

Aditya Kothari  
Research Presentation  
CIS-585 Advanced AI W23

# Image Synthesis using Denoising Diffusion Probabilistic Model



# Contents

---

- Image Synthesis using Deep Learning
- Techniques used for Image Synthesis
- Denoising Diffusion Probabilistic Model
- Dataset Used
- DDPM Model Implementation
- Challenges
- Training Parameters
- Generated Results
- Learnings

# Image Synthesis

---

- Image synthesis is the process of generating new images using a computer program or algorithm.
- Image synthesis has a long history in computer graphics, with early approaches focused on procedural techniques and rule-based systems.
- Deep learning has revolutionized image synthesis in recent years, enabling the creation of highly realistic and detailed images.
- Deep learning-based image synthesis has applications in a variety of fields, including entertainment, advertising, and medical imaging



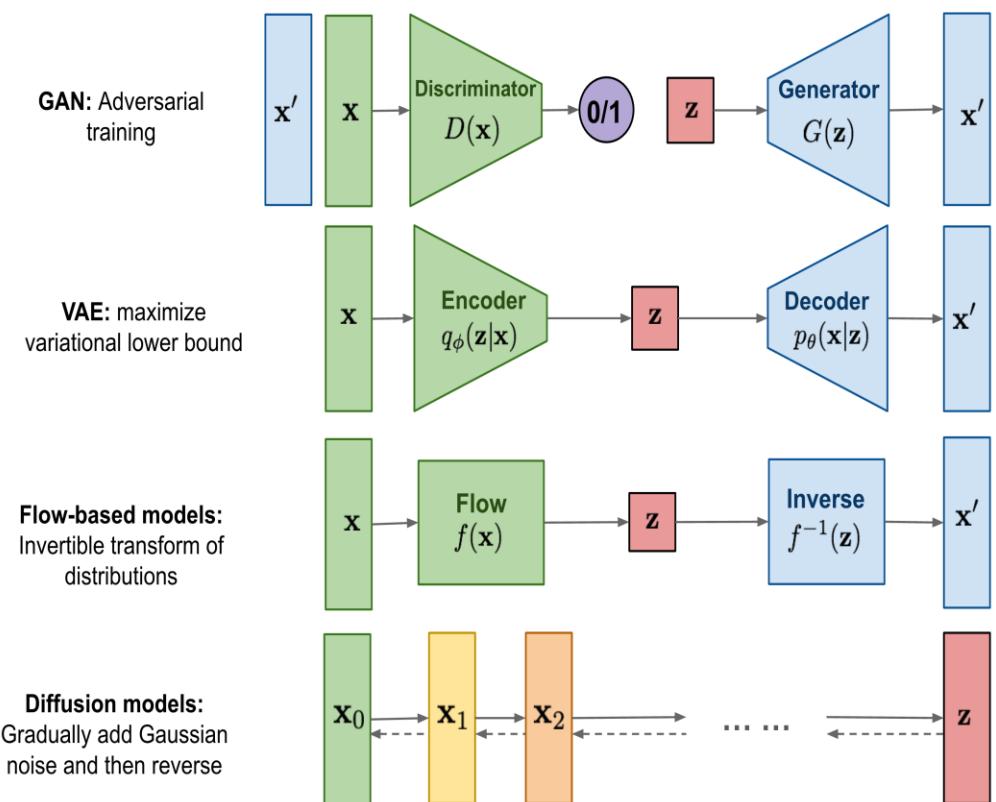
Image generated by Mid Journey

# Common Techniques used in Image Synthesis

---

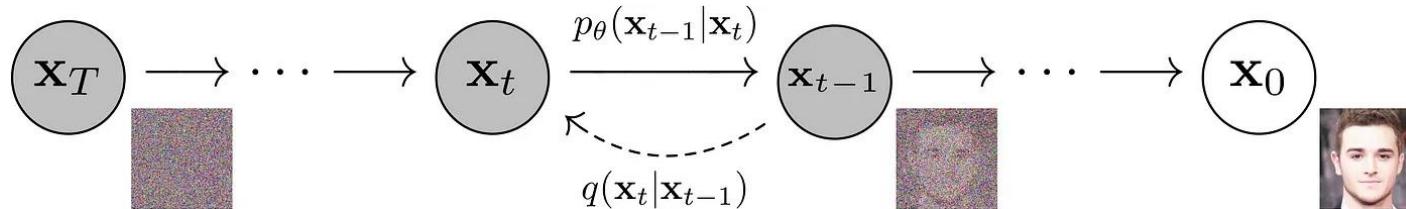
Commonly Used Image Synthesis Techniques are as follow:

- Generative Adversarial Networks (GANs)
  - The model learns to generate new data by having two neural networks compete against each other.
- Variational Autoencoders (VAEs)
  - The model learns to generate an image by encoding and decoding them through probabilistic framework
- Flow Based Generative Models
  - The model learns a series of invertible transformations to map simple distribution to complex image distribution.
- Diffusion Models
  - The model iteratively adds gaussian noise and then learns to reverse the process for generating an image



# Denoising Diffusion Probabilistic Model

- Denoising Diffusion Probabilistic Model (DDPM) is a generative model where the model learns the distribution of images by modeling the diffusion process that produces them.
- The complete process can be broken down in two steps:
  1. Forward Process: Given a dataset of images, noise is added step-by-step making the image less clear until only noise is left
  2. Backward Process: The model learns to undo each step of the forward process.



Forward Process Equations:

$$q(\mathbf{x}_t | \mathbf{x}_{t-1}) := \mathcal{N}(\mathbf{x}_t; \sqrt{1 - \beta_t} \mathbf{x}_{t-1}, \beta_t \mathbf{I})$$

$$\alpha_t := 1 - \beta_t$$

$$\bar{\alpha}_t := \prod_{s=1}^t \bar{\alpha}_s$$

$$q(\mathbf{x}_t | \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t} \mathbf{x}_0, (1 - \bar{\alpha}_t) \mathbf{I})$$

Backward Process Equations:

$$p_\theta(\mathbf{x}_{t-1} | \mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \sigma_t^2 \mathbf{I})$$

$$\boldsymbol{\mu}_\theta(x_t, t) = \frac{1}{\sqrt{\alpha_t}} (x_t - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_\theta(x_t, t))$$

Loss Function:

$$\mathbb{E}_{\mathbf{x}_0, \epsilon} \left[ \frac{\beta_t^2}{2\sigma_t^2 \alpha_t (1 - \bar{\alpha}_t)} \|\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t} \mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon, t)\|^2 \right]$$

# Datasets Used – Fashion MNIST

---

- A dataset of Zalando's article images
- Training set size = 60,000
- Testing set size = 10,000
- Image size = 28x28x1 (grayscale)
- Labels = 10



