like AlexNet (Krizhevsky et al. [2012]), ResNet (He et al. [2016]), and VGG (Simonyan and Zisserman [2015]).

The major strength of the proposed method lies in taking a very powerful adversarial learning framework from Goodfellow et al. [2014], which has been shown to create powerful generative models, and combining it with elements like the transposed convolution, which have shown to be highly effective when used in neural networks for image-tasks such as semantic segmentation. The robustness and effectiveness of this approach can be seen through the relatively better result produced by DCGAN in Figure 1:

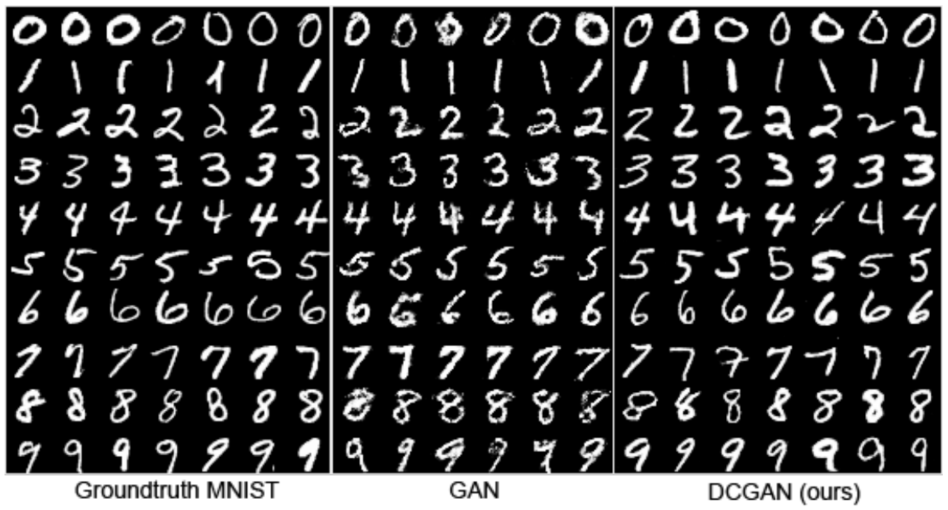


Figure 1: MNIST dataset generated result from baseline GAN and DCGAN (taken from the original paper). (Radford et al. [2016])

4 Experiments

For our experiments, we tried to replicate, as best as possible, the generator and discriminator network architectures given in Radford et al. [2016] as seen in Figure 2 while also following the provided guidelines in Figure 3. To train the generator and discriminator networks, we used the adversarial training algorithm from Goodfellow et al. [2014]. Steps in our experiment inlcude:

- Implement a detailed reconstruction of the DCGAN architecture to replicate the capabilities in Radford et al. [2016] and understand the reasoning behind this specific architecture.
- Experiment with simplifying or modifying the architecture by tuning hyperparameters such as the neural network layer depth and order, training loops, and epochs to try and achieve an even better performance on our datasets.
- Perform empirical analysis on the generative ability of the generator network by visually comparing the generated images to the training set images.
- Visualize some of the convolutional kernels in the discriminator network.

4.1 Dataset

This project uses .jpg images from CelebA Dataset (Li [2018], Liu et al. [2015]) to train and test the algorithm. The CelebFaces Attributes (CelebA) Dataset (Li [2018], Liu et al. [2015]) contains 202,599 celebrity face images. Initially we planned on using the version of this dataset that was available on Kaggle, however it turns out that this dataset is available from the torchvision datasets library, so we opted to use that version to simplify the data loading process. The images are already normalized since we are using the images provided by torchvision, but they are of different sizes, which makes them incompatible with our network architecture, as our discriminator expects a certain size image input. To address this,we apply various transforms, again using torchvision, such as center cropping and resizing using bilinear interpolation to make all the input images the same size.

