Generating Photorealistic Images of Dogs using Generative Adversarial Networks (GANs)

George Washington University

Columbian College of Arts & Sciences

DATS 6303 Deep Learning

Final Project - Group 7

Members: Cody Yu, Kismat Khatri, Jeffrey Hu, Ei Tanaka

Date: December 8, 2023

Table of Contents

Introduction

Our project aims to generate realistic images of dogs using advanced AI techniques known as Generative Adversarial Networks (GANs). The fascinating diversity and complexity of dog breeds present a unique challenge for synthetic image generation, making this project not only innovative but also a significant step forward in AI-generated art and data enhancement for machine learning. We utilize Deep Convolutional Generative Adversarial Networks (DCGANs) and UNIT 2d Diffusers to create square images of various dog breeds. Our primary goal is to improve the realism of these generated images. To achieve this, we experiment with different loss functions, adjust hyper-parameters, and add more convolutional layers to our models.

Background of GAN

Generative Adversarial Networks (GAN) represent a breakthrough in deep learning, initially introduced by Ian Goodfellow in 2014. Yann LeCun, a prominent figure in the field of artificial intelligence, praised GAN in 2016 as a remarkably innovative concept in machine learning over the past two decades. This technology is relatively new and has shown significant growth, particularly in 2018, with rapid advancements. Although still in its early stages, various models have emerged, featuring the GAN suffix, like Conditional GAN, DCGAN, Cycle GAN, and Stack GAN. The good news is that it's possible to keep pace with GAN's evolution.

Motivation

Our inspiration came from observing how GANs could create lifelike celebrity images. Intrigued by the potential of GANs in other areas, we explored Kaggle and found a competition titled "Generative Dog Images." Although the competition had ended, we decided to use the associated Stanford Dogs Dataset for our project. This challenge provided us with an excellent opportunity to deepen our understanding of GANs and their applications. Plus, creating dog images added an enjoyable element to our research.

Dataset

Training a Generative Adversarial Network (GAN) effectively necessitates a substantial dataset. To circumvent the laborious task of assembling a well-balanced collection of dog images ourselves, we opted for the Stanford Dogs Dataset. This publicly accessible dataset consists of 20,580 annotated images spanning over 120 dog breeds.

Despite its comprehensive classification, we encountered notable limitations in this dataset. A significant inconsistency was observed in the portrayal of dogs across the images. In some cases, dogs were prominently placed at the center, occupying a large part of the frame. In contrast, other images featured dogs positioned to the side or appearing distant.

This lack of uniformity, coupled with the variable and often noisy backgrounds - characterized by stark color differences even within the same breed - presented a substantial challenge. These factors contributed to a considerable degree of intra-class variation, complicating the task for the GAN. Specifically, it became more difficult for the generator component of the GAN to focus and accurately capture the specific features of the dog breeds that we aimed to highlight.

Overview of the DataSet

Our dataset contains 20579 images belonging to 120 dog breeds from all over the world. The number of each breeds' images mainly focuses on the range from 150 to 250.

	Dog Breed	Number of Observations
0	n02085620-Chihuahua	152
1	n02085782-Japanese_spaniel	185
2	n02085936-Maltese_dog	252
3	n02086079-Pekinese	149
4	n02086240-Shih-Tzu	214
.	.	.
.	.	.
.	.	.
116	n02113978-Mexican_hairless	155
117	n02115641-dingo	156
118	n02115913-dhole	150
119	n02116738-African_hunting_dog	169

Fig-1: Dog Images Description

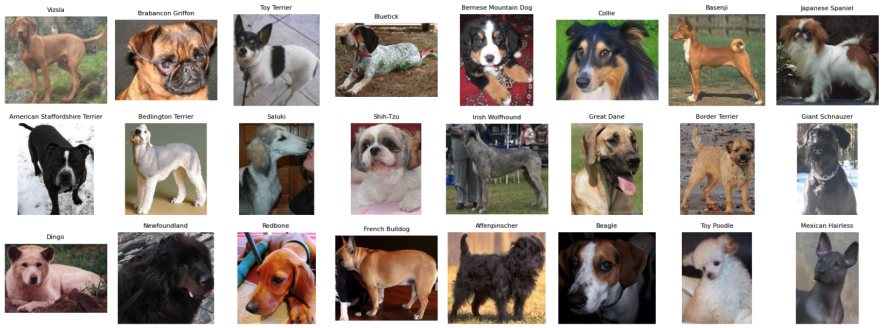


Fig-2: Dog Images

By checking the real Dog Images, their sizes vary and the presence of dogs is inconsistent. Their complex background would be distracting elements for our research, since some images are dominated by complex backgrounds rather than the dogs themes. Our goal is to create images predominantly featuring dogs. How to identify the dog features and eliminate the background influence would be the main challenge for us.

EDA(Exploratory Descriptive Analysis)

In the EDA section, we analyzed the distribution of images across different breeds to understand the dataset's composition better. Additionally, we explored the impact of various image augmentation techniques to enhance our dataset's diversity and robustness. This involved manipulating images in different ways, such as resizing, rotating, or altering brightness, to create a more comprehensive training set for our model. We also investigated the color distribution within the RGB channels to gain insights into the color patterns prevalent in the dataset.

