Synthetic Anime Face Generation Using Denoising Diffusion Probabilistic Models

A Diffusion Approach for Cartoon Face Generation

TSE Wang Pok · The University of Hong Kong, Hong Kong · *crazytse@connect.hku.hk*

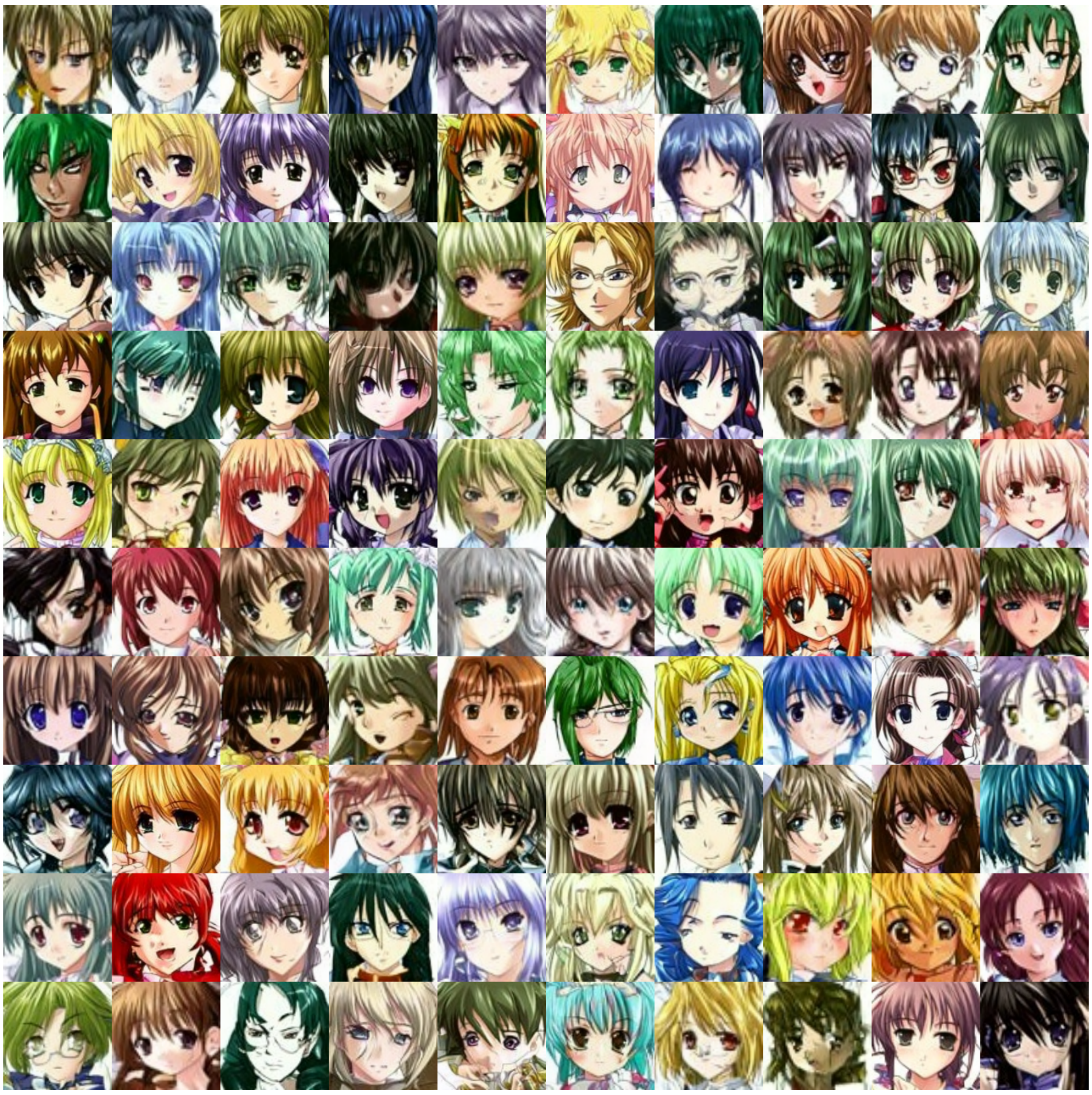


Figure 1. A 10x10 grid of anime faces generated by our DDPM model, demonstrating diversity in facial features, hairstyles, and expressions while maintaining anime aesthetics.