4.2 Model Architecture

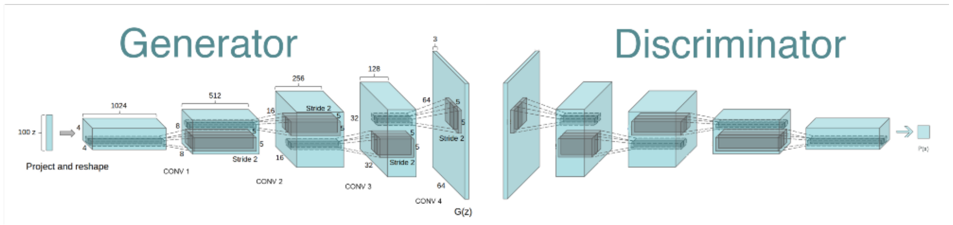
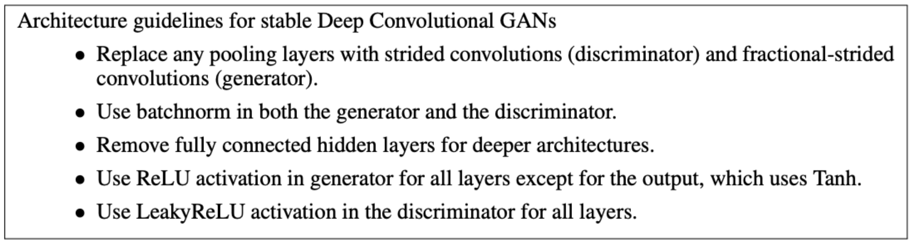


Figure 2: DCGAN architecture (Radford et al. [2016])



Architecture guidelines for stable Deep Convolutional GANs

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.

Figure 3: DCGAN model guidelines (Radford et al. [2016])

DCGAN is an extension of the popular GAN model, and consists of a generator and discriminator network. GANs (including DCGAN) are inherently unstable and hard to train because of the dynamic nature of their training routines, which consists of training two competing models. One of the main contributions of DCGAN is that it introduces a more stable training architecture resulting from the points summarized in Figure 3. We followed the guidelines when building our own implementation of DCGAN and experimented with different architecture to produce the best images.

4.2.1 Generator

We started off building our generator with 1 linear layer (100->1024) and 4 transposed convolution layers. We attempted to use (5×5) fractional-strided convolution layers as suggested by Figure 3, but failed to match the desire input shapes. The formula to calculate the size of the convolution output layer is $\frac{InputSize - KernelSize + 2 \times Padding}{Stride} + 1$, which seemed impossible to calculate the desired output shape, so we changed all convolution layers to be to (4×4) with stride of 2 and padding of 1.

Follow the guideline in Figure 3, we used ReLU for after each convolution layers except for the last, where tanh is applied. We also used batchnorm in each convolution block, but whether to apply batchnorm before or after the activation function has been a controversial topic and we decided to follow the implementation used by the original authors of the batchnorm paper (Ioffe and Szegedy [2015]) who chose to apply batchnorm after ReLU.

Initially our generator had trouble learning how to produce images because the discriminator was converging too quickly or converging to 0 within 1 epoch, diluting the training signal received by the generator. An example of this phenomenon can be seen in Figure 5, where the training loss for the generator starts off large, lowers a bit, and remains constant within 500 iterations, meaning that it stops learning very fast.

The Pytorch code depicting our final model architecture for the generator can be seen in Figure 4.

```
class Generator(nn.Module):
    def __init__(self, depth=128):
        super(Generator, self).__init__()
        self.conv1 = nn.ConvTranspose2d(100, depth*8, kernel_size=4, stride=1, padding=0, bias=False)
        self.bn1 = nn.BatchNorm2d(depth*8)

        self.conv2 = nn.ConvTranspose2d(depth*8, depth*4, kernel_size=4, stride=2, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(depth*4)

        self.conv3 = nn.ConvTranspose2d(depth*4, depth*2, kernel_size=4, stride=2, padding=1, bias=False)
        self.bn3 = nn.BatchNorm2d(depth*2)

        self.conv4 = nn.ConvTranspose2d(depth*2, depth, kernel_size=4, stride=2, padding=1, bias=False)
        self.bn4 = nn.BatchNorm2d(depth)

        self.conv5 = nn.ConvTranspose2d(depth, 3, kernel_size=4, stride=2, padding=1, bias=False)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.bn1(x)
        x = self.relu(self.conv2(x))
        x = self.bn2(x)
        x = self.relu(self.conv3(x))
        x = self.bn3(x)
        x = self.relu(self.conv4(x))
        x = self.bn4(x)
        x = self.tanh(self.conv5(x))
        return x
```

Figure 4: Our Pytorch code for the generator.

