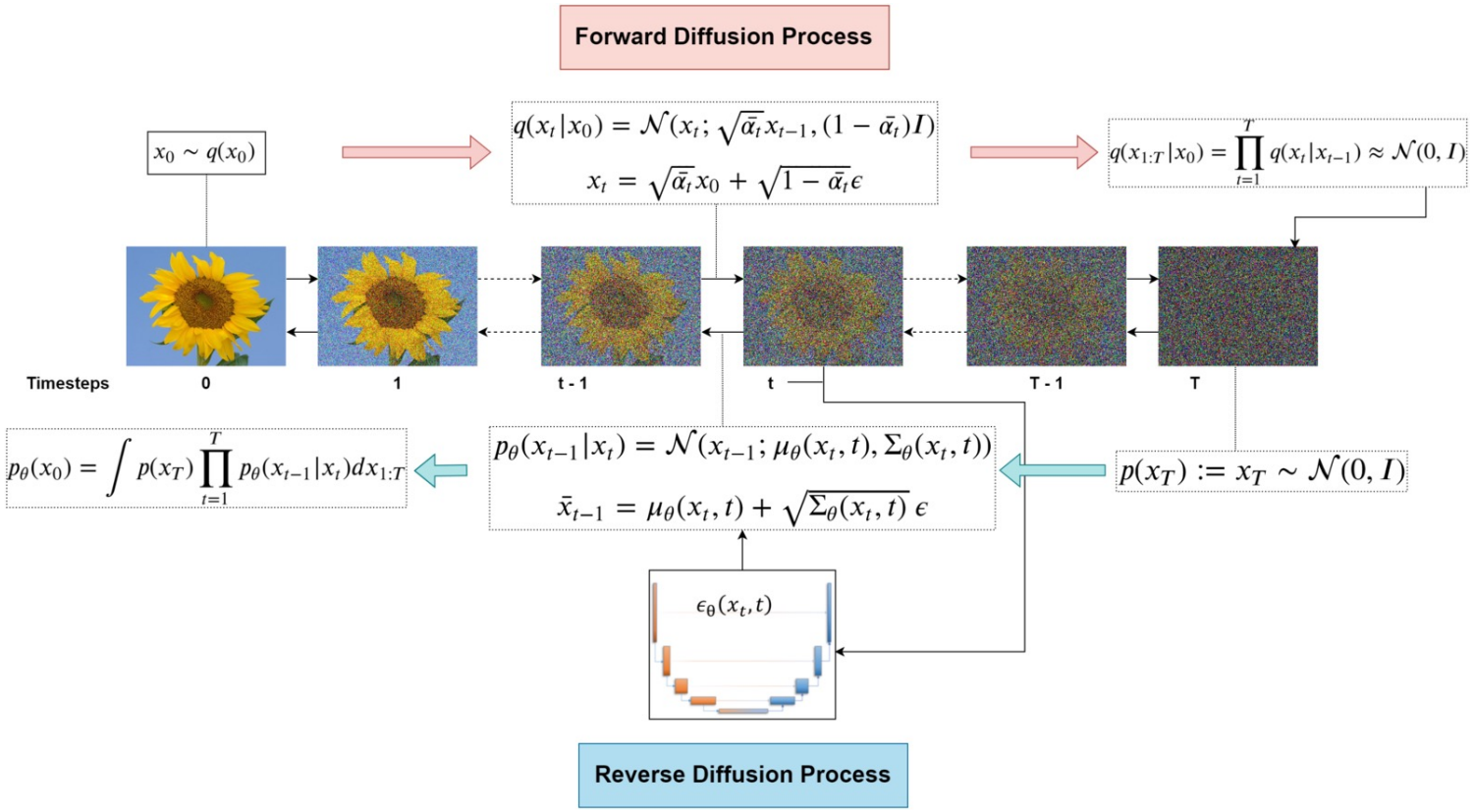
AI VIETNAM
All-in-One Course

Denoising Diffusion Probabilistic Model



Denoising Diffusion Probabilistic Models

Jonathan Ho
UC Berkeley
jonathanho@berkeley.edu

Ajay Jain
UC Berkeley
ajayj@berkeley.edu

Pieter Abbeel
UC Berkeley
pabbeel@cs.berkeley.edu

Abstract

We present high quality image synthesis results using diffusion probabilistic models, a class of latent variable models inspired by considerations from nonequilibrium thermodynamics. Our best results are obtained by training on a weighted variational bound designed according to a novel connection between diffusion probabilistic models and denoising score matching with Langevin dynamics, and our models naturally admit a progressive lossy decompression scheme that can be interpreted as a generalization of autoregressive decoding. On the unconditional CIFAR10 dataset, we obtain an Inception score of 9.46 and a state-of-the-art FID score of 3.17. On 256x256 LSUN, we obtain sample quality similar to ProgressiveGAN. Our implementation is available at `https://github.com/hojonathanho/diffusion`.

1 Introduction

Deep generative models of all kinds have recently exhibited high quality samples in a wide variety of data modalities. Generative adversarial networks (GANs), autoregressive models, flows, and variational autoencoders (VAEs) have synthesized striking image and audio samples [14, 27, 3, 58, 38, 25, 10, 32, 44, 57, 26, 33, 45], and there have been remarkable advances in energy-based modeling and score matching that have produced images comparable to those of GANs [11, 55].

Vinh Dinh Nguyen
PhD in Computer Science

Outline

- Objective
- What is Denoising Probability Diffusion Model
- Forward Diffusion Process: Review
- Reverse Diffusion Process: Explain and Implementation
- Denoise Probability Diffusion Model: Implementation
- Summary

Outline

- Objective
- What is Denoising Probability Diffusion Model
- Forward Diffusion Process: Review
- Reverse Diffusion Process: Explain and Implementation
- Denoise Probability Diffusion Model: Implementation
- Summary

Objective

Denoising Diffusion Probabilistic Models

Jonathan Ho
UC Berkeley
jonathanho@berkeley.edu

Ajay Jain
UC Berkeley
ajayj@berkeley.edu

Pieter Abbeel
UC Berkeley
pabbeel@cs.berkeley.edu

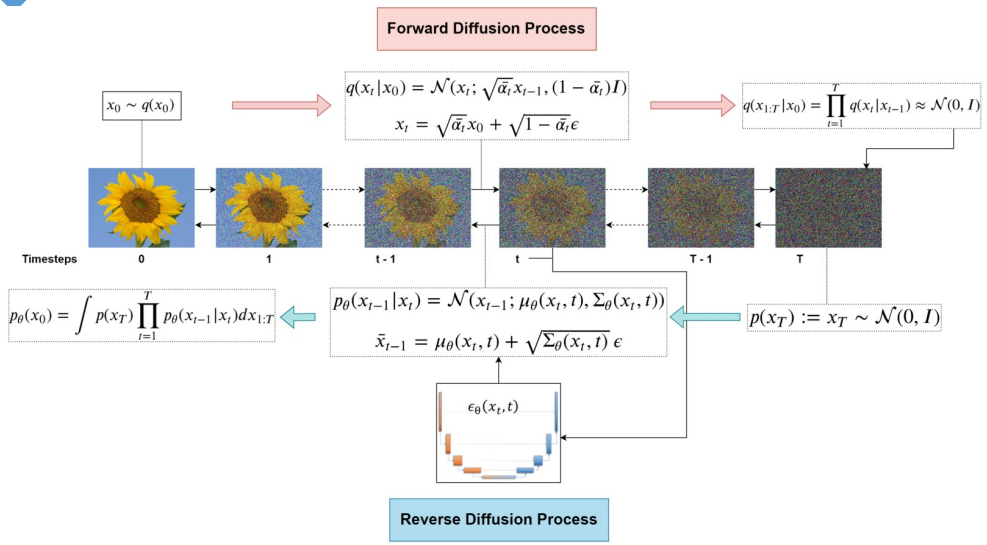
Abstract

We present high quality image synthesis results using diffusion probabilistic models, a class of latent variable models inspired by considerations from nonequilibrium thermodynamics. Our best results are obtained by training on a weighted variational bound designed according to a novel connection between diffusion probabilistic models and denoising score matching with Langevin dynamics, and our models naturally admit a progressive lossy decompression scheme that can be interpreted as a generalization of autoregressive decoding. On the unconditional CIFAR10 dataset, we obtain an Inception score of 9.46 and a state-of-the-art FID score of 3.17. On 256x256 LSUN, we obtain sample quality similar to ProgressiveGAN. Our implementation is available at `https://github.com/hojonathanho/diffusion`.

1 Introduction

Deep generative models of all kinds have recently exhibited high quality samples in a wide variety of data modalities. Generative adversarial networks (GANs), autoregressive models, flows, and variational autoencoders (VAEs) have synthesized striking image and audio samples [14, 27, 3, 58, 38, 25, 10, 32, 44, 57, 26, 33, 45], and there have been remarkable advances in energy-based modeling and score matching that have produced images comparable to those of GANs [11, 55].