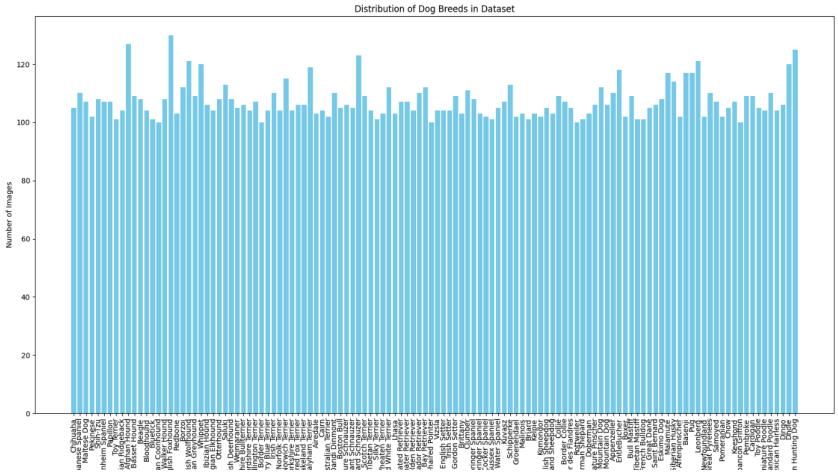


Fig-3: Dog Images Distribution

Fig-4: Augmented Dog Images

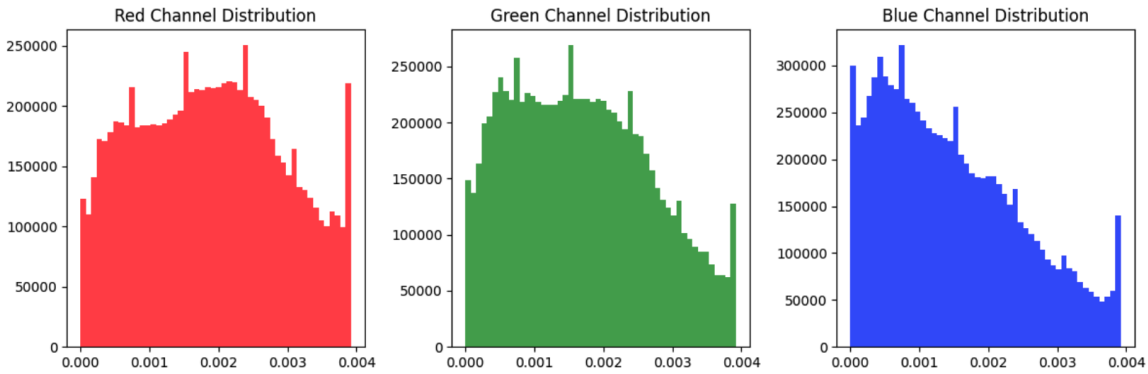


Fig-5: RGB Channels

Deep learning network and training algorithm

We used two models for this project: DCGAN (Deep Convolutional Generative Adversarial Network) and UNet2D Diffuser. In this section, we provide some background information about these models.

DCGAN

The Deep Convolutional Generative Adversarial Network (DCGAN) represents a substantial advancement in deep learning, specifically in generative models. Developed as an extension of the original Generative Adversarial Networks (GANs) introduced by Ian Goodfellow and colleagues in 2014. DCGANs incorporate the principles of convolutional neural networks (CNNs) into the GAN architecture, making them particularly adept for tasks related to image data. A notable innovation in DCGANs is convolutional layers in both the generator and discriminator, enhancing their suitability for image-related tasks. Additionally, DCGANs address the challenges of training traditional GANs, which were prone to model collapse, leading to limited diversity in the generated samples. DCGANs introduce several architectural constraints to overcome these challenges, resulting in more stable training and higher-quality outputs. Furthermore, DCGANs have demonstrated their ability in unsupervised feature learning, making them valuable in applications such as image recognition and classification.

In terms of architecture, DCGANs make essential modifications: they employ stridden convolutions in the discriminator to reduce spatial dimensions of images and use fractionally-stridden convolutions, or deconvolutions, in the generator for upscaling the input noise into a full-sized image. Both the generator and discriminator incorporate batch normalization, stabilizing the training process. Leaky ReLU activation functions, particularly in the discriminator, prevent the dying ReLU problem and promote gradient flow during training.

The training algorithm of DCGANs also deviates from traditional GANs. Training alternates between updating the discriminator, which learns to differentiate between authentic and generated images, and the generator, trained to produce images increasingly indistinguishable from actual images. Commonly, DCGANs employ a binary cross-entropy loss function to measure the discrepancy between the discriminator's predictions and the actual labels of images, whether real or fake. The discriminator (D) and generator (G) loss functions can be represented mathematically, showing the intricate relationship between these two components in the learning process. A typical DCGAN architecture comprises multiple layers of convolution for the discriminator and deconvolution for the generator, underlining the complexity and sophistication of this model in handling image data.

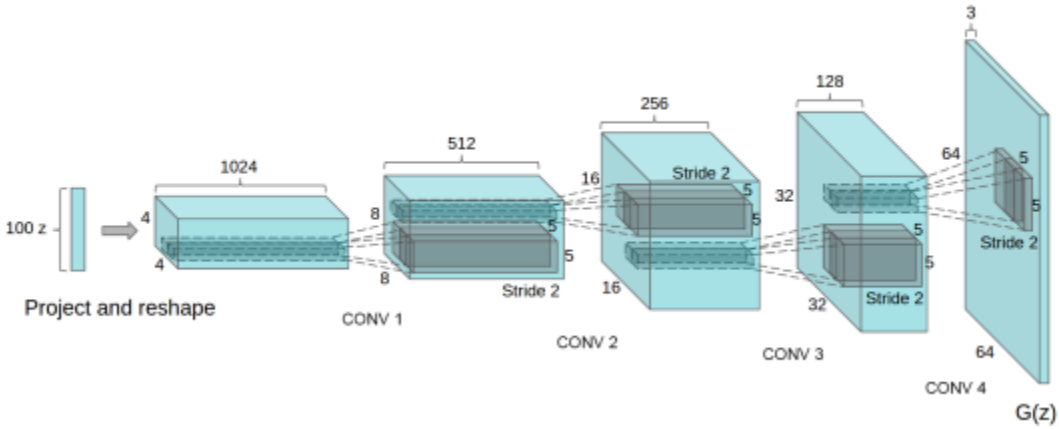


Fig-6: DCGAN Generator Architecture from Radford, A., Metz, L., & Chintala, S. (2015). Figure 1. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks." Retrieved from https://arxiv.org/pdf/1511.06434v2.pdf.

Here are some guidelines when we construct a DCGAN model:

- Instead of using max-pooling layers, we implemented convolution layers that have a stride setting. This change can allow for a more controlled downscaling process.
- Implement transposed convolution for upsampling. Unlike the regular convolution process, which reduces the input size, transposed convolution effectively increases the size, making it suitable for upsampling tasks.
- Incorporate Batch Normalization (BN) in all layers except the output layer of the Generator and the input layer of the Discriminator.
    - Batch Normalization can mitigate the issue of mode collapse and facilitate the development of deeper networks.
    - It helps in overcoming challenges related to poor parameter initialization.
    - Note that while Batch Normalization can lead to the creation of sharper images during training, this only holds true if other tuning steps are correctly implemented. Otherwise, it might result in suboptimal outcomes.