4.2.2 Discriminator

This discriminator essentially has the reverse structure of the generator, exchanging transposed convolutional layers for convolutional layers and ending with a linear layer. The linear layer is followed by a sigmoid activation function to determine the probability of the image being fake or real. We initially used a hand implementation of the log loss function in Goodfellow et al. [2014], however this led to some numerical instability, yielding -inf values. To address this we first tried clamping the input values to the log function using some epsilon so that the input to log would never be 0. Additionally, in the original GAN paper (Goodfellow et al. [2014]), the training algorithm suggests to update the discriminator weights k times for every update of the generator weight. We initially tried experimenting with this, and saw some success with k=1, and k=2, however, there were no visible differences in their performances so we ended up using k=1 for all further trainings to minimize the amount of computation. Overall the largest challenge with designing the discriminator was to make sure that it was strong enough to help the generator learn to produce good images, but not too strong to deplete the training signal to the generator.

Below are some of the additional changes we tried:

- We experimented with various initialization procedures but ended up initializing our weights using a zero-centered Gaussian distribution with a standard deviation of 0.2 as used by the authors of the DCGAN paper (Radford et al. [2016]). We also used this method to initialize the generator weights.
- We saw that the discriminator got stuck with high loss, meaning it didn't have enough capacity to discriminate between the training data and generated data, so we added an extra convolutional layer to get 5 convolution layers total.
- Then we found that the discriminator was converging to 0 far too quickly so we tried halving the number of convolutional filters (e.g. in the first convolutional layer we went from 128 output channels to 64) so it trained faster and converges later, but still saw poor results such as in Figure 5.
- We also tried adding Dropout layers to our discriminator to regularize it, but eventually found that we achieved better results without them.
- One of the improvements that we achieved came from removing the first batchnorm layer as suggested by the PyTorch DCGAN tutorial. This improved the convergence issue, but the discriminator was still converging faster than desired, hindering the generator's ability to learn.
- We learned from LSGAN (Mao et al. [2016]) that having a sigmoid layer could introduce the problem of vanishing gradient. We suspected that our convergence issue could be a result of this. In an attempt to improve upon the clamping method used to address the numerical instability in the hand implemented log loss function, we opted to remove the sigmoid layer and use the Binary Cross Entropy with logits loss provided by Pytorch instead. This significantly improved the training process.

The Pytorch code depicting our final model architecture for the discriminator can be seen in Figure 6.