Abstract. We present a diffusion-based approach for generating anime-style faces using Denoising Diffusion Probabilistic Models (DDPMs). Our method trains a U-Net architecture on an animation face dataset to iteratively transform random noise into coherent anime portraits. The model produces diverse facial features, hairstyles, and expressions while maintaining the distinctive aesthetic of anime art. Experimental results demonstrate stable training and high-quality generation of 100 unique anime faces.

1 Introduction

Anime face generation presents unique challenges, requiring preservation of stylistic conventions-exaggerated features, different hair colors, expressive eyes, and stylized proportions that must be consistently maintained while ensuring facial coherence and diversity. Traditional generative approaches such as Generative Adversarial Networks (GANs) **?** often suffer from training instability and mode collapse, limiting their effectiveness.

We apply Denoising Diffusion Probabilistic Models (DDPMs) **?** to anime face generation, leveraging their stable training process and strong mode coverage. Our method learns to denoise random Gaussian distribution into structured anime faces through iterative refinement. Trained on a Kaggle anime face dataset, our approach can generate diverse, high-quality characters consistent with anime aesthetics.

2 Methodology

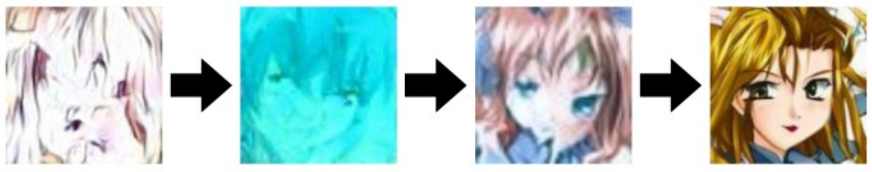


Figure 2. Generation workflow: from epoch 0 to epoch 199

2.1 Denoising Diffusion Probabilistic Models

Diffusion models work by learning to remove noise from images through two steps: first adding noise to clean images (forward process), then training a neural network to reverse this process (reverse process).

2.1.1 Forward Diffusion Process

We start with a clean anime face image x_0 and gradually add small amounts of random noise over 1000 steps. At each step t, the image becomes slightly more noisy until it become completely random noise. This process can be described as:

$$x_t = \sqrt{\alpha_t} x_{t-1} + \sqrt{1-\alpha_t}\epsilon \quad (1)$$

where ϵ is random noise and α_t controls how much noise we add at each step.

2.1.2 Reverse Denoising Process

The model learns to predict and remove the noise that was added during the forward process. Given a noisy image x_t at timestep t, the network predicts the noise $\epsilon_\theta(x_t, t)$ that needs to be removed.

$$\text{Loss} = ||\epsilon - \epsilon_\theta(x_t, t)||^2 \tag{2}$$

where ϵ is the actual noise that was added, and ϵ_θ is the noise predicted by our model.

2.2 Model Architecture

We use a U-Net **?** with residual connections and attention mechanisms, optimized for the generation of 128x128x3 anime faces. The architecture includes encoder-decoder blocks with self-attention at intermediate resolutions to capture both local details and global structure.

3 Results and Analysis

Our model achieved stable convergence over 200 training epochs with consistently decreasing loss. The generated 100 anime faces demonstrate:

3.1 Structural Coherence

Most of the generated faces maintain proper anatomical structure with correctly positioned facial features. Eyes, nose, and mouth exhibit consistent placement and proportions characteristic of anime aesthetics. The model learned to generate symmetrical faces without significant deformations or artifacts.