- Opt for LeakyReLU activation over ReLU, except for the output layer, which should use tanh for the Generator and sigmoid for the Discriminator.
    - LeakyReLU, an improved version of the ReLU activation function, allows a small, non-zero gradient when the input is negative, thereby enabling the backpropagation process through the network even for negative input values.
    - This activation choice ensures that the Discriminator backpropagates stronger gradients, including negative gradients, to the Generator, enhancing the learning process.

Final DCGANS Architecture

Generator	Discriminator
Input Noise Vector $z \in R^{100}$	Input RGB Image $\in R^{3*64*64}$
ConvTranspose2D, 64*8, stride=1 + BN + ReLU	Conv2D, 64 + LReLU
ConvTranspose2D, 64*4, stride=2 +BN + ReLU	Conv2D, 64*2 + BN + LReLU
ConvTranspose2D, 64*2, stride=2 + BN + ReLU	Conv2D, 64*4 + BN + LReLU
ConvTranspose2D, 64, stride=2 + BN + ReLU	Conv2D, 64*8 + BN + LReLU
ConvTranspose2D, 3 + Tanh	Conv2D, 1 + Sigmoid
Output Fake Images $\in R^{3*64*64}$	Output Probability $\in R^{1}$

UNet2D Diffuser

Diffusion Model

The first diffusion model was invented by Jascha Sohl-Dickstein, in 2015 at Stanford. He mentions the term “Diffusion probabilistic models”, noting that the probabilistic models of the time had “two conflicting objectives: tractability and flexibility”. His procedure, inspired by non-equilibrium statistical physics, was to use a Markov chain to, step-by-step, convert one

sequence slowly to another, or in the case of images, add the pixels to an image until the structure becomes completely “destroyed”, before starting the diffusion process to rebuild the lost image by, again, adding pixels over set time steps. While GANs had been introduced at that point (the idea of training inference and generative models against each other was published in 1995), Sohl-Dickstein presented the Diffusion Probabilistic models as an alternate window to the problem of generating distributions. In 2020, the Denoising Diffusion Probabilistic Models paper was released. It introduced the DDPM scheduler that would incrementally add noise to an image given n time steps, and worked with denoising autoencoders. The paper also noted that, on the 256x256 LSUN, CelebA-HQ, and CIFAR10 datasets, the diffusion model produced images that were similar or better in quality to the ProGAN, or Progressive Growing GAN, which was implemented with ways to mitigate “unhealthy competition” between the generator and discriminator of traditional GANs.

Fig-6-2: Diffusion Generation on CIFAR10, (from Ho, J., Jain, A., & Abbeel, P. (2020). Denoising diffusion probabilistic models)

UNet Architecture (Encoder-Decoder)

The UNet CNN architecture, developed by Ronneberger et al. in the paper “U-Net: Convolutional Networks for Biomedical Image Segmentation” in 2015, was first used to process biomedical images for image segmentation, often used with data like pixelated slices of diseases for identification and to distinguish borders around each pixel. The symmetric “U” shape of the architecture contains the contracting or encoding path on the left side, fitted with downsampling blocks and max pooling layers, and the right or expanding side with upsampling blocks and deconvolutional layers (or transposed convolutional layers). This symmetric architecture not only allows for segmentation maps that are the same size as the original image tile, which was important for allowing both of the images to line up to correctly identify and outline the afflicting disease, but also allowed for sequence-to-sequence problem solving, and in the case of UNet2D, the ability to generate a 2D diffused image given training set. By taking noisy images are random timesteps, UNet2D can learn to diffuse at each step of the image generation process.

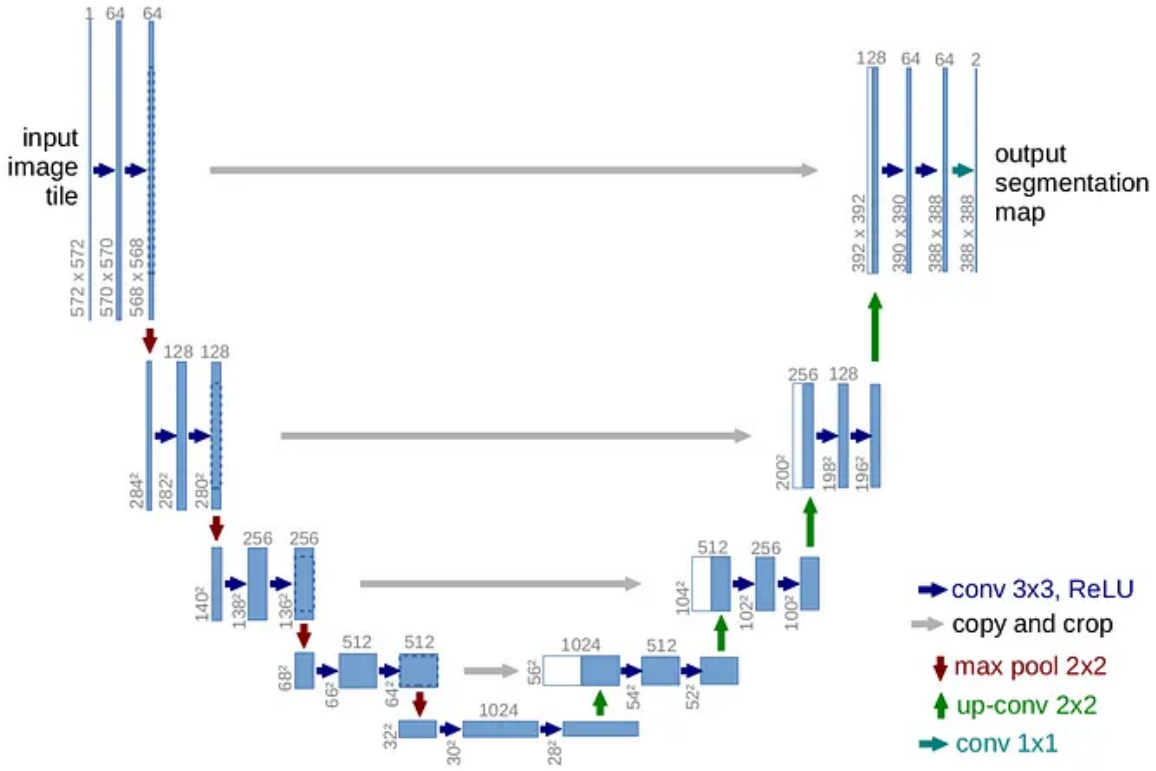


Fig-6-3: UNet2D Diffusion Model Architecture

In the UNet2D model used for this diffusion process, two ResNet layers were passed alongside a ResNet downsampling block, five times to form five UNet blocks, along with a ResNet Attention block for spatial self-attention (for image data, pixel attention scores or how they contribute to other pixels). The upsampling blocks mirrored the downsampling blocks, thus shaping the “U”.

