Report of HW5

Huanyu Wang 522030910212

Task 1

The final best parameters of my model:

- **IMG_SIZE = 64**
  The size of the input images. Smaller image sizes can speed up training but may result in lower-quality image generation.
- **batch_size = 128**
  The number of samples in each batch during training. Larger batch sizes require more memory but can reduce the variance of the gradient estimates.
- **train_num_steps = 40000**
  The total number of training steps, which represents the number of times the model will process the entire training dataset. A larger number of steps allows the model to learn more effectively, but it also requires more time and computational resources.
- **lr = 1e-3**
  The learning rate controls how much the model's weights are updated with each step. Relatively small learning rate can help the model converge slowly but steadily without overshooting the optimal parameters. However, if it's too small, training may be unnecessarily slow.
- **grad_steps = 1**
  The number of gradient steps per update. In this case, the model updates the weights once per batch. Larger gradient steps can be used for more effective updates, but this could increase memory usage and training time.
- **ema_decay = 0.997**
  The exponential moving average decay factor. EMA is used to maintain a smoother version of the model weights during training. A decay factor of 0.997 means that the EMA model is updated slowly to preserve stability, which often helps improve model generalization.
- **channels = 128**
  The number of channels in the first layer of the model. More channels can allow the model to capture more complex patterns, but it also increases the computational cost.
- **dim_mults = (1, 2, 4)**
  The scaling of the model's dimensionality at each layer of the network. As the model progresses deeper, the number of channels increases by factors of 1, 2, 4, and 8. This progressive increase in dimensions helps the model capture more complex features at higher layers.
- **timesteps = 3000**
  The number of timesteps in the diffusion process. In a diffusion model, data is gradually corrupted by noise over a series of timesteps, and the model learns to reverse this process. A higher number of timesteps allows the model to perform more fine-grained denoising but can also increase training time.
- **beta_schedule = 'linear'**
  The schedule for the noise variance during the diffusion process. A linear schedule means that the noise level increases gradually over the timesteps. Linear schedules are commonly used as they provide a smooth transition between clean and noisy data, which helps the model learn better denoising strategies.

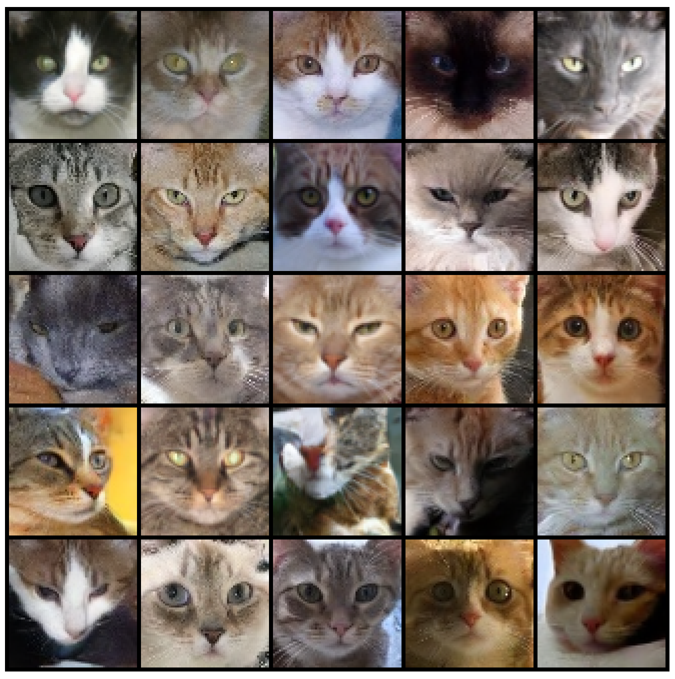
The main improvement includes increasing the channels, which enables the model to catch more details from the image, at the cost of more training and inferencing time.

The training number steps are set to 40000 because I observe that the more training steps, the better the result is. I think the training number steps can even be higher, but due to the high cost of time, I cannot afford to try higher steps.

Besides, learning rate is decreased and batch size is increased, which can make the training process more delicate. Also, a larger batch size enables the training process to go through the training dataset for more times, thus improving the result.

Also, more timesteps can add more noise and make the model generate better.

The sample of random generation with diffusion(1000 cat faces):



The FID score is **18.707184981213345**

Task 2

Fusion generation with diffusion(50 pairs):

The parameter lambda is set to **0.5**, trying to get the exact average feature of the two cat faces.