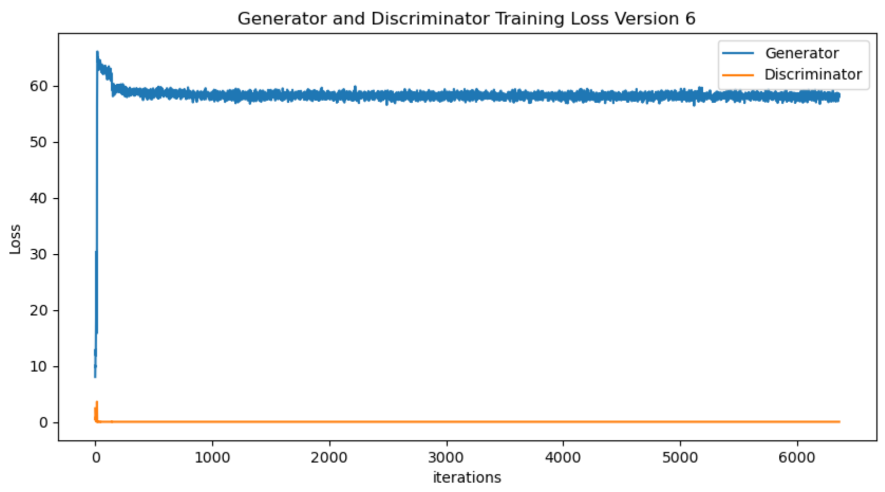
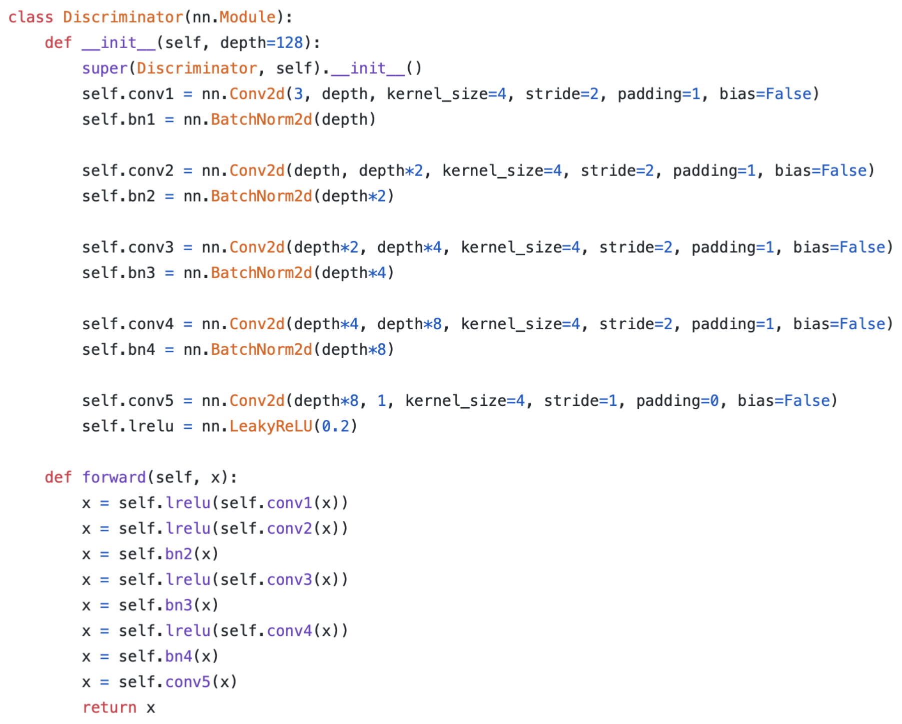


Figure 5: Example of our collapsed model with discriminator converging too soon.



```python
class Discriminator(nn.Module):
    def __init__(self, depth=128):
        super(Discriminator, self).__init__()
        self.conv1 = nn.Conv2d(3, depth, kernel_size=4, stride=2, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(depth)

        self.conv2 = nn.Conv2d(depth, depth*2, kernel_size=4, stride=2, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(depth*2)

        self.conv3 = nn.Conv2d(depth*2, depth*4, kernel_size=4, stride=2, padding=1, bias=False)
        self.bn3 = nn.BatchNorm2d(depth*4)

        self.conv4 = nn.Conv2d(depth*4, depth*8, kernel_size=4, stride=2, padding=1, bias=False)
        self.bn4 = nn.BatchNorm2d(depth*8)

        self.conv5 = nn.Conv2d(depth*8, 1, kernel_size=4, stride=1, padding=0, bias=False)
        self.lrelu = nn.LeakyReLU(0.2)

    def forward(self, x):
        x = self.lrelu(self.conv1(x))
        x = self.lrelu(self.conv2(x))
        x = self.bn2(x)
        x = self.lrelu(self.conv3(x))
        x = self.bn3(x)
        x = self.lrelu(self.conv4(x))
        x = self.bn4(x)
        x = self.conv5(x)
        return x
```

Figure 6: Our Pytorch code for the discriminator.

4.2.3 Optimizer

For the optimizer, we initially tried using SGD, as used by Goodfellow et al. [2014] but ultimately we chose to optimize both models using the Adam optimizer following Radford et al. [2016]. We used learning rate of 0.002, β_1 of 0.5, and β_2 of 0.999 for both the generator and discriminator.

4.3 Results

After much hyperparameter tuning and a combination of the changes listed in the previous sections, the final training loss for our model is shown in Figure 7. Additionally we show in Figures 8,10, and 12 the images generated by our generator network after 1, 10, and 20 epochs respectively. We also show visualizations of the features from the first convolutional layer of our discriminator network in Figures 9,11, and 13 after 1, 10, and 20 epochs respectively.