Experimental Setup

DCGAN

Discriminator

The purpose of the discriminator is to distinguish all fake images. A discriminator works like a classifier with the sole task of determining if an image is real or fake. It is essential to incorporate downsampling methods into Discriminator.

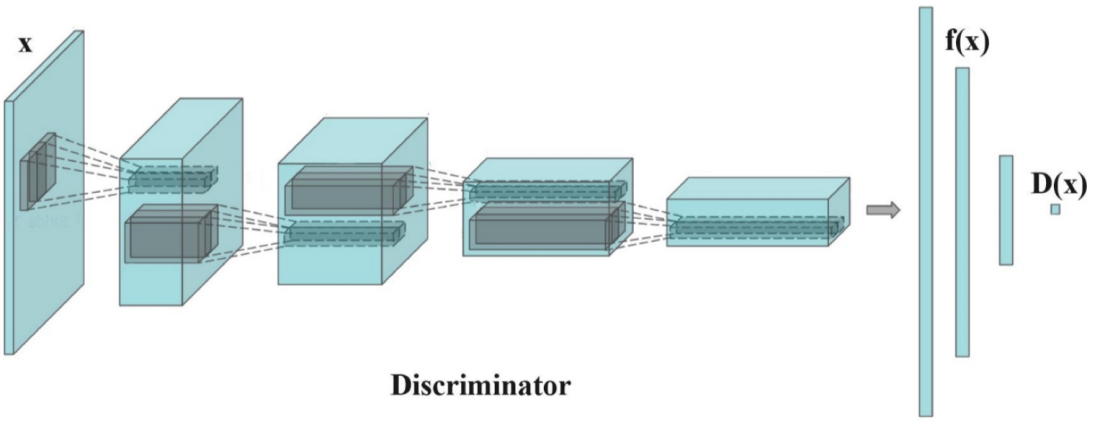


Fig-7: Example Architecture of Discriminator

Generator

The purpose of the generator is to generate realistic images from a random noise vector. In our project, it is designed to generate dog images. From a vector to an image, upsampling methods are always applied to the Generator.

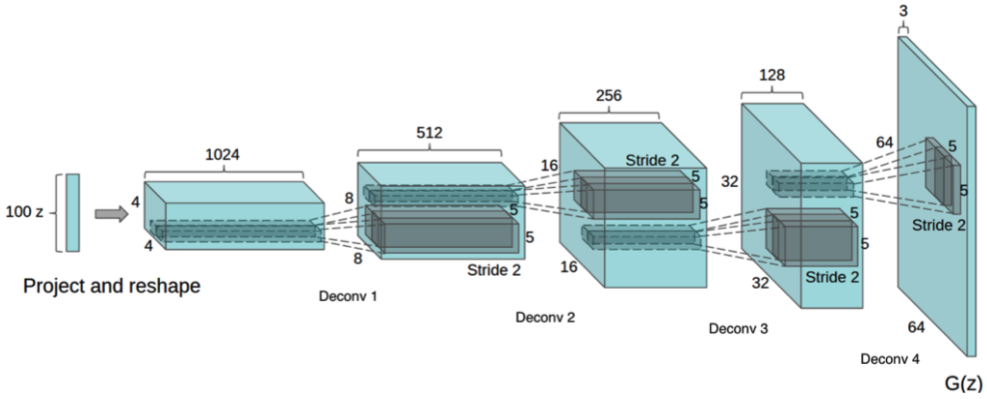


Fig-8: Example Architecture of Generator

Training Discriminator

In training the Discriminator for a DCGAN, the process involves several key steps. All weights within this network are initialized. Following this, the initialized Generator is utilized to generate a batch of fake images from random noise inputs. Next, real images, randomly selected from the original dataset, are labeled as class 1, while the fake images produced by the Generator are

labeled as class 0. Lastly, the Discriminator is trained to differentiate between these real and fake images, during which its weights are updated accordingly.

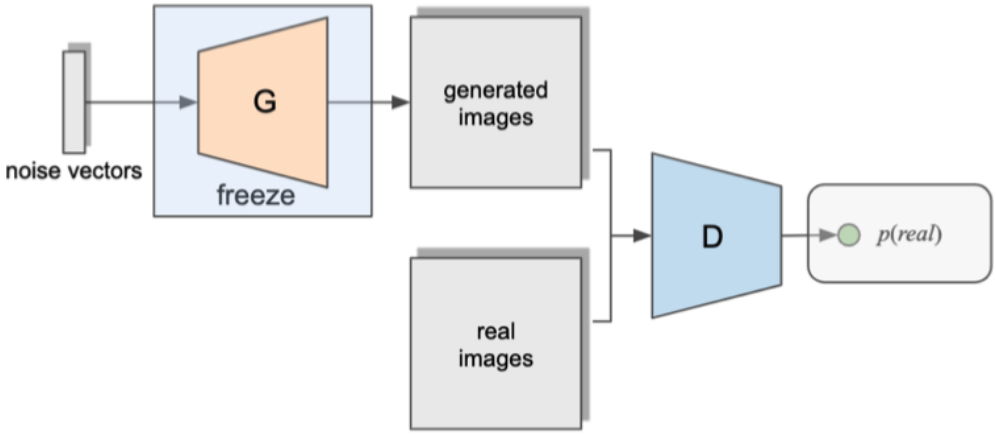


Fig-9: Training Discriminator

Training Generator

To train the Generator in a DCGAN, first, ensure the Discriminator's weights remain fixed. Begin by inputting a batch of random noise into the Generator to produce fake images. Then, feed these fake images into the already-trained Discriminator to evaluate the likelihood of them being perceived as real. During the training of the Generator, the goal is to have the input noise classified as class 1 (real). Therefore, use the target label 1 to compute the loss and gradients, which will in turn update the Generator's weights.