As the source images exist many black cats, the fusion image of two dark-hair cats is not satisfying. Thus, I manually selected 50 pairs of cats that are not both black.

Here are some quite satisfying results:

The FID score is **70.54680137714281**

Task 3

```python
def model_predictions(self, x, t, clip_x_start = False, rederive_pred_noise = False):
    """
    Perform predictions for the reverse process in a diffusion model.

    Returns:
    - pred_noise (torch.Tensor): The predicted noise component at timestep t, representing the model's
    learned approximation of the noise added during the forward diffusion process.
    - x_start (torch.Tensor): The reconstructed clean sample based on the predicted noise.

    1. U-Net (self.model) in Diffusion:
        U-Net learns to estimate the noise component added
        during the forward process. By doing so, it enables the reverse process to iteratively denoise
        the noisy input and reconstruct the original data.

    2. Reconstructing x_start:
        - x_start is derived using the reverse process formula:
            x_0 ≈ (x_t - sqrt(1 - alpha_t) * ε) / sqrt(alpha_t).
        - The method self.predict_start_from_noise implements this calculation.
    """



def p_sample(self, x, t: int):
    """
    Perform a single step of the reverse diffusion process, generating a sample x_t-1
    from x_t at the given timestep t.

    Process Overview:
    1. The function first computes the model's predicted mean and log variance at the
    given time step t using the p_mean_variance method.
    2. The denoised image is generated by adding the scaled noise to the predicted mean,
    based on the model's predicted variance.

    Returns:
    - pred_img: The denoised image predicted by the model at time step t. This is the output of
    the reverse diffusion process for this step.
    - x_start: The estimated original image sample, which can be used for further processing
    or reference in the model's denoising steps.
    """



def p_sample_loop(self, shape, return_all_timesteps = False):
    """
    This function performs the full sampling loop for generating an image using the reverse diffusion process.
    It iteratively applies the p_sample function over all time steps,
    starting from random noise and progressively denoising it through the reverse diffusion steps.
    The loop starts with a random tensor and applies the reverse diffusion process (denoising) step by step,
    producing an image. Optionally, it can return the intermediate images at each time step.

    Process Overview:
    1. Initialize the input image as random noise of the specified shape.
    2. Iterate through the time steps in reverse order, starting from the final diffusion step (t = num_timesteps) and
    progressively applying the reverse denoising process (using p_sample).
    3. For each time step, p_sample is called to obtain a denoised image, and the intermediate images are stored.

    Returns:
    - ret: The generated image after the full reverse diffusion process.
    """
```

```python
def p_sample_loop2(self, shape, img1, img2, lamda,return_all_timesteps = False):
    """
    This function performs the reverse diffusion sampling loop to generate an image by combining
    two input images (img1 and img2) through a weighted interpolation, and then applying the
    reverse diffusion process iteratively.
    It starts by generating noisy versions of img1 and img2 at random time steps, and then
    interpolates between these noisy images based on the weight lamda. The loop then applies
    reverse denoising steps (p_sample) to progressively denoise the image over multiple timesteps.

    Process Overview:
    1. Two noisy images are generated from img1 and img2 at random time steps using the q_sample function.
    2. The two noisy images are then interpolated using lamda to form the initial image,
    which follows the formula of the diffusion model.
    3. The image is normalized before starting the reverse diffusion loop.
    4. The reverse diffusion process is applied iteratively using the p_sample function, and intermediate images
    are collected.
    5. Optionally, return all intermediate images (if return_all_timesteps is True), or just the final generated image.
    6. After the loop, the generated image is unnormalized and returned.

    Returns:
    - ret: The generated image after the reverse diffusion process.
    """


def q_sample(self, x_start, t, noise=None):
    """
    This function generates a noisy version of the input image at a given time step t
    by adding noise to it. The amount of noise added depends on the model's parameters at that time step.
    It uses the cumulative product of the diffusion steps (through sqrt_alphas_cumprod and sqrt_one_minus_alphas_cumprod)
    to control the strength of the noise added based on the current time step t. If no custom noise is provided,
    standard Gaussian noise is generated and added to the input.

    Process Overview:
    1. If no custom noise is provided, Gaussian noise is generated with the same shape as x_start.
    2. The noise is scaled and added to the original image based on the model's parameters (sqrt_alphas_cumprod
    and sqrt_one_minus_alphas_cumprod) at the given time step t, which follows the formuta of the diffusion model.
    3. The function returns the noisy image, which is a weighted combination of the original image and the noise.

    Returns:
    - A noisy version of the input x_start at time step t.
    """


def p_losses(self, x_start, t, noise = None):
    """
    This function computes the loss for a single timestep in the reverse diffusion process.
    The loss is based on the difference between the predicted noise and the true noise added to the original image
    during the forward diffusion process.
    The function first generates a noisy version of the input image at time step t using the q_sample
    function. It then computes the model's output by passing the noisy image through the model. The loss is calculated
    by comparing the model's output to the true noise and applying a weighting factor that depends on the current time
    step.

    Process Overview:
    1. A noisy version of the original image is generated using the q_sample function at the specified
    time step t by adding Gaussian noise.
    2. The noisy image is passed through the model to predict the noise.
    3. The loss between the predicted noise and the true noise is computed using the model's output.
    4. The loss is weighted by a factor that may vary with the time step t.

    Returns:
    - The average loss for this timestep, which measures how well the model predicted the noise added to x_start
    at time step t.
    """
```