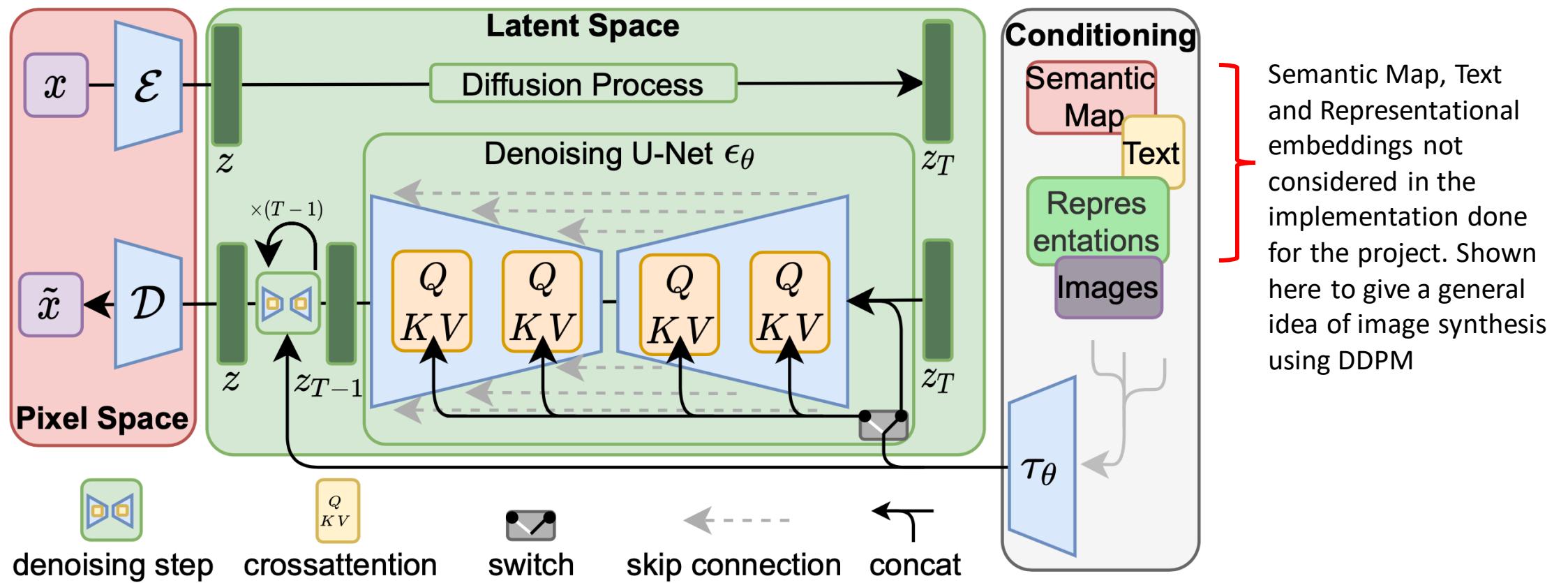
# Datasets Used – Flowers 102

---

- A dataset of 102 different categories of flowers commonly occurring in UK
- Total set size = 8189
- Image size = 256x256x3
- Labels = 102



# DDPM Implementation



# DDPM Implementation

```
# DDPM class
class MyDDPM(nn.Module):
    def __init__(self, network, n_steps=200, min_beta=10 ** -4, max_beta=0.02, device=None, image_chw=(3, 32, 32)):
        super(MyDDPM, self).__init__()
        self.n_steps = n_steps
        self.device = device
        self.image_chw = image_chw
        self.network = network.to(device)
        self.betas = torch.linspace(min_beta, max_beta, n_steps).to(device) # Number of steps is typically in the order of thousands
        self.alphas = 1 - self.betas
        self.alpha_bars = torch.tensor([torch.prod(self.alphas[:i + 1]) for i in range(len(self.alphas))]).to(device)

    def forward(self, x0, t, eta=None):
        # Make input image more noisy (we can directly skip to the desired step)
        n, c, h, w = x0.shape
        a_bar = self.alpha_bars[t]

        if eta is None:
            eta = torch.randn(n, c, h, w).to(self.device)

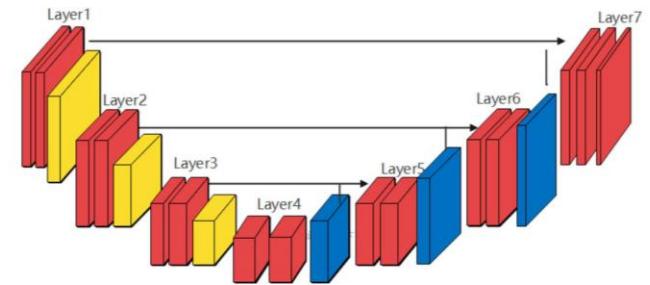
        noisy = a_bar.sqrt().reshape(n, 1, 1, 1) * x0 + (1 - a_bar).sqrt().reshape(n, 1, 1, 1) * eta
        return noisy

    def backward(self, x, t):
        # Run each image through the network for each timestep t in the vector t.
        # The network returns its estimation of the noise that was added.
        return self.network(x, t)
```

$$\alpha_t := 1 - \beta_t$$

$$\bar{\alpha}_t := \prod_{s=1}^t \alpha_s$$

$$q(\mathbf{x}_t | \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t} \mathbf{x}_0, (1 - \bar{\alpha}_t) \mathbf{I})$$