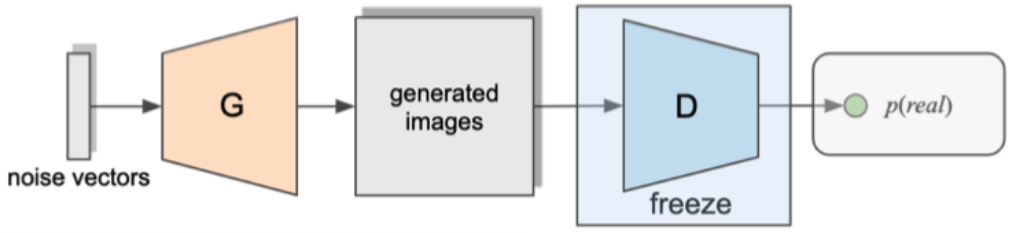


Fig-10: Training Generator

Train Details

- Preprocessing, scaled to range of tanh activation [-1, 1]

Images will be preprocessed to match the input requirements of the network. Since our generator uses a Tanh activation function in the output layer, images will be scaled to the range of [-1, 1] to align with the Tanh output.

- Batch_Size = 64

The size of mini-batches is set at 64. This size is often chosen based on computational constraints (like memory limits) and how well it fits the model's learning dynamics.

- Weight Initialization

Weight Initialization: Weights of convolution layers will be initialized from a normal distribution with a mean of 0.0 and a standard deviation of 0.02, which is a common practice in training deep neural networks to prevent models from being stuck at the start of training.

- Learning Rate

An initial learning rate of 0.001 with a Beta value of 0.5 for the Adam optimizer is used. The learning rate might need to be adjusted based on the model's convergence rate during training.

- Non-saturating activation functions(ReLU, LeakyReLu)

The issue of saturating loss function cannot provide sufficient gradient for G to learn. It also comes with a vanishing gradient problem in this minimax game. By implementing non-saturating activation, it allows the pass of negative with a small value.

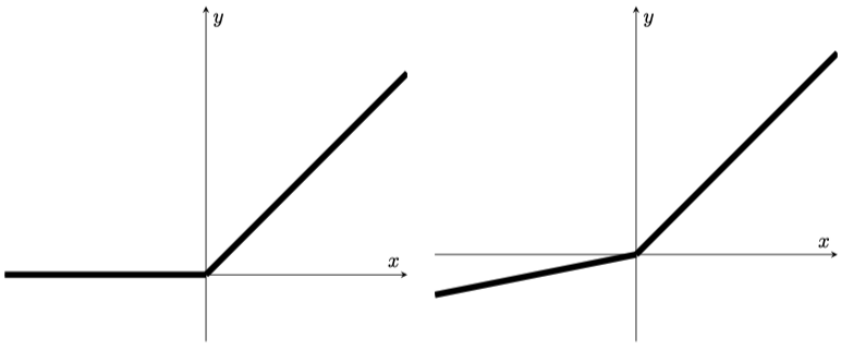


Fig-11: ReLU vs LeakyReLU

UNet2D Diffuser

Train Details

- Preprocessing, random horizontal flips, tensor image normalized to [0.5, 0.5] (mean, std)

The preprocessing of only random horizontal flips was to ensure that the dataset integrity would not be altered during the training process (color shifting and vertical flips on dogs rarely appear in real life). The image's distribution was normalized 0.5, effectively shifting the pixel intensity values the same way the GAN settings were scaled from [-1, 1].

- Image_Size = 128

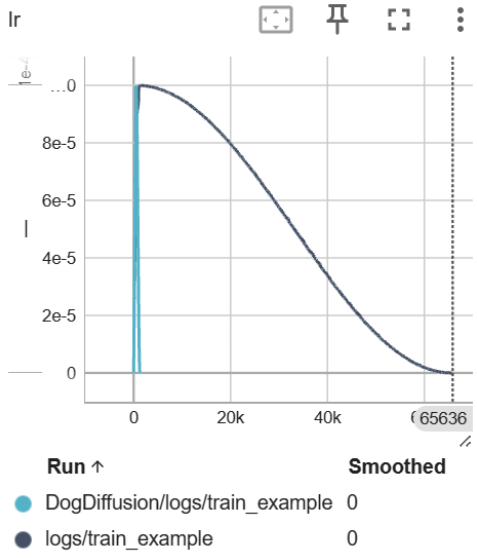
An image size of 128 was chosen to balance training costs and pattern retention of the original image. For the downsampling blocks of the model's architecture to accurately capture context, a minimum resolution was required. If possible, a higher image quality could have led to better model performance in the same training period. The evaluation batch size was also set to 16.

- Batch_Size = 16

The batch size was set at 16, due to instance GPU limitations. Lower batch sizes were not chosen, either due to increasing the minutes per epoch and thus, lowering the number of epochs for training purposes, and also due to the large number of dog classes and visual differences within the dataset, as well as instability problems in the model's learning process. If possible, a higher batch size could have led to better model performance over the same training period.

- Learning Rate = 0.001

While the learning rate was set to 0.001, the training configuration allowed for 500 learning rate warmup steps before reaching the original learning rate. Learning rate was also adjusted throughout the epochs and training process, shown in the figure. As the model warms up, the model slowly reaches its original learning rate, before gradually decreasing throughout the lifetime of the training process. In the light blue line, the graph shows a quick warmup rise into a quick descent, as the lifetime of the training process is very short for a 1 epoch model. In the dark blue line, the learning rate similarly warms up, but then decreases slowly to a tenth of the original set learning rate over the training period.



- Optimizer = AdamW

AdamW is a basic variation of the famous and pervasive Adam optimizer, with added Weight Decay. Weight Decay is also known as Ridge Regression, or L2, and simply adds a penalty to the gradient depending on the weight's size, which can help prevent overfitting.