We notice that within 1 epoch of training, our implementation of DCGAN is already able to create outlines and general features of human faces from random noise. By 10 epochs of training, our model improved significantly and is able to produce some very realistic human faces, while some still exhibit quite a bit of distortion. Notably, as we continue to train for another 10 epochs, we do not observe visible improvements, meaning that the generator is learning much less and discriminator can no longer effectively distinguish the generated results from actual images after the first 10 or so epochs.

5 Conclusion

5.1 Takeaways

We're very happy with how our model turned out, given this was our first attempt at creating a GAN-like model, and we were surprised with the quality of images we were able to generate. Probably the biggest takeaway from this project for us was learning to tune our model architecture and hyperparameter choices in order to achieve stable training. In the "normal" supervised settings that we typically deal with, optimization is typically much more straightforward, so learning to optimize

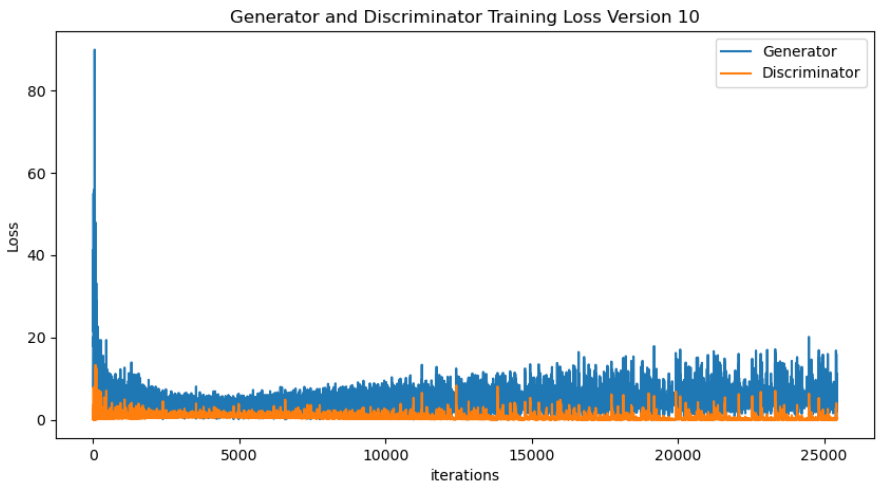


Figure 7: CelebA final training loss for 20 epochs.

models in this adversarial setting was a really important learning opportunity for us. Additionally, this project gave us great exposure to exciting new ideas regarding adversarial training, as well as additional experience working with Pytorch.

5.2 Future works

We hope in the future to extend this work to more datasets, tune our hyperparameters further to generate better images, and also see how DCGAN and related architectures can be used in other settings to tackle other exciting problems.