# DDPM Implementation

```

# Unet class
class MyUNet(nn.Module):
    def __init__(self, n_channels=3, n_steps=500, time_emb_dim=256):
        super(MyUNet, self).__init__()

        # Sinusoidal embedding
        self.time_embed = nn.Embedding(n_steps, time_emb_dim)
        self.time_embed.weight.data = sinusoidal_embedding(n_steps, time_emb_dim)
        self.time_embed.requires_grad_(False)

        # First half
        self.input_conv = MyBlock((n_channels, 32, 32), n_channels, 64)
        self.down1 = nn.Sequential(
            nn.MaxPool2d(2),
            MyBlock((64, 16, 16), 64, 64),
            MyBlock((64, 16, 16), 64, 128)
        )
        self.te1 = self._make_te(time_emb_dim, 64)
        self.sa1 = MyTESA(128, 16)
        self.down2 = nn.Sequential(
            nn.MaxPool2d(2),
            MyBlock((128, 8, 8), 128, 128),
            MyBlock((128, 8, 8), 128, 256)
        )
        self.te2 = self._make_te(time_emb_dim, 128)
        self.sa2 = MyTESA(256, 8)
        self.down3 = nn.Sequential(
            nn.MaxPool2d(2),
            MyBlock((256, 4, 4), 256, 256),
            MyBlock((256, 4, 4), 256, 256)
        )
        self.te3 = self._make_te(time_emb_dim, 256)
        self.sa3 = MyTESA(256, 4)
        self.bottleneck = nn.Sequential(
            MyBlock((256, 4, 4), 256, 512),
            MyBlock((512, 4, 4), 512, 512),
            MyBlock((512, 4, 4), 512, 256)
        )

        # Second half
        self.up = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)
        self.up1_conv = nn.Sequential(
            MyBlock((512, 8, 8), 512, 256),
            MyBlock((256, 8, 8), 256, 128)
        )
        self.te4 = self._make_te(time_emb_dim, 512)
        self.sa4 = MyTESA(128, 8)
        self.up2_conv = nn.Sequential(
            MyBlock((256, 16, 16), 256, 128),
            MyBlock((128, 16, 16), 128, 64)
        )
        self.te5 = self._make_te(time_emb_dim, 256)
        self.sa5 = MyTESA(64, 16)
        self.up3_conv = nn.Sequential(
            MyBlock((128, 32, 32), 128, 64),
            MyBlock((64, 32, 32), 64, 64)
        )
        self.te6 = self._make_te(time_emb_dim, 128)
        self.sa6 = MyTESA(64, 32)
        self.out_conv = nn.Conv2d(64, n_channels, (1, 1))
    
```

# DDPM Implementation

$$\mu_{\theta}(x_t, t) = \frac{1}{\sqrt{\alpha_t}} \left( x_t - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_{\theta}(x_t, t) \right)$$



```
# Generate new image samples for a given DDPM model, a given number of samples to be generated and a given device
def generate_new_images(ddpm, n_samples=16, device=None, frames_per_gif=100, gif_name="sampling.gif", c=3, h=32, w=32):
    frame_idxs = np.linspace(0, ddpm.n_steps, frames_per_gif).astype(np.uint)
    frames = []

    with torch.no_grad():
        if device is None:
            device = ddpm.device

        # Starting from random noise
        x = torch.randn(n_samples, c, h, w).to(device)

        for idx, t in enumerate(list(range(ddpm.n_steps))[::-1]):
            # Estimating noise to be removed
            time_tensor = (torch.ones(n_samples, 1)*t).to(device).long()
            eta_theta = ddpm.backward(x, time_tensor)
            alpha_t = ddpm.alphas[t]
            alpha_t_bar = ddpm.alpha_bars[t]

            # Partially denoising the image
            x = (1/alpha_t.sqrt())*(x-(1-alpha_t)/(1-alpha_t_bar).sqrt()*eta_theta)

            if t>0:
                z = torch.randn(n_samples, c, h, w).to(device)

                # sigma_t squared = beta_t
                beta_t = ddpm.betas[t]
                sigma_t = beta_t.sqrt()

                # Adding some more noise like in Langevin Dynamics fashion
                x = x + sigma_t * z

    frames = [x[i].cpu().permute(1, 2, 0).numpy() for i in frame_idxs]
```

# Challenges

---

- Adjusting the UNet model size for the "Flowers102" dataset
  - The available GPU had a memory of 4GB, so could not use a larger network
  - The size of the generated image had to be limited at 32x32x3
- Understanding the importance of self-attestation in UNet architecture
- Calculating the size of input to the hidden layers
- Tuning the hyper-parameters