- Epochs = 50

The number of Epochs ran was 50. The model saw (subjectively, from analyzing the images after every 20 epochs) noticeable improvement in capturing the shape, color, and patterns of the dogs as more epochs ran.

Result

DCGAN

In the context of DCGAN compared to a simple GAN with an MLP structure, it's observed that a finely-tuned deep convolutional structure in DCGANs can discern more intricate details in images. However, without proper tuning, DCGANs may perform worse than a basic GAN with MLP, sometimes producing images with block-like artifacts. Even after addressing these issues, DCGANs can still struggle with generating images that closely resemble the target, indicating a misdirection in the learning process of both the Discriminator and Generator. Despite some improvements in image quality, DCGANs are not immune to the mode collapse issue. The tuning process for DCGANs, involving two deep CNNs, is significantly more complex and sensitive to changes than basic GANs, often requiring extensive effort and experimentation. Solutions from other studies or projects may not always be directly applicable or effective in enhancing the performance of the GAN model in specific projects.

Fig-13: Generated Dog Images after 100 epoch

Fine-Tuning Results

- 20% Dropout on Generator's three hidden layers and Discriminator's two hidden layers and 40 Degree Random Rotation

Reason : Dropout is a regularization technique used to prevent overfitting; it can encourage the generator to explore a wider range of outputs, potentially leading to more diverse and realistic image generation. Also A 20% dropout rate in two hidden layers of the discriminator can help

the discriminator from becoming too powerful compared to the generator, especially in the early phases of training. By weakening the discriminator slightly, the generator has a better chance of improving its image generation capabilities without being overwhelmed by the discriminator's accuracy. And 40-degree rotation can also add diversity to the dataset without introducing unrealistic or overly distorted images.

Fig-13.1: Fine-tuning Results 1

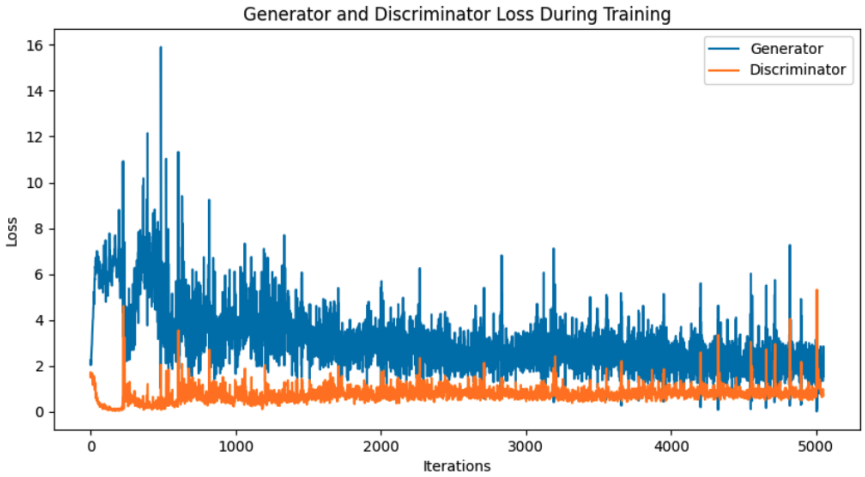


Fig-13.2: Fine-tuning Results 1

Interpretation: From the fake image showcase, it shows that our model is able to capture those colorful information and some dogs' shapes, but fails to present the dog with detailed

information. The loss plot shows that there is some fluctuation in the first 1000 iterations, which indicates some model instability.

- Add LayerNorm in Discriminator

Reason: It would normalize the activations of the previous layer for each data sample, making the training less sensitive to the scale of weights and the initial values. It may improve the training stability, make the model less sensitive to choice of hyperparameters.

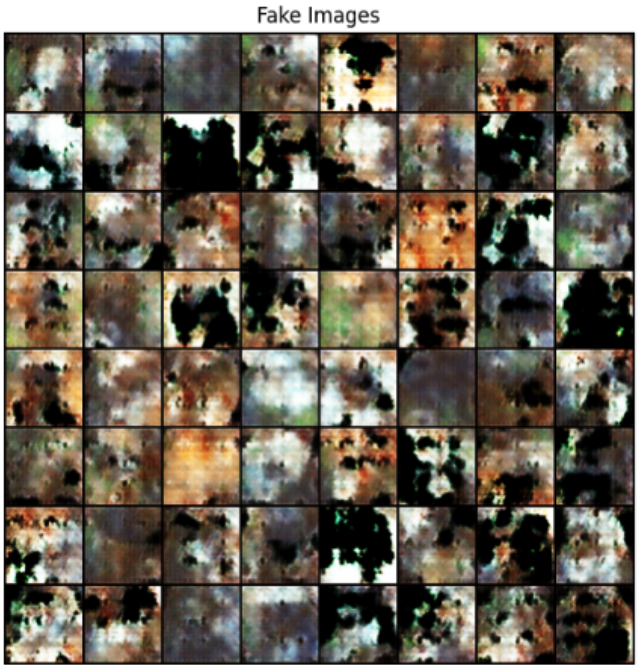


Fig-14.1: Fine-tuning Results 2

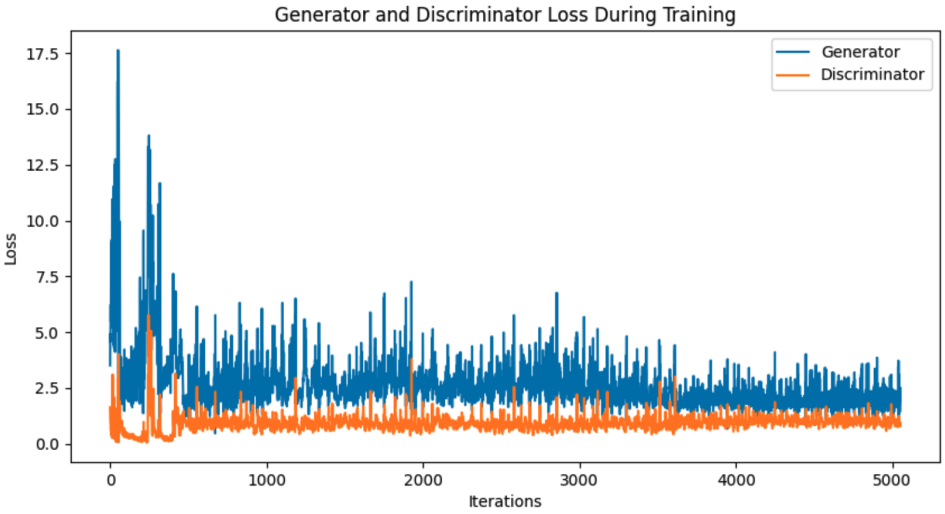


Fig-14.2: Fine-tuning Results 2

Interpretation: After this fine-tuning techniques, it is clear to see that the outcome of fake images are not descent, even though from the loss plot, the model tends to be stable after around 500 iterations.