Task 4

Train and test the diffusion model:

Because when calculating fid score, all images should be of the same size, the cats_crop images are resized to 64 in advance.

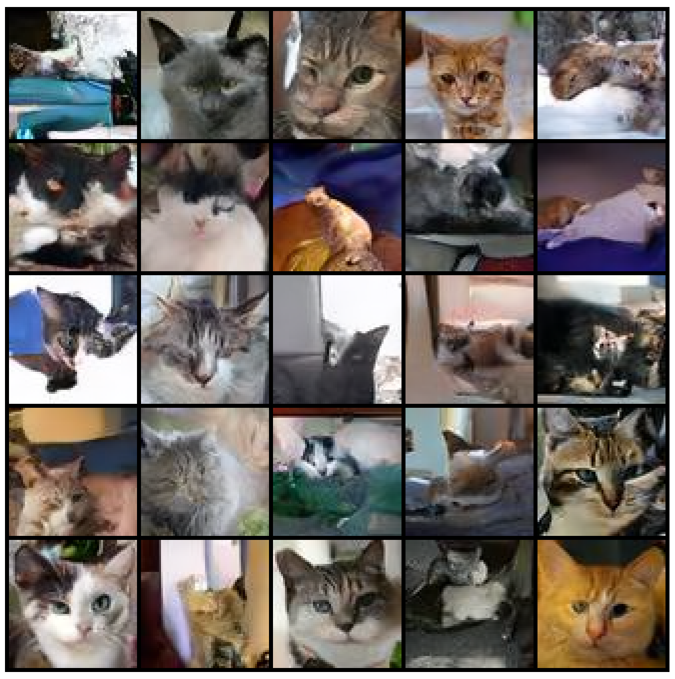
```
import os
from PIL import Image

source_imgs_dir='./Data/cats_crop/'
target_imgs_dir='./Data/resized/'
count = 0
for file in os.listdir(source_imgs_dir):
    im = Image.open(source_imgs_dir + file)
    out = im.resize((64, 64), Image.LANCZOS)
    out.save(target_imgs_dir + file)
    count += 1
    if count % 500 == 0:
        print(count, 'images resized')
```

The final best parameters of my model, which generally follows the parameters of the above model:

```
IMG_SIZE = 64
batch_size = 128
train_num_steps = 40000
lr = 1e-3
grad_steps = 1
ema_decay = 0.997
channels = 128
dim_mults = (1, 2, 4)
timesteps = 2000
beta_schedule = 'linear'
```

The sample of random generation with this model(1000 cat faces):