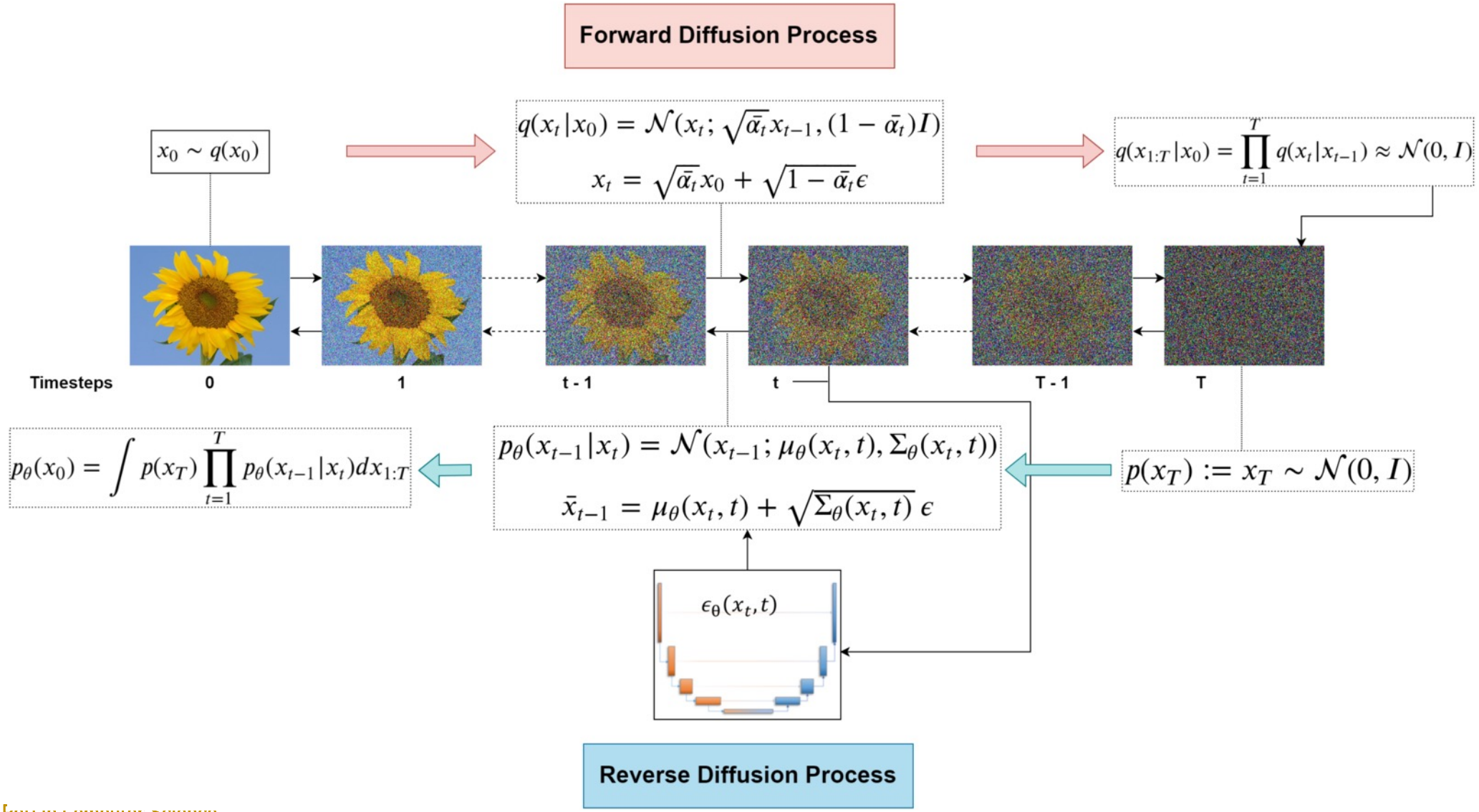
1. Understand a Probability Diffusion Model
2. Understand a Forward Diffusion Process
3. Understand a Reverse Diffusion Process
4. Be able to Implement a Denoising Diffusion Model Using Pytorch



Outline

- Objective
- What is Denoising Probability Diffusion Model
- Forward Diffusion Process: Review
- Reverse Diffusion Process: Explain and Implementation
- Denoise Probability Diffusion Model: Implementation
- Summary

Denoise Probability Diffusion Model

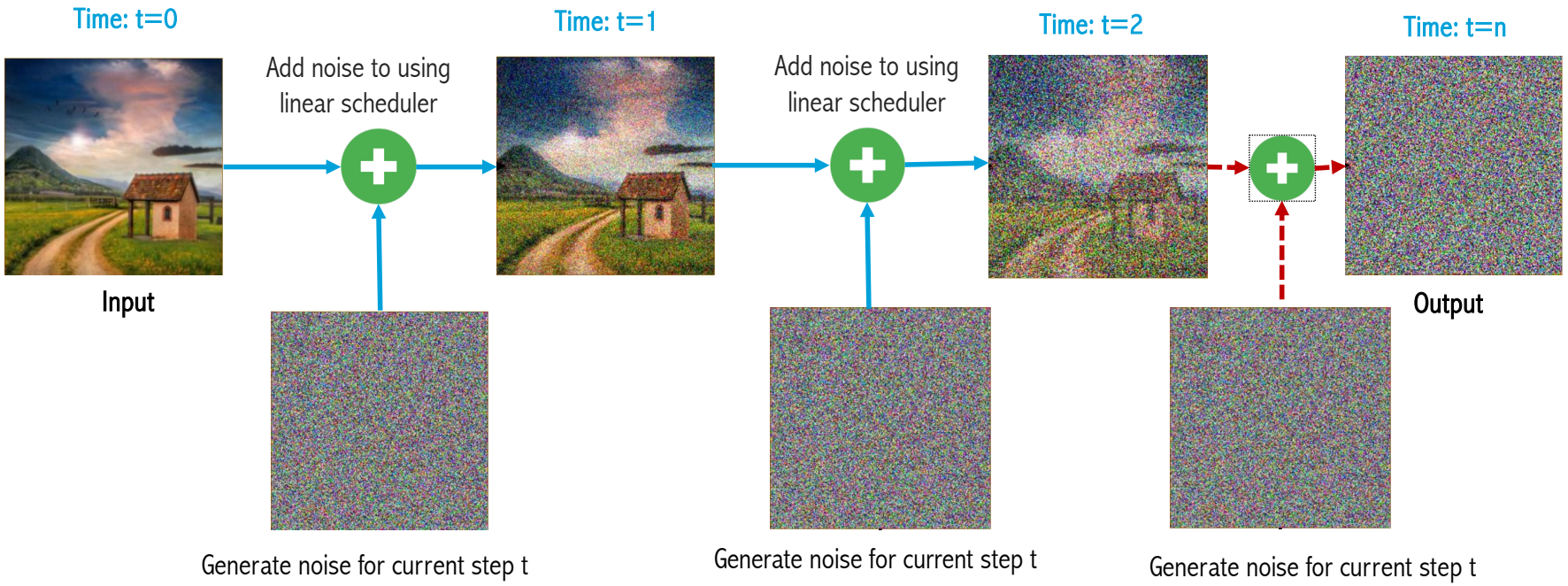


Outline

- Objective
- What is Denoising Probability Diffusion Model
- Forward Diffusion Process: Review
- Reverse Diffusion Process: Explain and Implementation
- Denoise Probability Diffusion Model: Implementation
- Summary

Forward Diffusion Process: Review

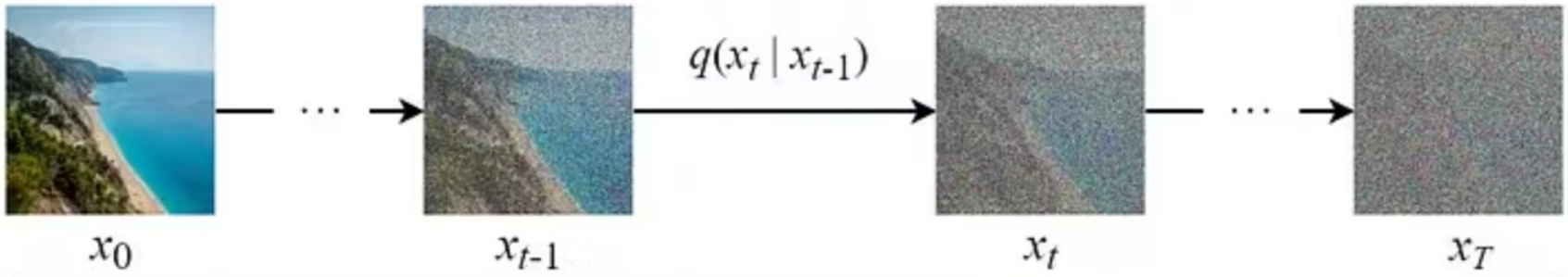
$$q(x_t|x_{t-1}) = \mathcal{N}(x_t, \sqrt{1-\beta_t}x_{t-1}, \beta_t I)$$



The forward diffusion method progressively adds Gaussian noise to the input picture x0, for a total of T steps. The technique will generate a series of noisy picture samples x1,..., x T.

When T is large, the resultant image will be fully noisy, as if it were sampled from an isotropic Gaussian distribution.

Forward Diffusion Process: Review



Distribution of the noised images — Output — Mean μ_t — Variance Σ_t

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I)$$

Notations:

- t : time step (from 0 to T)
- x_0 : a data sampled from the real data distrition $q(x)$ (i.e. $x_0 \sim q(x)$)
- β_t : variance schedule ($0 \leq \beta_t \leq 1$, and β_0 = small number, β_T = large number)
- I : identity matrix

If $z \sim \mathcal{N}(\mu, \sigma^2)$ then $z = \mu + \sigma\varepsilon$ where $\varepsilon \sim \mathcal{N}(0,1)$

Instead of creating an algorithm to add noise to the picture repeatedly, we may use a closed-form formula to directly sample a noisy image at a specified time step t.

$$x_t = \sqrt{1-\beta_t}\, x_{t-1} + \sqrt{\beta_t}\, \varepsilon_{t-1}$$

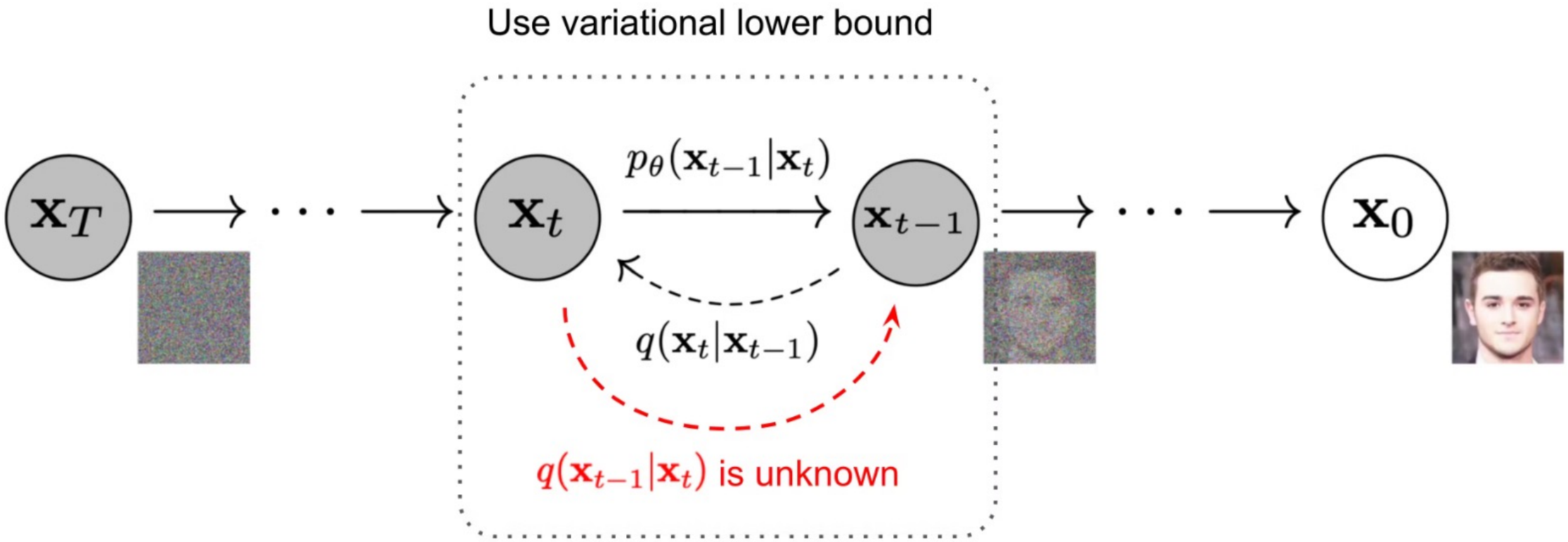
$$\varepsilon \sim \mathcal{N}(0, I)$$
$$\alpha_t = 1 - \beta_t$$
$$\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$$