- Apply LeakyReLu with 0.2 both on Generator and Discriminator

Reason: The "leak" in LeakyReLU is defined by a small slope with 0.2, which means that LeakyReLU will output some small negative value when the input is negative.The implementation in Generator can help Generator maintaining diversity in generated images, as it prevents the loss of gradient information throughout the Layers. Similarly for Discriminator, LeakyReLU ensures that all neurons remain active and contribute to distinguishing between real and fake images, which is vital for effectively training the generator.

Fig-15.1: Fine-tuning Results 3

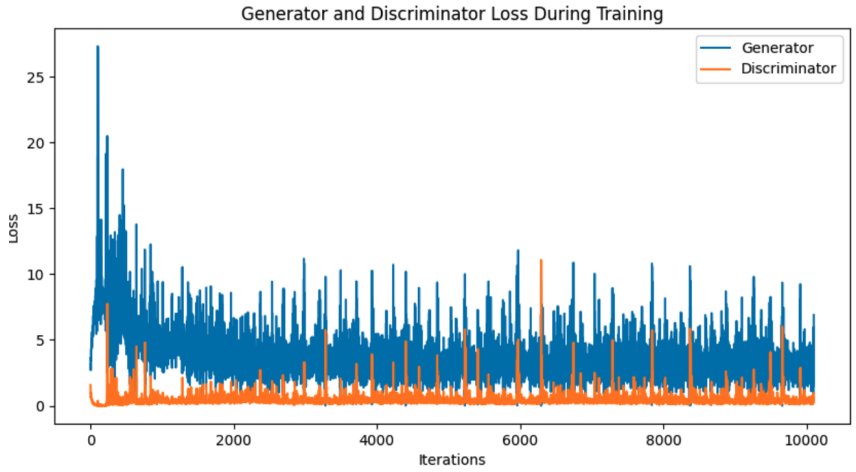


Fig-15.2: Fine-tuning Results 3

Interpretation: Compared with previous fine-tuning, this result shows a much better performance. It is able to capture the shape features but still fails to capture the detailed information.

- BETA = 0.8, Giving higher weights on recent gradient

Reason: By setting Beta1 to 0.8, we are trying to place more emphasis on the more recent gradients as opposed to the historical average. This means the optimizer will be more responsive to recent changes in the gradient, which can lead to faster adjustments in the model weights.

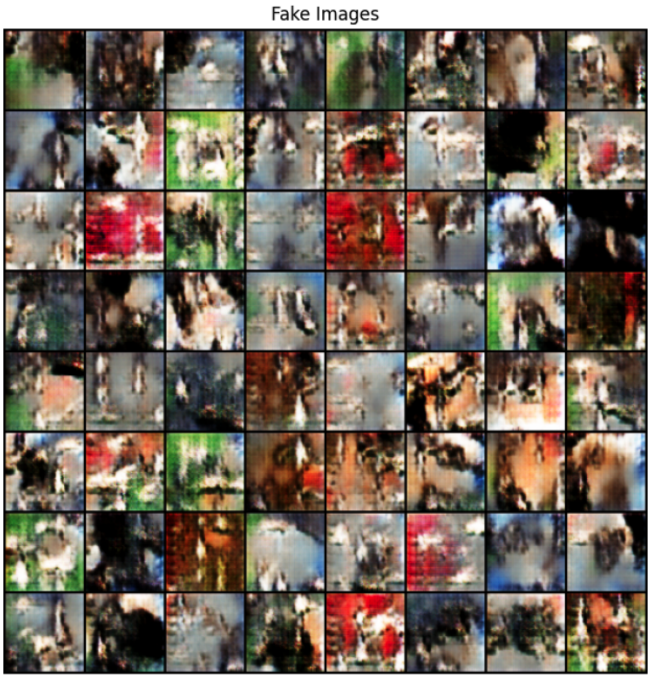


Fig-16.1: Fine-tuning Results 4

Fig-16.1: Fine-tuning Results 4

Interpretation: By adjusting the BETA value, the model turns worse. And from the loss plot, the Discriminator is overpowering the Generator, which means that it is difficult for Generator to improve.

UNet2D Diffuser

The loss function used for the model was mean squared error. MSE specifically targets better reconstruction at the pixel level for noise prediction problems, penalizing between the predicted and actual noise values that was added in the forward process of the noise scheduler, and also helps reassure the stability of the model.
Here are some example images from multiple epochs of the model.

Fig 17-1/2, Epoch 0 and 1 of the Diffusion Evaluation

From Epoch 0 and 1, the original noisy and abstract image with which the model must derive a dog picture from is seen. From Epoch 1, a few minor changes can be noticed. Some colors, like in image 1, 7, and 9 (left to right, top to bottom), are darkened, and the quality of the images are sharper than the original templates.

Fig 17-3/4, Epoch 20 and 40 of the Diffusion Evaluation

There is a big change between the first few epochs, and the latter epochs in 20 and 40. Between the first and twentieth epoch, the image begins to focus on the center of the image, as shapes and

patterns can be made out. In the 20th epoch, image 10 begins to take the shape of a dog, while image 13 captures the pattern of a dog's nose and tongue. In the 40th epoch, these images are sharpened further, and although the patterns of the dogs are still difficult to see in most images, progress is still noticeable.

Fig 17-5, Epoch 50

Finally, in the 50th epoch, more colors and backgrounds are sharpened. The first image has changed drastically, as the patterns of legs can now be made out from the original images, behind a green background as opposed to the bluish background at the start. Likewise, the center of each image tends to focus on a black, beige, or white subject, which are more common dog fur colors than the original colors in the image, tinted by the background.