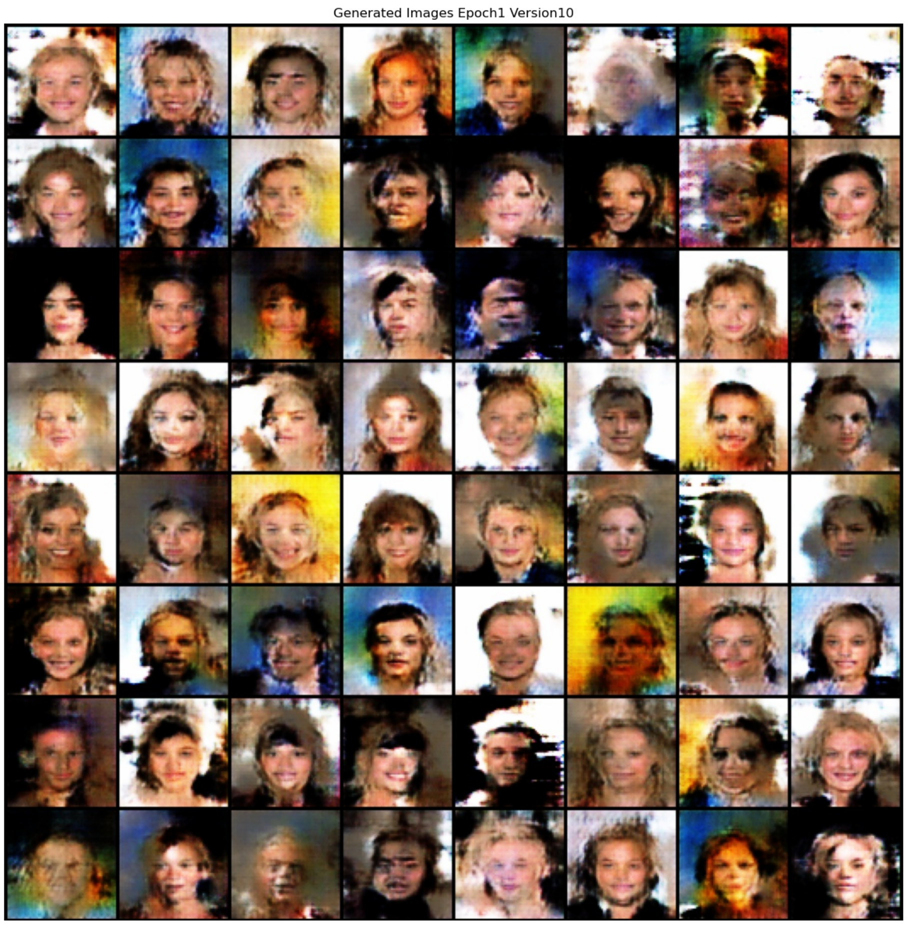


Figure 8: Generated CelebA images after 1 epoch of training.

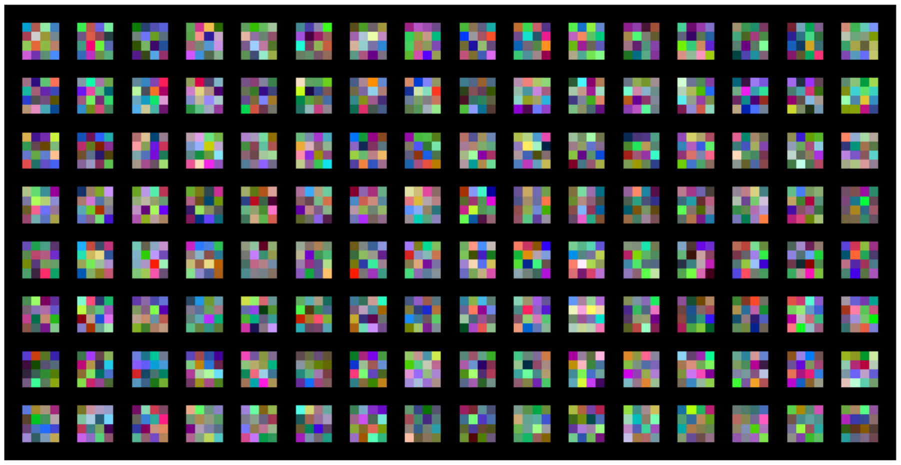


Figure 9: First convolutional layer kernel visualizations from discriminator network after 1 epoch of training.