$$x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1-\bar{\alpha}_t}\, \varepsilon$$

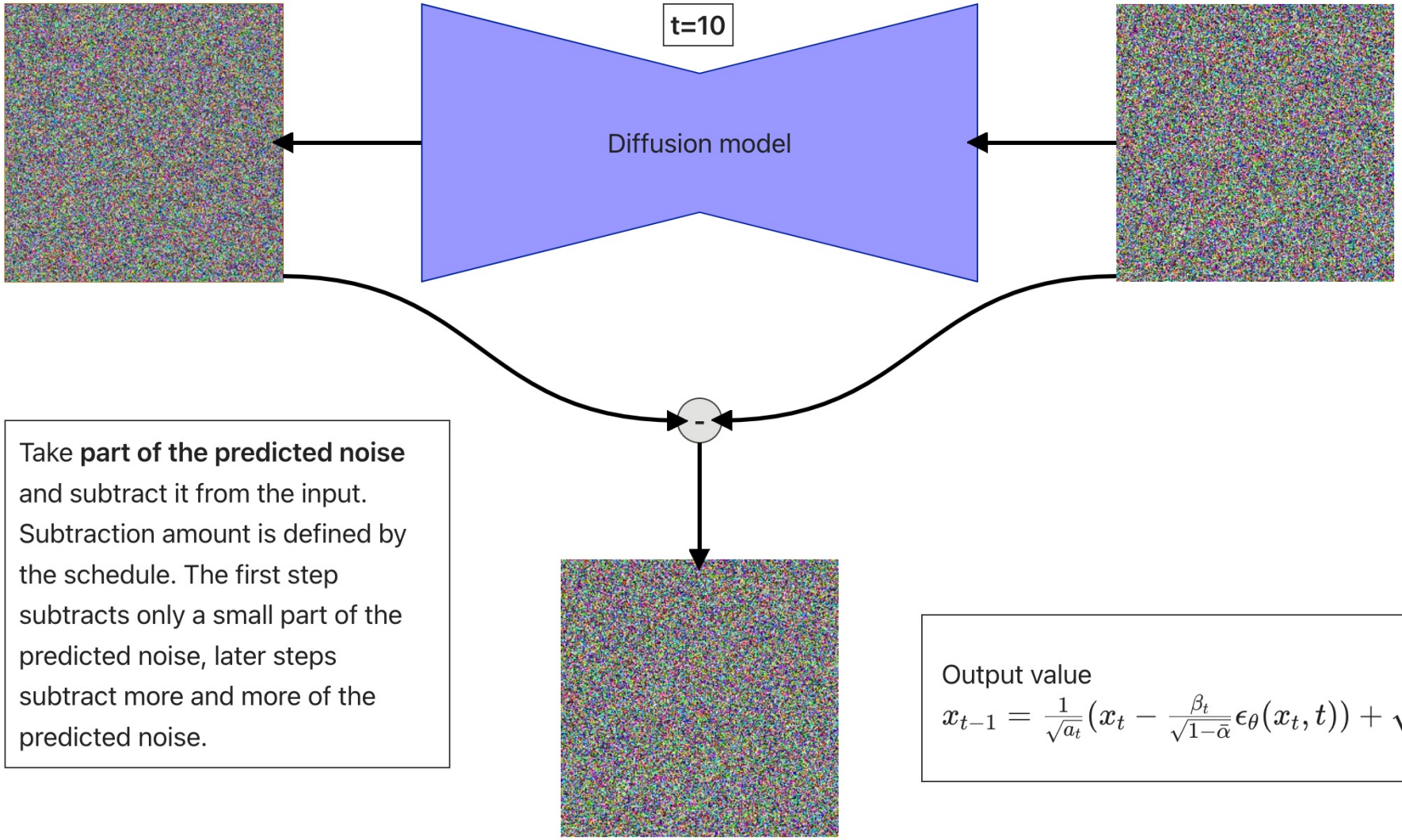
Outline

- Objective
- What is Denoising Probability Diffusion Model
- Forward Diffusion Process: Review
- Reverse Diffusion Process: Explain and Implementation
- Denoise Probability Diffusion Model: Implementation
- Summary

Reverse Diffusion Process



Reverse Diffusion Process

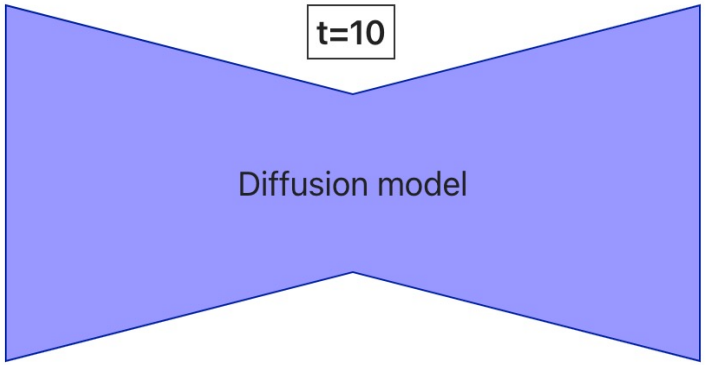


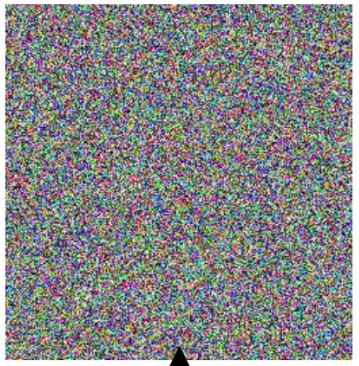
Take **part of the predicted noise** and subtract it from the input. Subtraction amount is defined by the schedule. The first step subtracts only a small part of the predicted noise, later steps subtract more and more of the predicted noise.

Output value

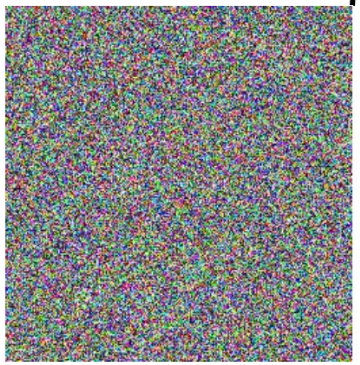
$$x_{t-1} = \frac{1}{\sqrt{a_t}}(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}}}\epsilon_\theta(x_t, t)) + \sqrt{\beta_t}\epsilon$$

Reverse Diffusion Process





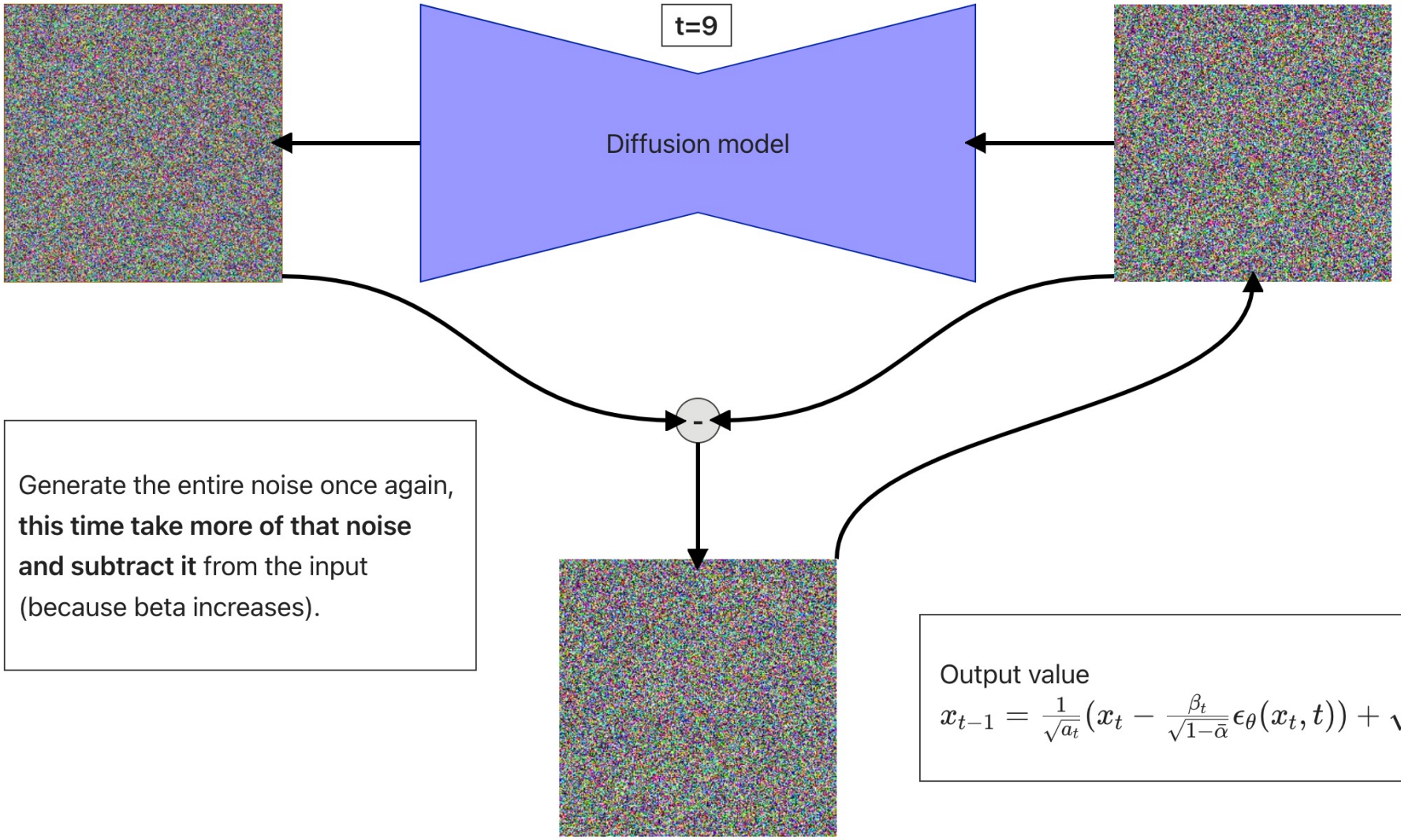
Output from the **t** timestep becomes an input in **t-1**.