Summary and conclusions

In the future, we could improve the preprocessing steps in our dataset, to help more patterns of the dog images reach the training process of our models. We could also explore more models, like pretrained GANs and diffusion models, and more types of problems, including adding prompts to help with the fine tuning process of pretrained image generation models.

References

A. Radford, L. Metz, S. Chintala, Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks (2015)

Avinash H. (2017). The GAN Zoo. GitHub. https://github.com/hindupuravinash/thegan-zoo

Ho, J., Jain, A., & Abbeel, P. (2020). Denoising diffusion probabilistic models. Advances in neural information processing systems, 33, 6840-6851.

Ian Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, S. Ozair, Y. Bengio, Generative Adversarial Networks (2014)

Jonathan H. (2018). GAN — Ways to improve GAN performance. Towards Data Science. https://towardsdatascience.com/gan-ways-to-improve-gan-performance-acf37f9f59b

Jonathan H. (2018). GAN — DCGAN (Deep convolutional generative adversarial networks) . Medium. https://medium.com/@jonathan_hui/gan-dcgan-deep-convolutionalgenerative-adversarial-networks-df855c438f

Jonathan H. (2018). GAN — Why it is so hard to train Generative Adversarial Networks! Medium. https://medium.com/@jonathan_hui/gan-why-it-is-so-hard-to-traingenerative-advisory-networks-819a86b3750b

Khosla, A., Jayadevaprakash, N., Yao, B., & Li, F.-F. (n.d.). Novel Dataset for Fine-Grained Image Categorization: Stanford Dogs. Retrieved April 27, 2022, from http://people.csail.mit.edu/khosla/papers/fgvc2011.pdf

Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation. arXiv preprint arXiv:1710.10196.

Ledig, C., Theis, L., Huszaʹr, F., Caballero, J., Cunningham, A., Acosta, A., Aitken, A., Tejani, A., Totz, J., Wang, Z., et al. Photo-realistic single image super-resolution using a generative adversarial network. arXiv preprint arXiv:1609.04802, 2016.

Sha, M. A. (2021, January 5). Training Generative Adversarial Networks (GANs) for DOG using PyTorch. Medium. https://medium.com/@mirjanalisha/training-generative-adversarial-networks-gans-for-dog-using-pytorch-26d15ec73d1b

Sohl-Dickstein, J., Weiss, E., Maheswaranathan, N., & Ganguli, S. (2015). Deep Unsupervised Learning using Nonequilibrium Thermodynamics. In Proceedings of the 32nd International Conference on Machine Learning (ICML) (Vol. 37, pp. 2256–2265). PMLR. Retrieved from https://proceedings.mlr.press/v37/sohl-dickstein15.pdf

Stanford Dogs dataset for Fine-Grained Visual Categorization. (n.d.). Vision.stanford.edu. http://vision.stanford.edu/aditya86/ImageNetDogs/

Stanford-dogs/data/stanford_dogs_data.py at master · zrsmithson/Stanford-dogs. (n.d.). GitHub. Retrieved December 9, 2023, from https://github.com/zrsmithson/Stanford-dogs/blob/master/data/stanford_dogs_data.py

Zhang, J. (2019, October 18). UNet line by line explanation. Medium. https://towardsdatascience.com/unet-line-by-line-explanation-9b191c76baf5

Appendix

```python
# Generator
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # Input is Z, going into a convolution
            nn.ConvTranspose2d(NZ, NGF*8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(NGF*8),
            nn.ReLU(True),
            # State size: (NGF*8) x 4 x 4
            nn.ConvTranspose2d(NGF*8, NGF*4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(NGF*4),
            nn.ReLU(True),
            # State size: (NGF*4) x 8 x 8
            nn.ConvTranspose2d(NGF*4, NGF*2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(NGF*2),
            nn.ReLU(True),
            # State size: (NGF*2) x 16 x 16
            nn.ConvTranspose2d(NGF*2, NGF, 4, 2, 1, bias=False),
            nn.BatchNorm2d(NGF),
            nn.ReLU(True),
            # State size: NGF x 32 x 32
            nn.ConvTranspose2d(NGF, NC, 4, 2, 1, bias=False),
            nn.Tanh()
            # State size: NC x 64 x 64
        )

    def forward(self, input):
        return self.main(input)
```

Code 1: The Generator of the DCGAN

```
# Discriminator
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # Input is (NC) x 64 x 64
            nn.Conv2d(NC, NDF, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # State size: NDF x 32 x 32
            nn.Conv2d(NDF, NDF*2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(NDF*2),
            nn.LeakyReLU(0.2, inplace=True),
            # State size: (NDF*2) x 16 x 16
            nn.Conv2d(NDF*2, NDF*4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(NDF*4),
            nn.LeakyReLU(0.2, inplace=True),
            # State size: (NDF*4) x 8 x 8
            nn.Conv2d(NDF*4, NDF*8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(NDF*8),
            nn.LeakyReLU(0.2, inplace=True),
            # State size: (NDF*8) x 4 x 4
            nn.Conv2d(NDF*8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)
```

Code 2: The Discriminator of DCGAN

```python
model = UNet2DModel(
    sample_size=config.image_size,  # the target image resolution
    in_channels=3,  # the number of input channels, 3 for RGB images
    out_channels=3,  # the number of output channels
    layers_per_block=2,  # how many ResNet layers to use per UNet block
    block_out_channels=(128, 128, 256, 256, 512, 512),  # the number of output channels for each UNet block
    down_block_types=(
        "DownBlock2D",  # a regular ResNet downsampling block
        "DownBlock2D",
        "DownBlock2D",
        "DownBlock2D",
        "AttnDownBlock2D",  # a ResNet downsampling block with spatial self-attention
        "DownBlock2D",
    ),
    up_block_types=(
        "UpBlock2D",  # a regular ResNet upsampling block
        "AttnUpBlock2D",  # a ResNet upsampling block with spatial self-attention
        "UpBlock2D",
        "UpBlock2D",
        "UpBlock2D",
        "UpBlock2D",
    ),
)
```

Code 3: UNet2DModel Architecture