The FID score is **37.2648131649874**

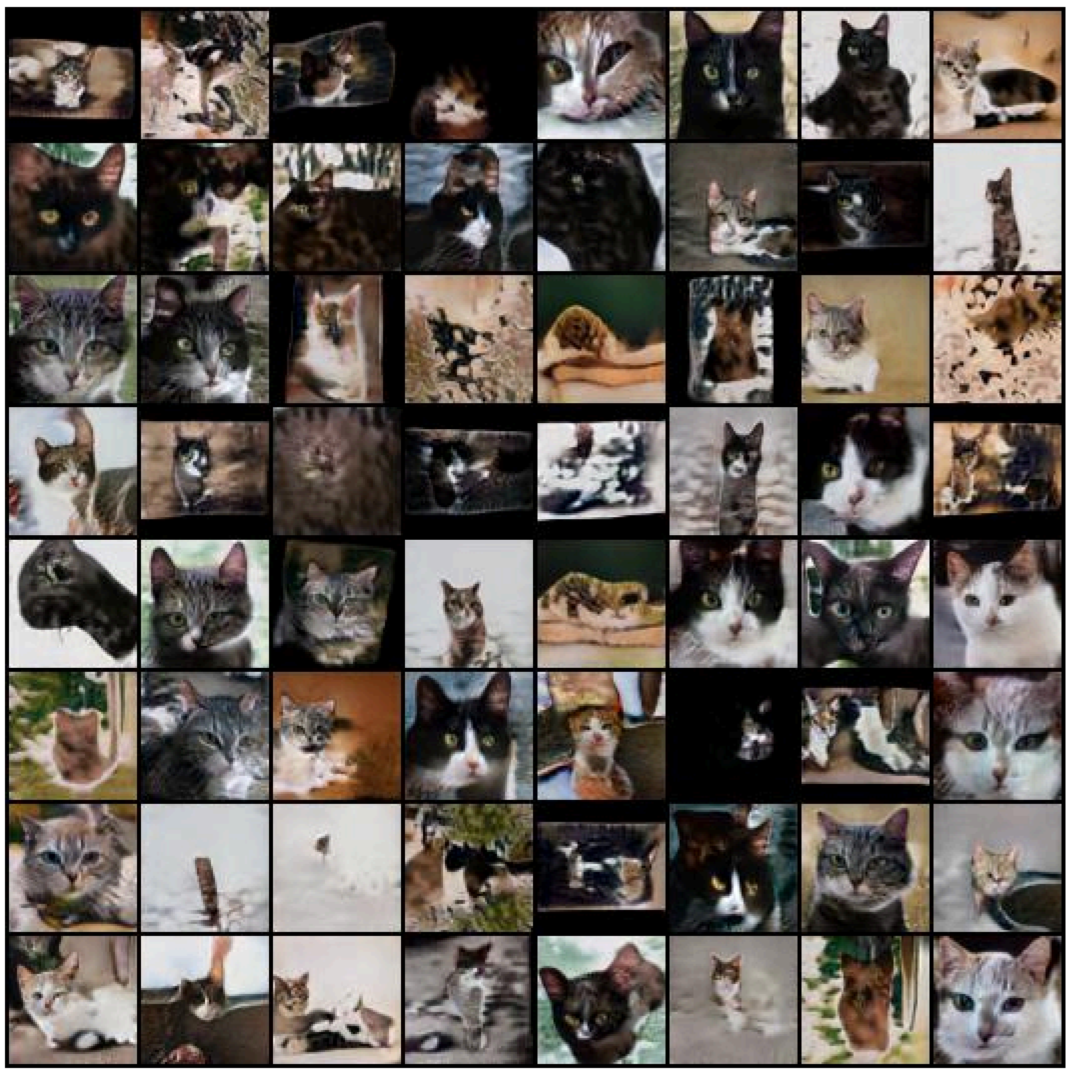
Train and test the styleGAN:

Here I use the cats_crop images that have been reshaped to square. Besides, the image-size is set to 64 to match the image size when training my own diffusion model above.

The final checkpoint is trained after 40000 steps also in order to match the training num steps of my diffusion model.

```
stylegan2_pytorch --data ./Data/cats_crop --image-size 64
```

The sample of random generation with this model:



Analyzing the differences

The main differences between diffusion models and styleGAN are:

1. Diffusion models use forward process (Diffusion) and reverse process (Denoising) to train the network, while styleGAN trains the generative model and the diecriminative model.
2. Diffusion models are much stabler during training, while styleGAN may be unstable.
3. Diffusion models are much slower when generating images because of its iterative denoising process, while styleGAN is faster during inference.
4. Both models produce high-quality results, but diffusion models are often more stable and capable of generating diverse outputs, while styleGAN excels in producing high-resolution and highly realistic images.

In this case, the generation of the styleGAN2 model is much better than those of mine diffusion model.
This is because of some advanced technologies and methods that styleGAN2 uses when training the model:

1. Remove the normalization and change the AdaIN network in order to eliminate the droplet artifacts.
2. Move the operations such as appling bias and noise outside the style block.
3. Base normalization on the expected statistics of the incoming feature maps, but without explicit forcing.
4. Lazy regularization: Compute the regularization terms less frequently than the main loss function.
5. Remove progressive growing structure, but using a skip generator and a residual discriminator.