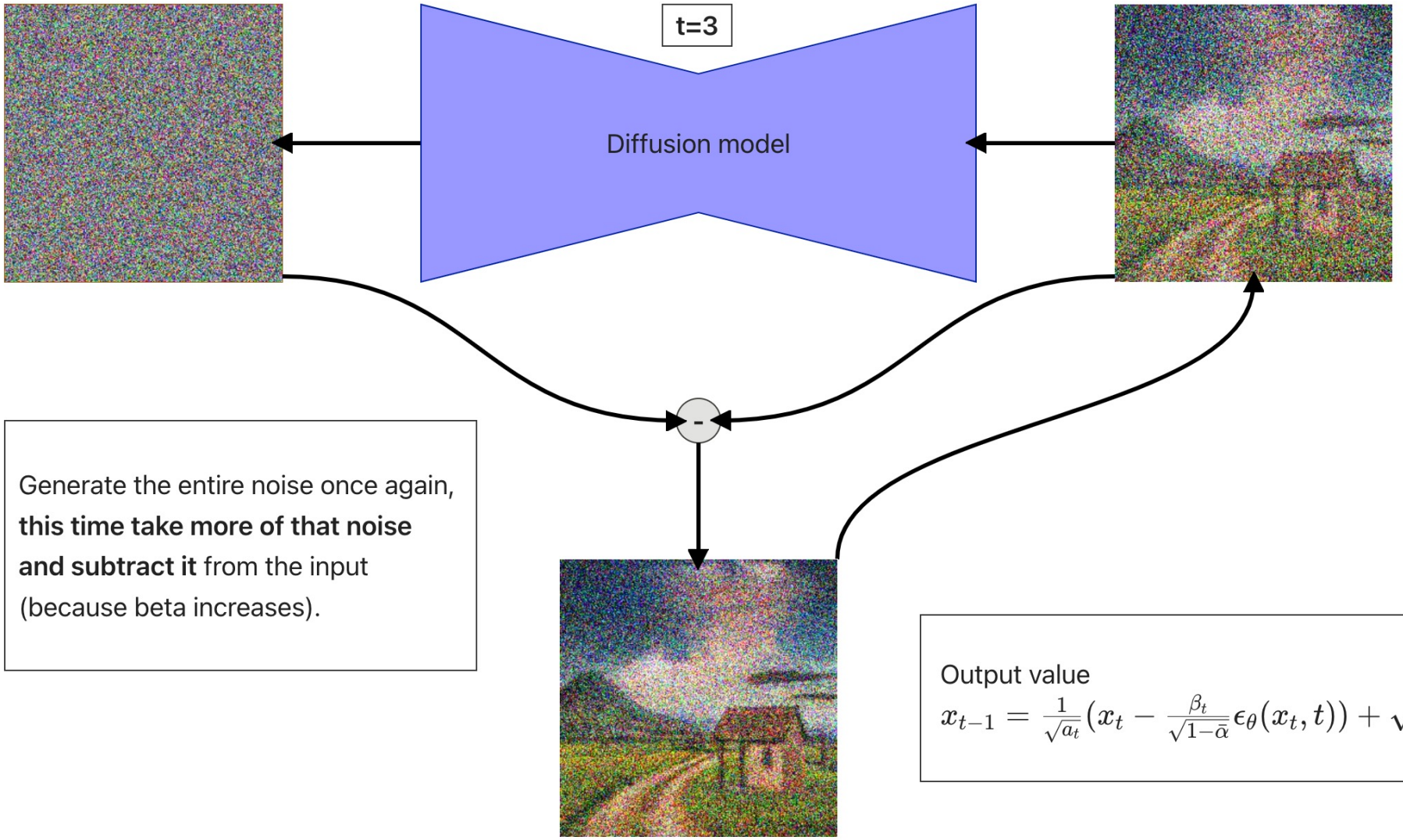
Output value

$$x_{t-1} = \frac{1}{\sqrt{a_t}}(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}}}\epsilon_\theta(x_t, t)) + \sqrt{\beta_t}\epsilon$$

Reverse Diffusion Process



Reverse Diffusion Process

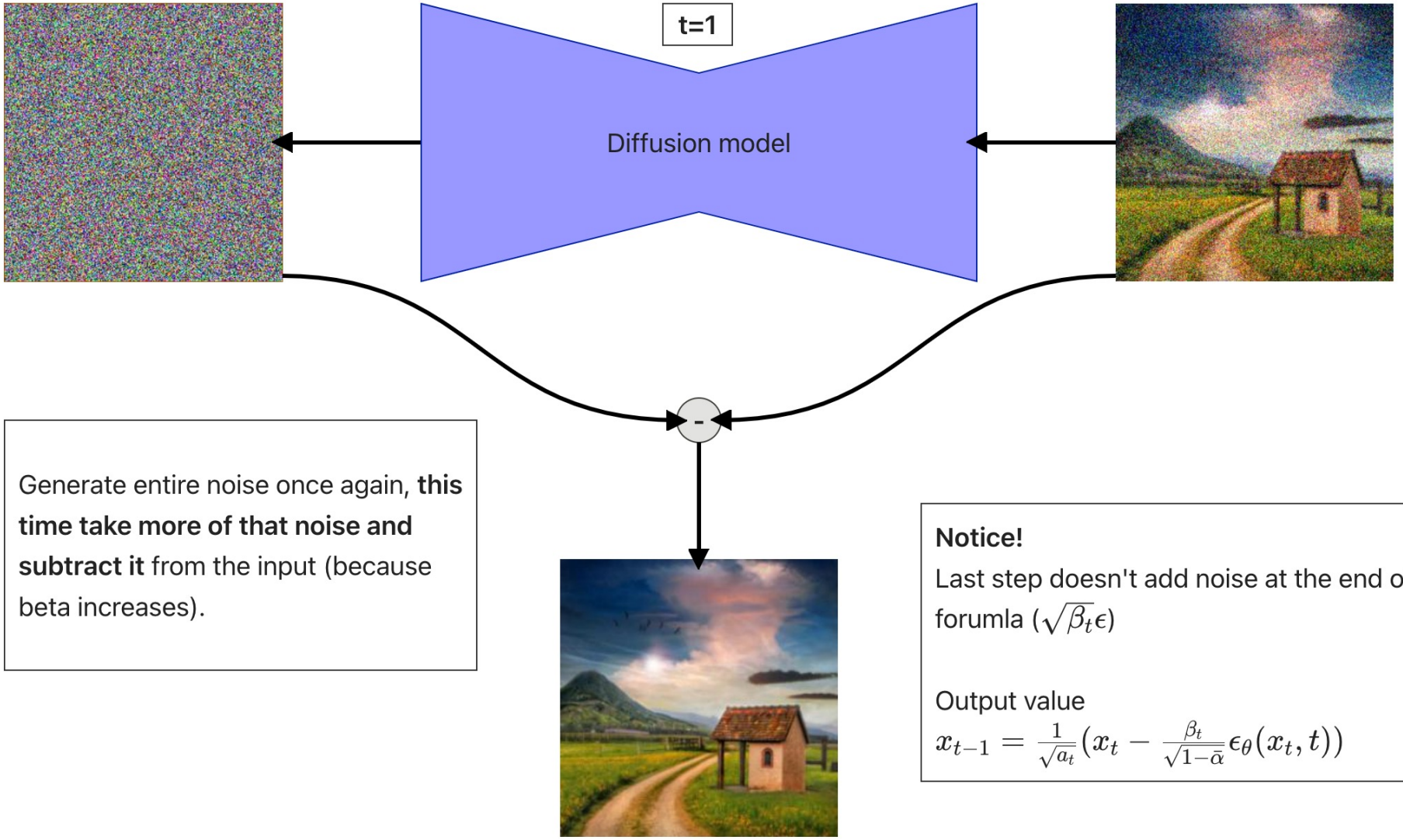


Generate the entire noise once again, **this time take more of that noise and subtract it** from the input (because beta increases).

Output value

$$x_{t-1} = \frac{1}{\sqrt{a_t}}(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}}}\epsilon_\theta(x_t, t)) + \sqrt{\beta_t}\epsilon$$

Reverse Diffusion Process



Generate entire noise once again, **this time take more of that noise and subtract it** from the input (because beta increases).