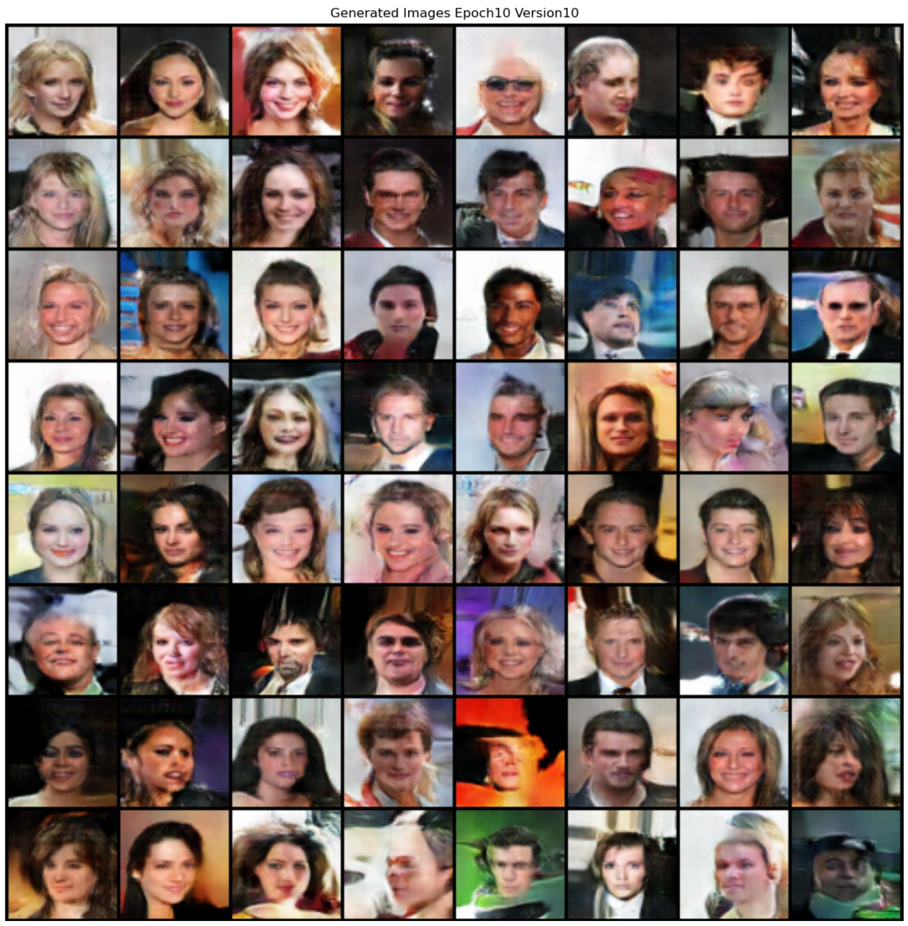


Figure 10: Generated CelebA images after 10 epochs of training.

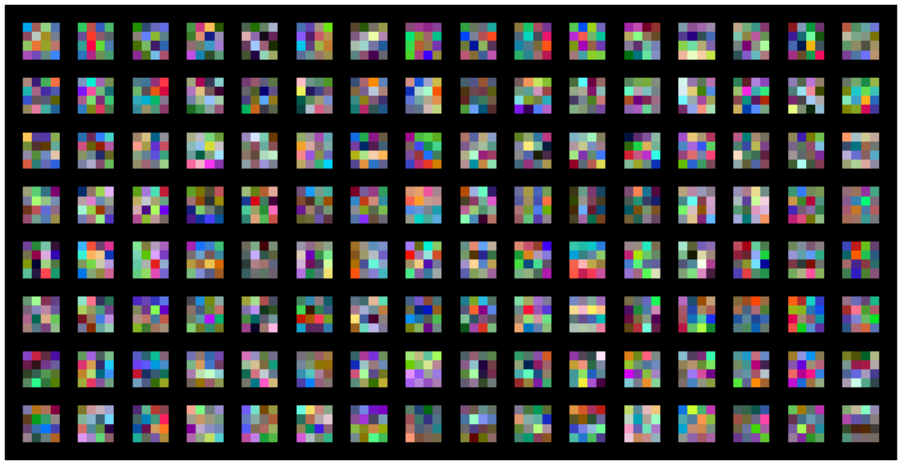


Figure 11: First convolutional layer kernel visualizations from discriminator network after 10 epochs of training.