# Training Parameters

---

Dataset – Fashion MNIST

Batch Size = 28

Learning Rate = 0.001

Epoch = 20

Number of steps = 1000

Min Beta = 0.0001

Max Beta = 0.02

Dataset – Flower 102

Batch Size = 2

Learning Rate = 0.0002

Epoch = 10000

Number of steps = 500

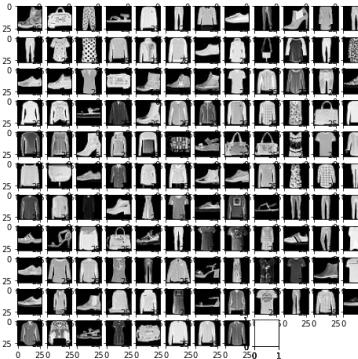
Min Beta = 0.0001

Max Beta = 0.02

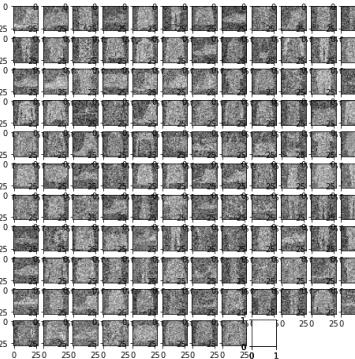
Dataset	Total Parameters	Trainable Parameters	GPU	Training Time	Best Model Loss
Fashion MNIST	20722881	20594881	Quadro T2000	~3hrs	
Flowers 102	20728259	20600259		~100hrs	0.015

# Generated Results – Fashion MNIST

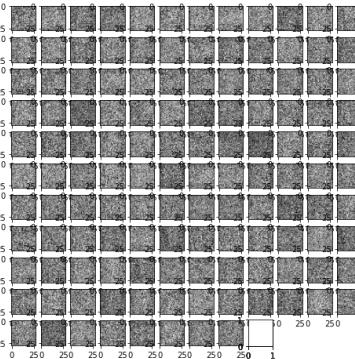
## Original images



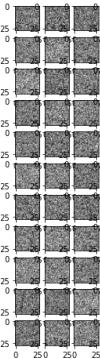
DDPM Noisy images 25%



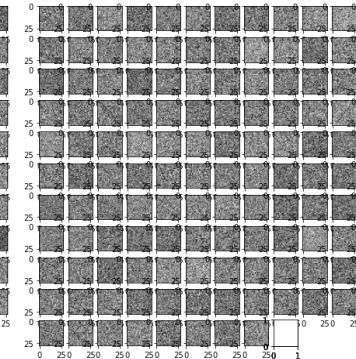
DDPM Noisy images 50%



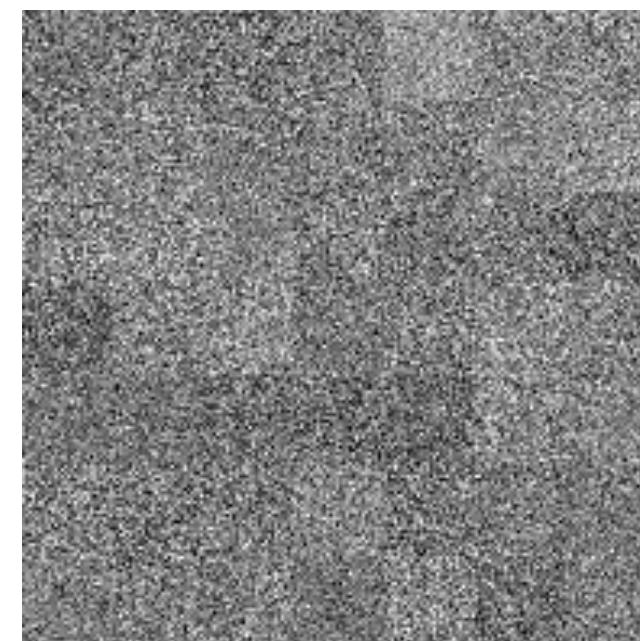
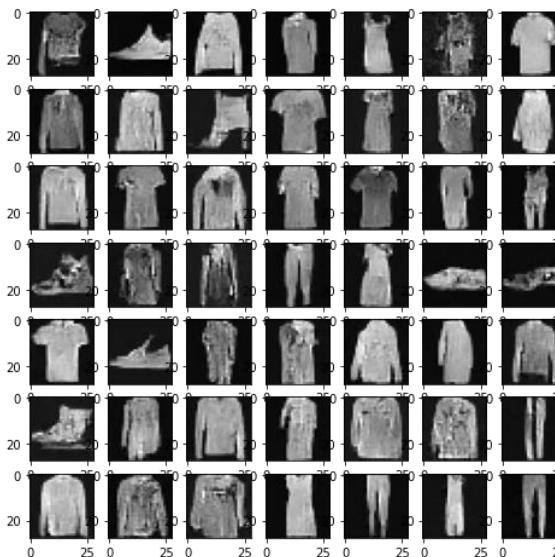
DDPM Noisy images 75%



DDPM Noisy images 100

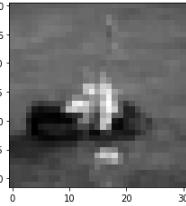
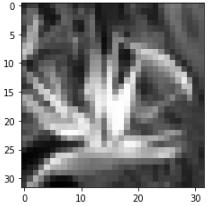