Notice!
Last step doesn't add noise at the end of the forumla ($\sqrt{\beta_t}\epsilon$)

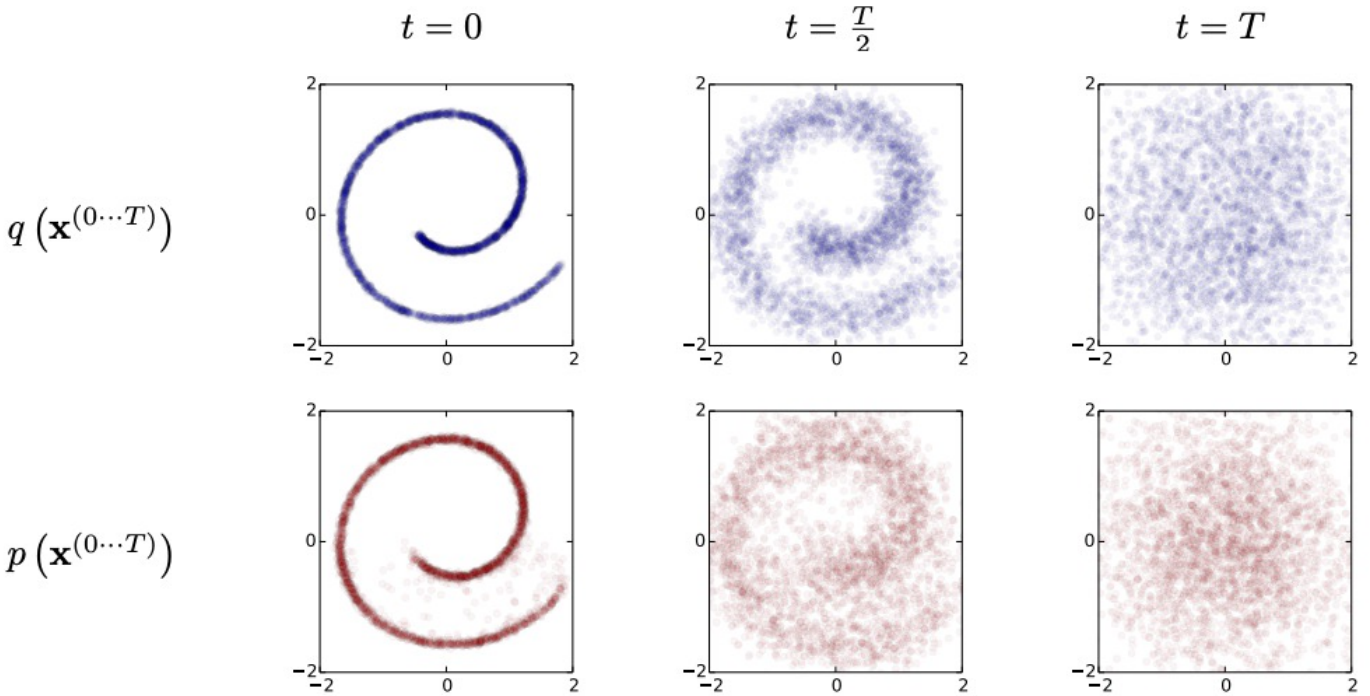
Output value
$x_{t-1} = \frac{1}{\sqrt{a_t}}(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}}}\epsilon_\theta(x_t, t))$

Reverse Diffusion Process

$$p_\theta(x_{0:T}) := p(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t)$$

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}, \mu_\theta(x_t, t), \sum_\theta(x_t, t))$$

- $\mu_\theta(x_t, t)$ (mean)
- $\sum_\theta(x_t, t)$ which equals $\sigma_t^2 I$ (variance)



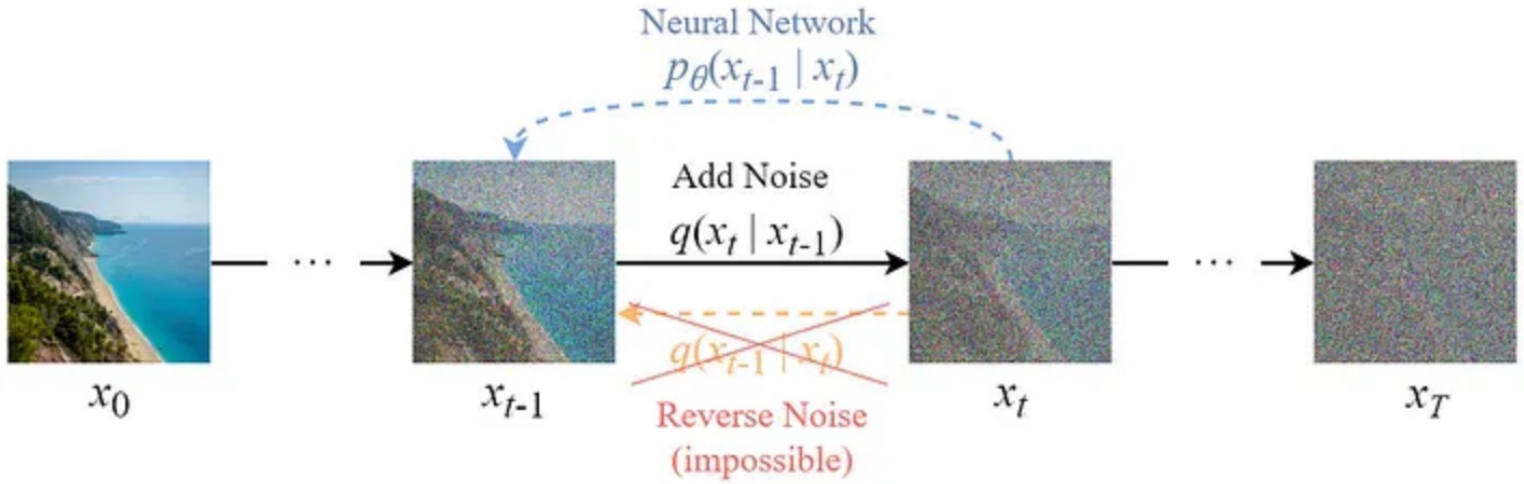
$$\mu_\theta(x_t, t) = \frac{1}{\sqrt{\alpha_t}}(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t))$$

$$x_{t-1} = \mathcal{N}(x_{t-1}, \frac{1}{\sqrt{\alpha_t}}(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)), \sqrt{\beta_t}\epsilon)$$

$$x_{t-1} = \frac{1}{\sqrt{a_t}}(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}}}\epsilon_\theta(x_t, t)) + \sqrt{\beta_t}\epsilon$$

$\epsilon_\theta(x_t, t)$ is our **model's output** (predicted noise)

Reverse Diffusion Process



Target Distribution $q(x_{t-1}|x_t) = \mathcal{N}(x_{t-1};\ \tilde{\mu}_t(x_t, x_0),\ \tilde{\beta}_t I)$

Approximated Distribution $p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1};\ \mu_\theta(x_t, t),\ \Sigma_\theta(x_t, t))$

Learnable parameters
(Neural Network)

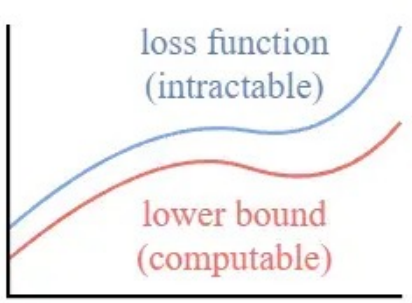
Unlike the forward process, we cannot use $q(x_{t-1} \mid x_t)$ to reverse the noise since it is intractable (uncomputable).

Train a neural network $p\theta(x_{t-1} \mid x_t)$ to approximate $q(x_{t-1} \mid x_t)$. The approximation $p\theta(x_{t-1} \mid x_t)$ follows a normal distribution and its mean and variance are set as follows:

$$\begin{cases} \mu_\theta(x_t, t) & := \tilde{\mu}_t(x_t, x_0) \\ \Sigma_\theta(x_t, t) & := \tilde{\beta}_t I \end{cases}$$

Loss Function

Define a loss as a Negative Log-Likelihood : $\text{Loss} = -\log(p_\theta(x_0))$ — Depends on $x_1, x_2, ..., x_T$ Therefore it is intractable!



Instead of optimizing the intractable loss function itself, we can optimize the Variational Lower Bound.

$$-\log p_\theta(x_0) \leq -\log p_\theta(x_0) + D_{\text{KL}}(q(x_{1:T}|x_0)\|p_\theta(x_{1:T}|x_0))$$

$$\vdots$$

$$-\log p_\theta(x_0) \leq \mathbb{E}_q\left[\log \frac{q(x_{1:T}|x_0)}{p_\theta(x_{0:T})}\right]$$ Variational Lower Bound

$$\vdots$$

$$-\log p_\theta(x_0) \leq \mathbb{E}_q[\underbrace{D_{\text{KL}}(q(x_T|x_0) \parallel p_\theta(x_T))}_{L_T} + \sum_{t=2}^{T} \underbrace{D_{\text{KL}}(q(x_{t-1}|x_t, x_0) \parallel p_\theta(x_{t-1}|x_t))}_{L_{t-1}} \underbrace{-\log p_\theta(x_0|x_1)}_{L_0}]$$

① ② ③

This term compares the target denoising step q and the approximated denoising step pθ.