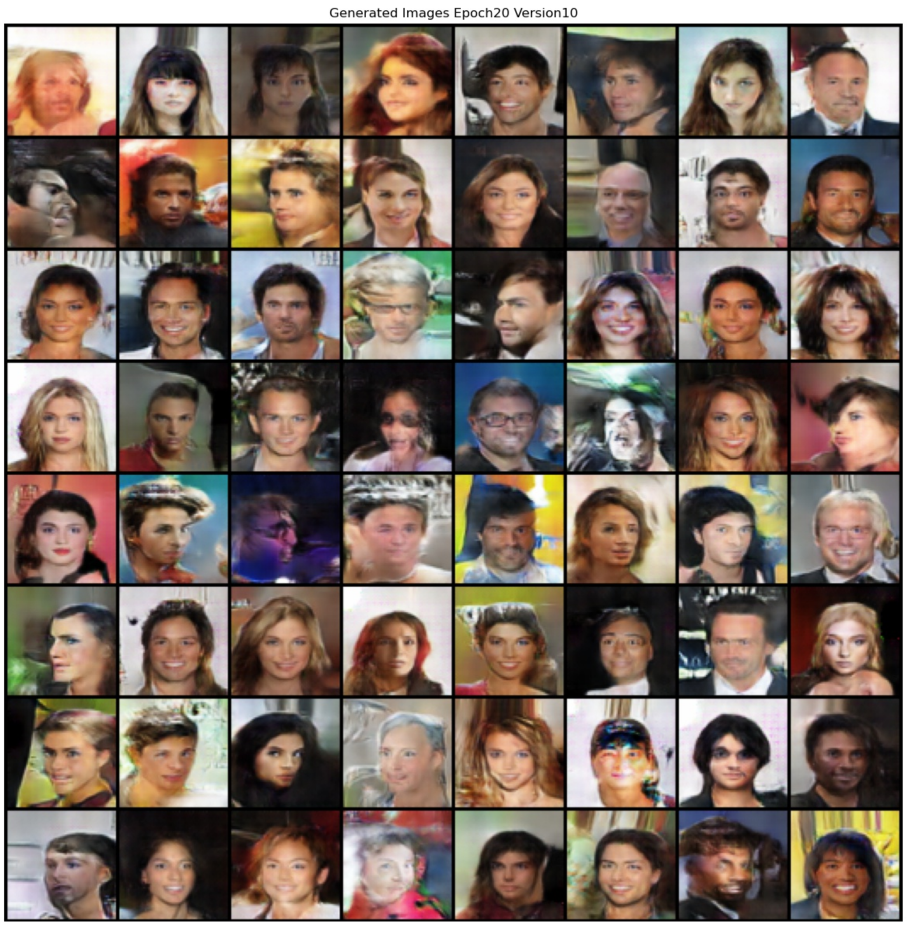


Figure 12: Generated CelebA images after 20 epochs of training.

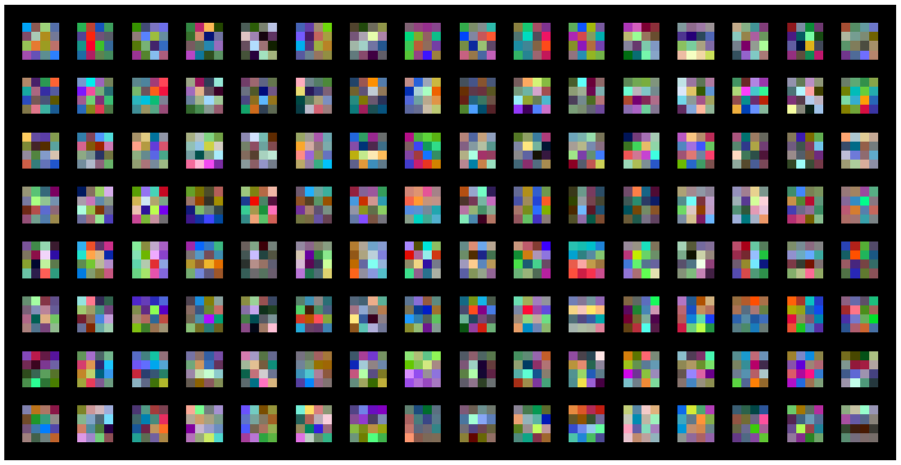


Figure 13: First convolutional layer kernel visualizations from discriminator network after 20 epochs of training.

References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR. URL `http://proceedings.mlr.press/v37/ioffe15.html`.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2014.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.

Jessica Li. Celebfaces attributes (celeba) dataset. In *Kaggle*, June 2018.

Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

Xudong Mao, Qing Li, Haoran Xie, Raymond Y. K. Lau, and Zhen Wang. Multi-class generative adversarial networks with the L2 loss function. *CoRR*, abs/1611.04076, 2016. URL `http://arxiv.org/abs/1611.04076`.

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2016.

Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation, 2021.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with pixelcnn decoders, 2016.

Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning, 2018.