3.2 Style Consistency

The generated faces faithfully adhere to anime art conventions, including:

- Large, expressive eyes with varied shapes and colors
- Simplified facial features with smooth skin textures
- Exaggerated hair styles with vibrant colors and detailed strands
- Consistent character proportions across all generations

3.3 Diversity Analysis

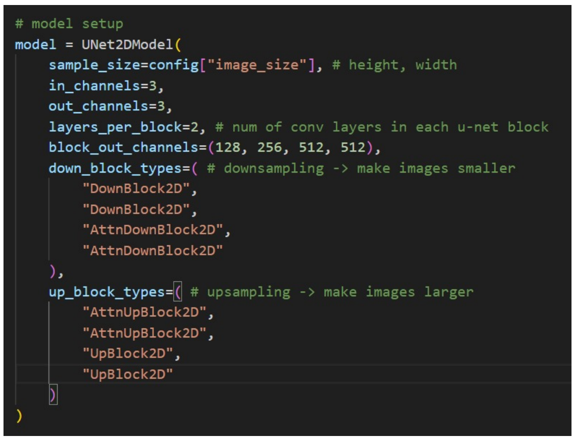
The 100 generated samples demonstrate substantial variation in:

- **Hair Characteristics**: Multiple colors (pink, blue, silver, brown, black), styles (long, short, twintails, ponytails), and textures
- **Eye Features**: Different shapes (round, almond-shaped), sizes, and colors while maintaining anime exaggeration

- **Facial Expressions**: Range from neutral to smiling, surprised, and serious expressions
- **Character Archetypes**: Various character types including school-age, fantasy, and contemporary styles

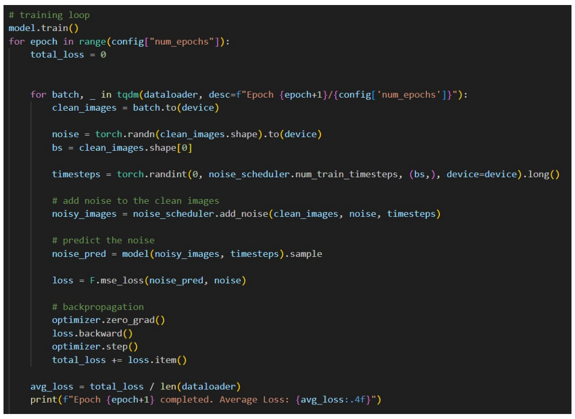
The results confirm DDPMs' effectiveness for stylized character generation, producing plausible anime faces while maintaining training stability.

4 Code Examples



```python
# model setup
model = UNet2DModel(
    sample_size=config["image_size"], # height, width
    in_channels=3,
    out_channels=3,
    layers_per_block=2, # num of conv layers in each u-net block
    block_out_channels=(128, 256, 512, 512),
    down_block_types=( # downsampling -> make images smaller
        "DownBlock2D",
        "DownBlock2D",
        "AttnDownBlock2D",
        "AttnDownBlock2D"
    ),
    up_block_types=( # upsampling -> make images larger
        "AttnUpBlock2D",
        "AttnUpBlock2D",
        "UpBlock2D",
        "UpBlock2D"
    )
)
```

Figure 3. Model Setup



```python
# training loop
model.train()
for epoch in range(config["num_epochs"]):
    total_loss = 0


    for batch, _ in tqdm(dataloader, desc=f"Epoch {epoch+1}/{config['num_epochs']}"):
        clean_images = batch.to(device)

        noise = torch.randn(clean_images.shape).to(device)
        bs = clean_images.shape[0]

        timesteps = torch.randint(0, noise_scheduler.num_train_timesteps, (bs,), device=device).long()

        # add noise to the clean images
        noisy_images = noise_scheduler.add_noise(clean_images, noise, timesteps)

        # predict the noise
        noise_pred = model(noisy_images, timesteps).sample

        loss = F.mse_loss(noise_pred, noise)

        # backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    avg_loss = total_loss / len(dataloader)
    print(f"Epoch {epoch+1} completed. Average Loss: {avg_loss:.4f}")
```

Figure 4. Training Loop