- No learnable parameters
- Just a Gaussian noise

Constant ⇩ Ignorable

Stepwise denoising term

Reconstruction term

Loss Function

$$-\log p_\theta(x_0) \le \mathbb{E}_q\Big[\underbrace{D_{\mathrm{KL}}(q(x_T|x_0) \,\|\, p_\theta(x_T))}_{L_T} + \sum_{t=2}^{T} \underbrace{D_{\mathrm{KL}}(q(x_{t-1}|x_t, x_0) \,\|\, p_\theta(x_{t-1}|x_t))}_{L_{t-1}} \underbrace{-\log p_\theta(x_0|x_1)}_{L_0}\Big]$$

① ② ③

- No learnable parameters
- Just a Gaussian noise

Constant ⇩ Ignorable

Stepwise denoising term

Reconstruction term

$$\underbrace{D_{\mathrm{KL}}(q(x_{t-1}|x_t, x_0) \,\|\, p_\theta(x_{t-1}|x_t))}_{L_{t-1}}$$

$$q(x_{t-1}|x_t, x_0) = \mathcal{N}(x_{t-1}; \tilde{\boldsymbol{\mu}}(x_t, x_0), \tilde{\beta}_t I) \qquad p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \boldsymbol{\mu}_\theta(x_t, t), \beta_t I)$$

Neural Network

$$\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t} \cdot \beta_t$$

$$\tilde{\boldsymbol{\mu}}_t(x_t, x_0) = \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} x_t + \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t} x_0 \quad \text{where } x_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}(x_t - \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_t)$$

$$\vdots$$

$$\tilde{\boldsymbol{\mu}}_t(x_t) = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_t\right)$$

To approximate the target denoising step q, we only need to approximate its mean using a neural network. So we set the approximated mean μθ to be in the same form as the target mean $\tilde{\mu}_t$ (with a learnable neural network εθ):

$$\tilde{\boldsymbol{\mu}}_t(x_t) = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_t\right)$$

$$\text{Set} \quad \boldsymbol{\mu}_\theta(x_t, t) = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(x_t, t)\right)$$

Loss Function

The comparison between the target mean and the approximated mean can be done using a mean squared error (MSE):

$$
\begin{aligned}
L_t &= \mathbb{E}_{x_0,\varepsilon}\left[\frac{1}{2\sigma_t^2}\left\|\tilde{\boldsymbol{\mu}}_t(x_t) - \boldsymbol{\mu}_\theta(x_t,t)\right\|^2\right] \\
&= \mathbb{E}_{x_0,\varepsilon}\left[\frac{1}{2\sigma_t^2}\left\|\frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\varepsilon}_t\right) - \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\varepsilon}_\theta(x_t,t)\right)\right\|^2\right] \\
&= \mathbb{E}_{x_0,\varepsilon}\left[\frac{(1-\alpha_t)^2}{2\alpha_t(1-\bar{\alpha}_t)\sigma_t^2}\left\|\boldsymbol{\varepsilon}_t - \boldsymbol{\varepsilon}_\theta(x_t,t)\right\|^2\right]
\end{aligned}
$$

ignorable

$$
L_t^{\text{simple}} = \mathbb{E}_{t\sim[1,T],x_0,\boldsymbol{\varepsilon}_t}\left[\left\|\boldsymbol{\varepsilon}_t - \boldsymbol{\varepsilon}_\theta(x_t,t)\right\|^2\right]
$$

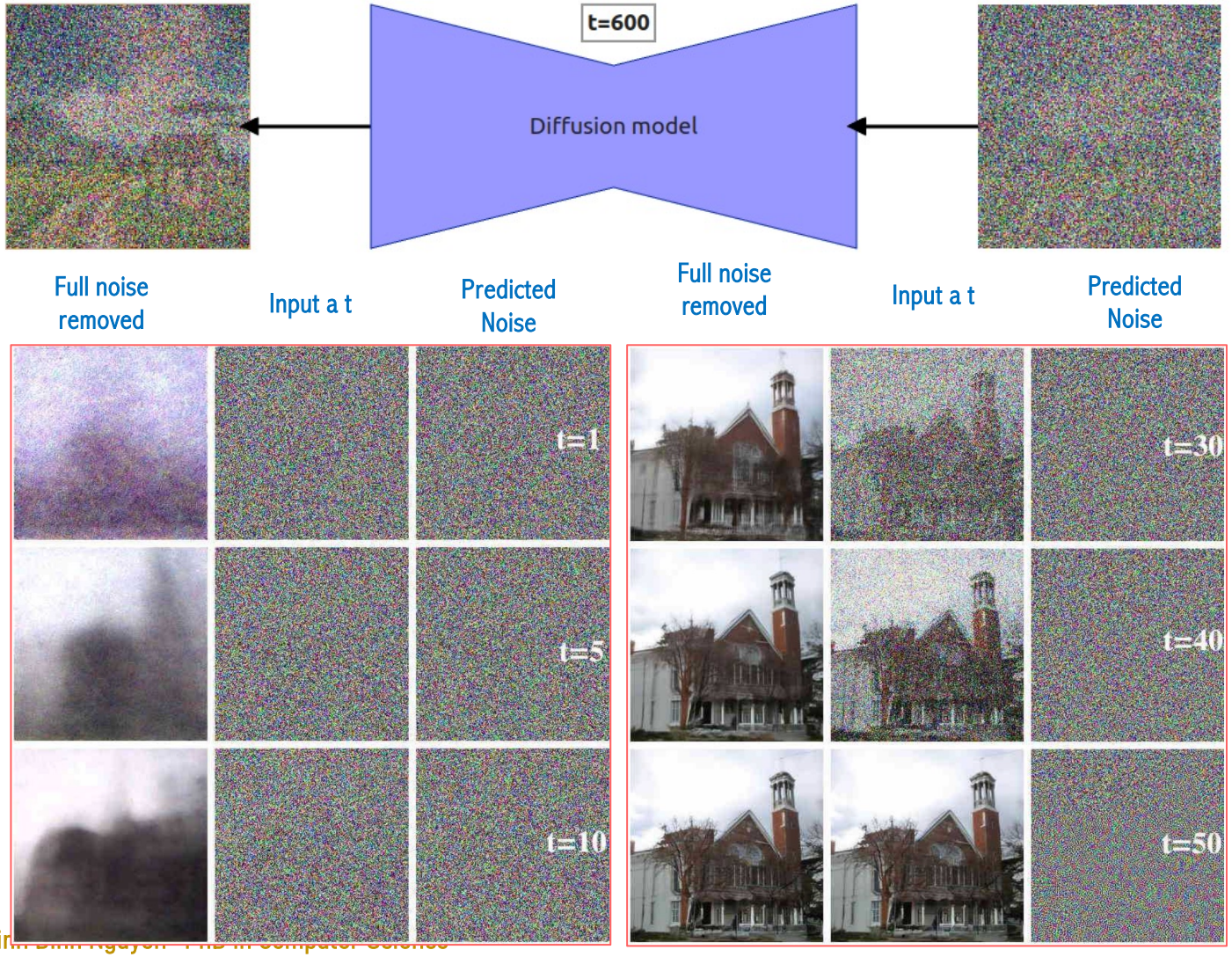
$$
x_t = \sqrt{\bar{\alpha}_t}\,x_0 + \sqrt{1-\bar{\alpha}_t}\,\varepsilon
$$

$$
L_{\text{simple}} = \mathbb{E}_{t,x_0,\boldsymbol{\varepsilon}}\left[\left\|\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}_\theta(x_t,t)\right\|^2\right]
$$

Experimentally, better results can be achieved by ignoring the weighting term and simply comparing the target and predicted noises with MSE.

So, it turns out that to approximate the desired denoising step q, we just need to approximate the noise ε_t using a neural network $\varepsilon\theta$.

Reverse Diffusion Output Visualization



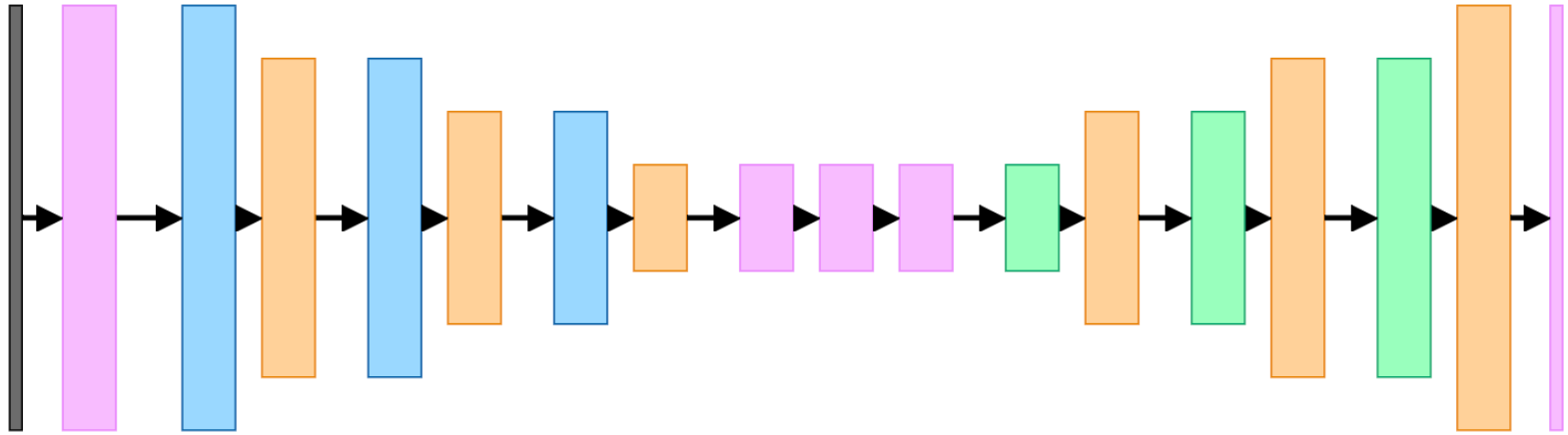
> ***Notice!*** *Because this is a reverse process when we say* ***t=1****, the value of the* β_t *is set to* β_{T-t+1}*, where* ***T*** *is the total number of steps. E.g. when t=1 we're using* β_{50} *for t=2 we're using* β_{49} *and so on.*

Each time we predict the noise using a neural network, we subtract part of it and move to the next step. That is how the diffusion process works. But **what will happen if we just subtract all the noise?**

- You can use fewer timesteps in your schedule when doing the inference after the model is trained.
- You can use a different schedule when doing the inference.