## Final result

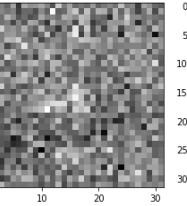
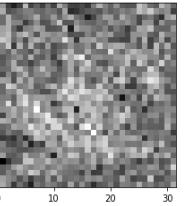


# Generated Results – Flower 102

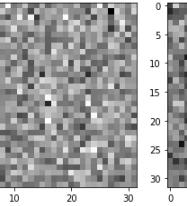
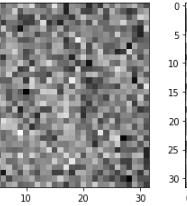
Original images



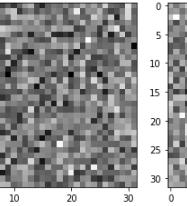
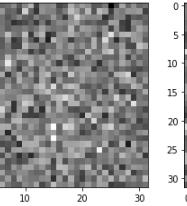
DDPM Noisy images 25%



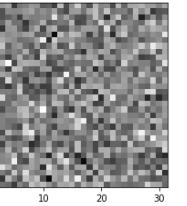
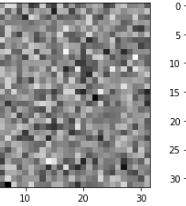
DDPM Noisy images 50%



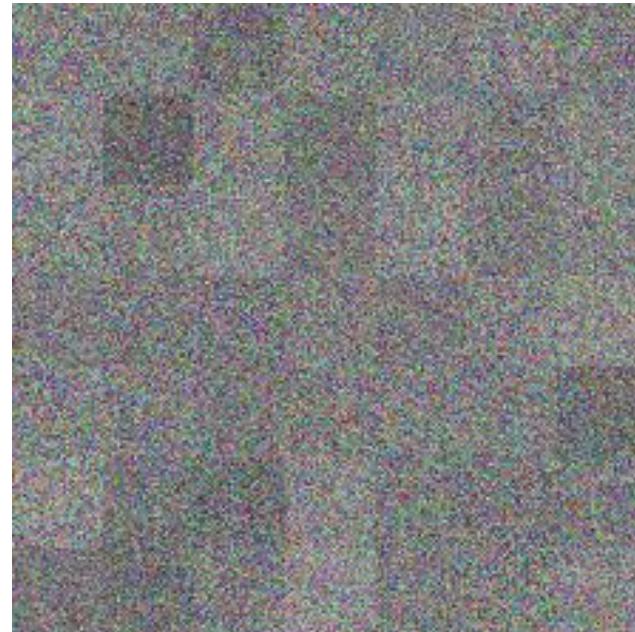
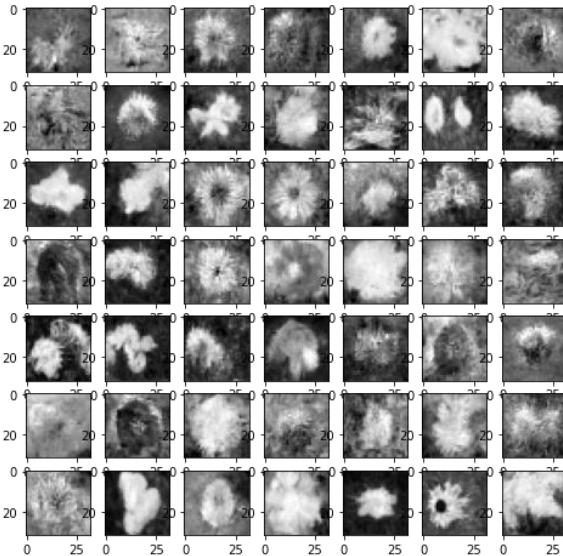
DDPM Noisy images 75%



DDPM Noisy images 100%



Final result



# Learnings & Future Work

---

## Learnings

- Theory behind DDPM and its use in famous image synthesis algorithms such as DALL-E / ImageGen.
- It is difficult to have a generalized model for all the datasets.
- Tuning the hyper-parameters is difficult and time consuming.
- It is a challenge to train larger networks on local machine (with limited hardware resources).

## Future Work

- Train the implemented model on a mixed dataset and see how is the performance on such a set.
- Incorporate text embedding to have text-to-image synthesis model