Diffusion Model Architecture

U-Net Architecture

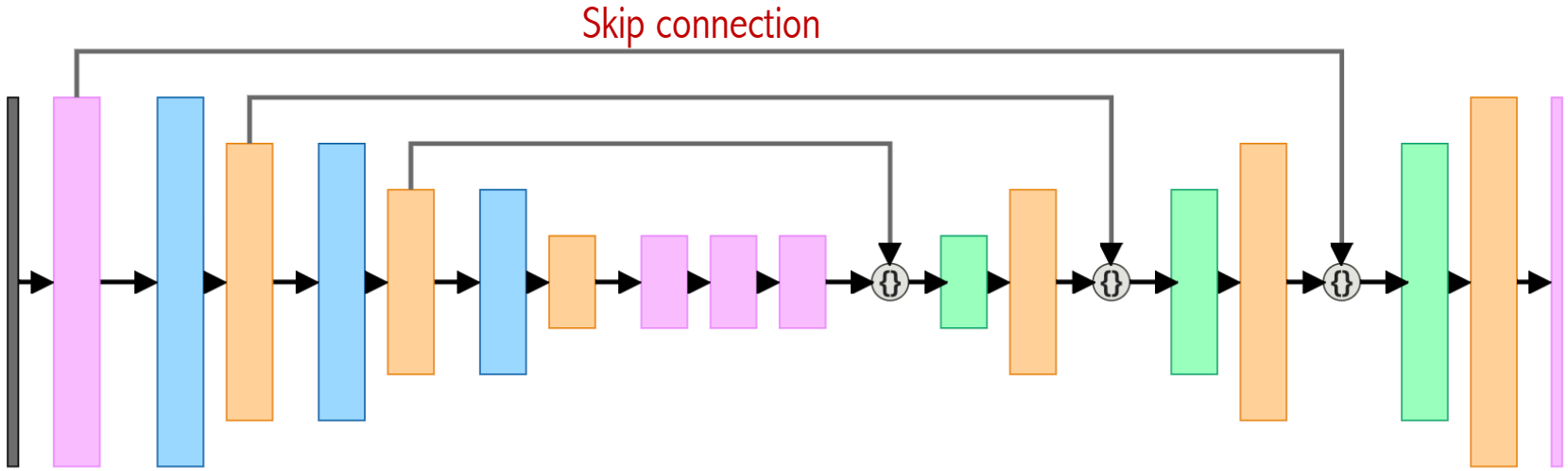


Input
Conv Block
Downsample Block
Self-Attention Block
Upsample Block
Embedding vector

We start with the modified **U-Net architecture**.
Basic ResNet Blocks were replaced by either
Self-Attention blocks or modified ResNet blocks.

Diffusion Model Architecture

U-Net Architecture

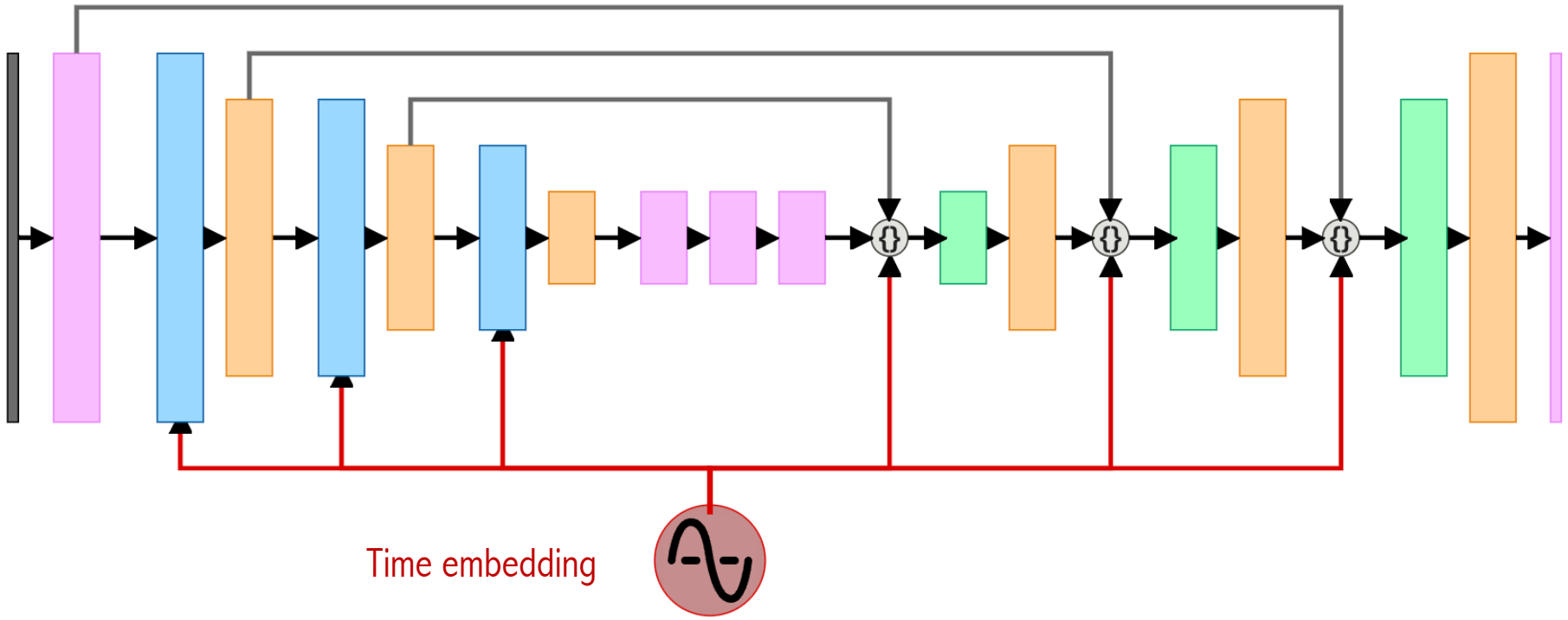


To prevent information lose, we're adding **skip connections to the upsampling blocks**. Notice where those connections are coming from.

Input
Conv Block
Downsample Block
Self-Attention Block
Upsample Block
Embedding vector

Diffusion Model Architecture

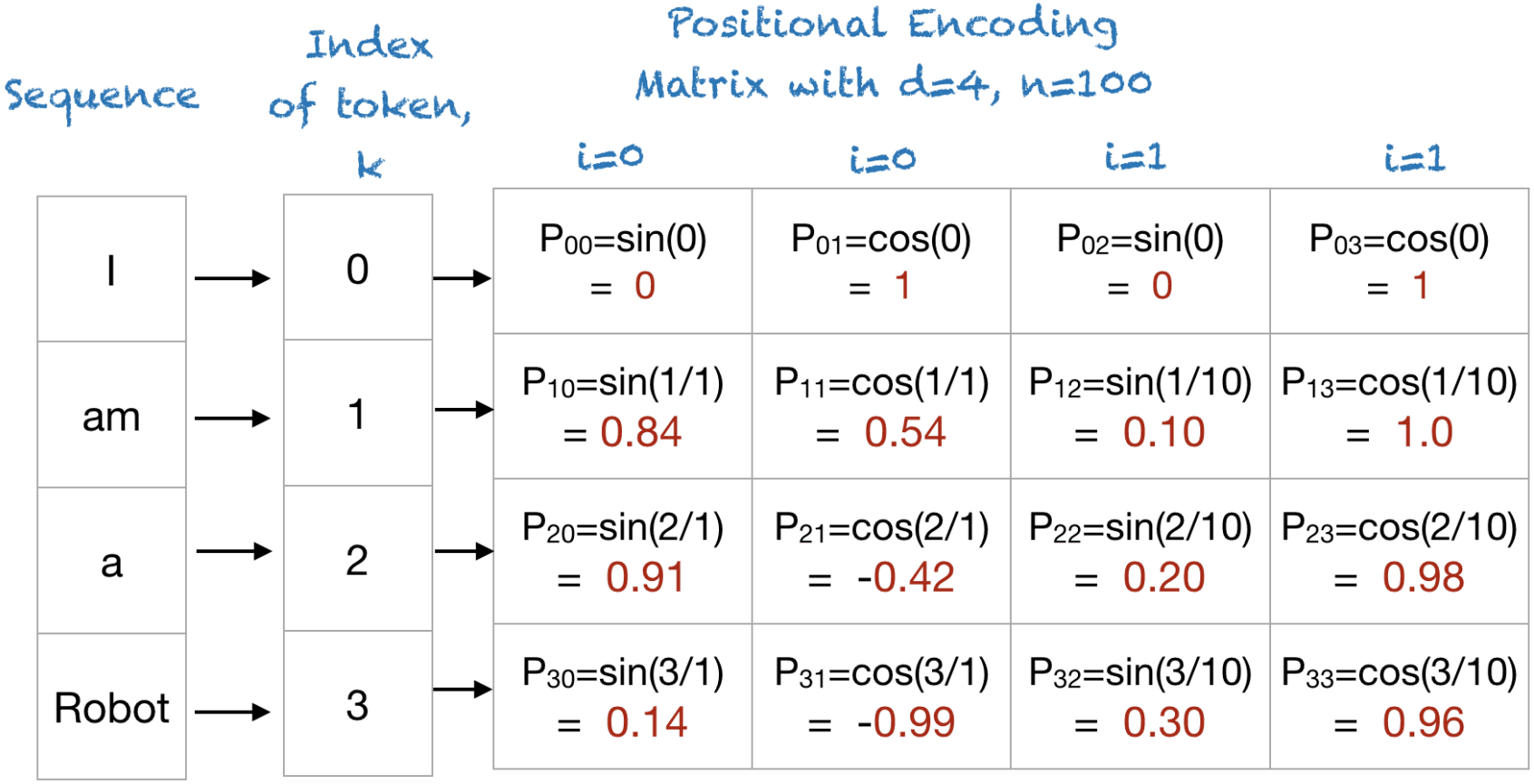
U-Net Architecture



Input
Conv Block
Downsample Block
Self-Attention Block
Upsample Block
Embedding vector

The next step is to add information about the current timestep **t**. To do that we're using **sinusoidal embedding** and that information is added to **all downsample and upsample** blocks.

Embedding: Positional encoding visualization



Positional Encoding Matrix for the sequence 'I am a robot'

U-Net Time Embedding: Implementation

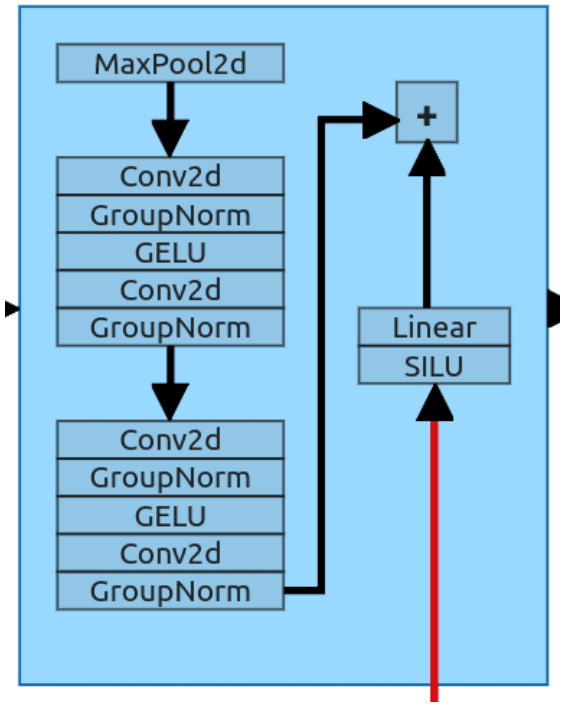
```python
def get_timestep_embedding(timesteps, embedding_dim: int):
    """
    Retrieved from https://github.com/hojonathanho/diffusion/blob/master/diffusion_tf/nn.py#LL90C1-L109C13
    """
    assert len(timesteps.shape) == 1

    half_dim = embedding_dim // 2
    emb = math.log(10000) / (half_dim - 1)
    emb = torch.exp(torch.arange(half_dim, dtype=torch.float32, device=timesteps.device) * -emb)
    emb = timesteps.type(torch.float32)[:, None] * emb[None, :]
    emb = torch.concat([torch.sin(emb), torch.cos(emb)], axis=1)

    if embedding_dim % 2 == 1:  # zero pad
        emb = torch.pad(emb, [[0, 0], [0, 1]])

    assert emb.shape == (timesteps.shape[0], embedding_dim), f"{emb.shape}"
    return emb
```

```python
t = (torch.rand(100)*10).long()
get_timestep_embedding(t,64)
```

```
tensor([[-0.7568,  0.1689,  0.8038,  ...,  1.0000,  1.0000,  1.0000],
        [ 0.0000,  0.0000,  0.0000,  ...,  1.0000,  1.0000,  1.0000],
        [ 0.8415,  0.6765,  0.5244,  ...,  1.0000,  1.0000,  1.0000],
        ...,
        [ 0.0000,  0.0000,  0.0000,  ...,  1.0000,  1.0000,  1.0000],
        [ 0.0000,  0.0000,  0.0000,  ...,  1.0000,  1.0000,  1.0000],
        [ 0.0000,  0.0000,  0.0000,  ...,  1.0000,  1.0000,  1.0000]])
```

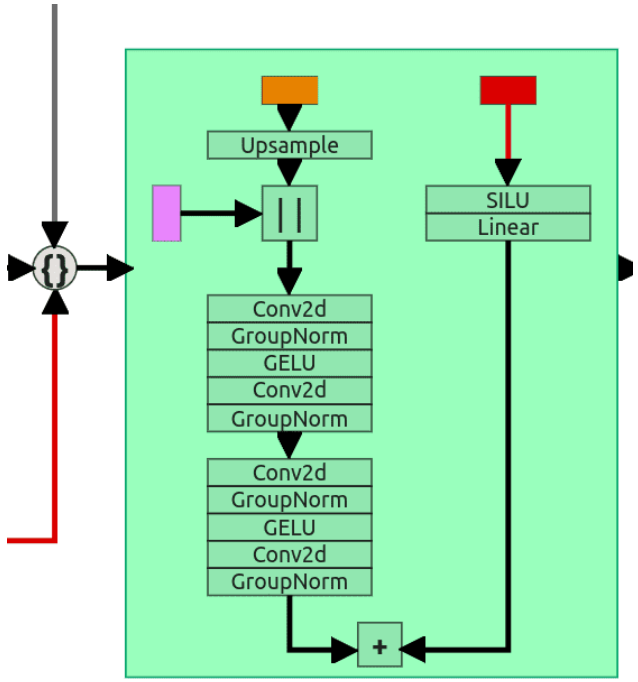
U-Net Downsampling: Implementation

```python
class Downsample(nn.Module):

    def __init__(self, C):
        """
        :param C (int): number of input and output channels
        """
        super(Downsample, self).__init__()
        self.conv = nn.Conv2d(C, C, 3, stride=2, padding=1)

    def forward(self, x):
        B, C, H, W = x.shape
        x = self.conv(x)
        assert x.shape == (B, C, H // 2, W // 2)
        return x
```



U-Net Upsample: Implementation

```python
class Upsample(nn.Module):

    def __init__(self, C):
        """
        :param C (int): number of input and output channels
        """
        super(Upsample, self).__init__()
        self.conv = nn.Conv2d(C, C, 3, stride=1, padding=1)

    def forward(self, x):
        B, C, H, W = x.shape

        x = nn.functional.interpolate(x, size=None, scale_factor=2, mode='nearest')

        x = self.conv(x)
        assert x.shape == (B, C, H * 2, W * 2)
        return x
```



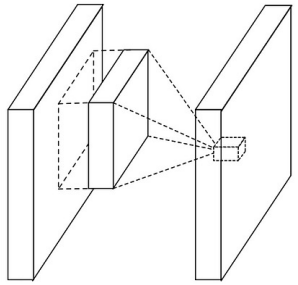

```python
downsample = Downsample(64)
img = torch.randn((10, 64, 400, 400))
img_down = downsample(img)
print("Original image: ", img.shape)

upsample = Upsample(64)
img_up = upsample(img_down)
print("Upsampling image: ", img_up.shape)
```

```
Original image:  torch.Size([10, 64, 400, 400])
Upsampling image:  torch.Size([10, 64, 400, 400])
```

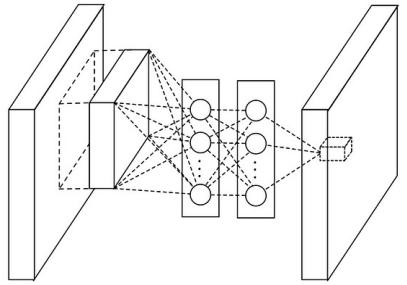
U-Net ResNet: Implementation

```
downsample = Downsample(64)
img = torch.randn((10, 64, 400, 400))
img_down = downsample(img)
print("Original image: ", img.shape)

upsample = Upsample(64)
img_up = upsample(img_down)
print("Upsampling image: ", img_up.shape)

nin = Nin(64,128)
nin(img).shape
```

```
Original image:  torch.Size([10, 64, 400, 400])
Upsampling image:  torch.Size([10, 64, 400, 400])
torch.Size([10, 128, 400, 400])
```



(a) Linear convolution layer

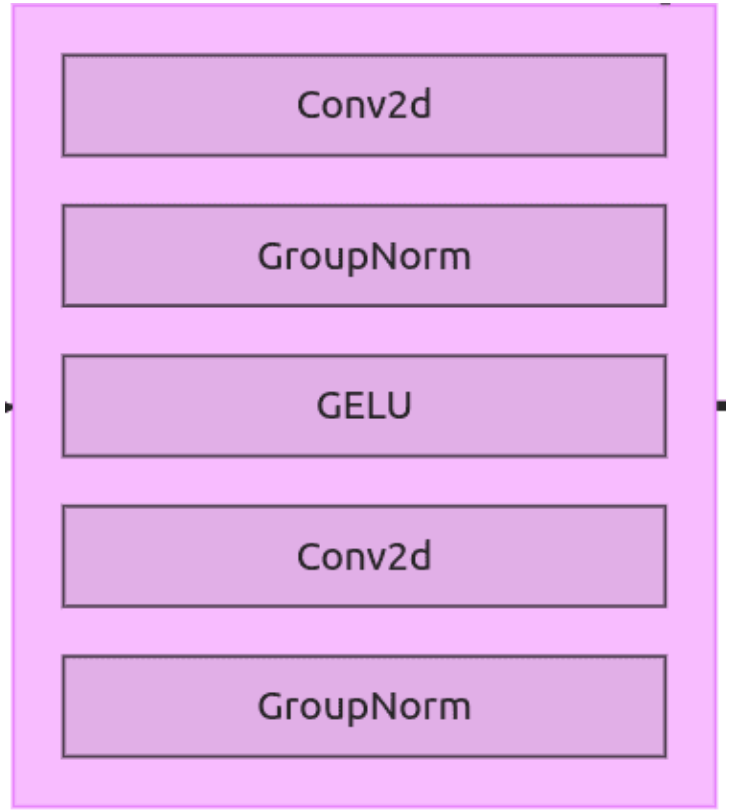


(b) Mlpconv layer

The classic models use linear convolutional layers and the layers are followed by an activation function to scan the input, while the NiN uses multilayer perceptron convolutional layers, at which each layer includes a micro-network.

U-Net ResNet: Implementation

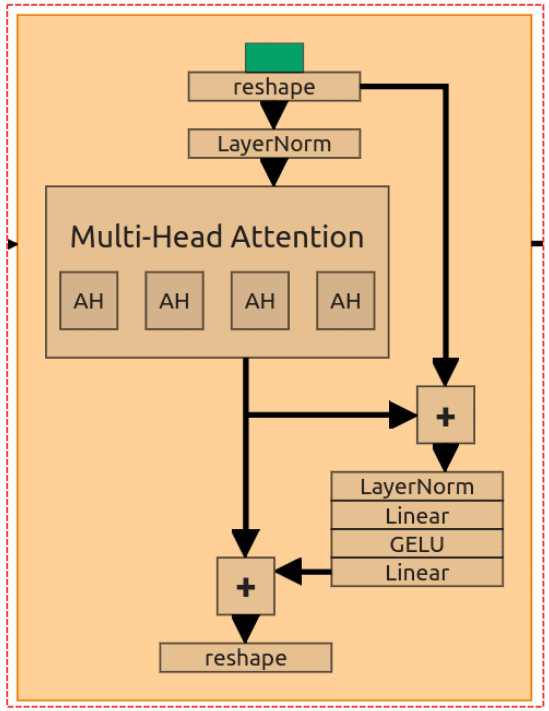
```
t = (torch.rand(10)*10).float()
t = get_timestep_embedding(t,512)

downsample = Downsample(64)
img = torch.randn((10, 64, 128, 128))
img_down = downsample(img)
print("Original image: ", img.shape)
print("Down image: ", img_down.shape)

upsample = Upsample(64)
img_up = upsample(img_down)
print("Upsampling image: ", img_up.shape)

nin = Nin(64,128)
img = nin(img_up)
print("Nin ", img.shape)

resnet = ResNetBlock(128, 128, 0.1)
img = resnet(img, t)
print(img.shape)
```




```
Original image:  torch.Size([10, 64, 128, 128])
Down image:  torch.Size([10, 64, 64, 64])
Upsampling image:  torch.Size([10, 64, 128, 128])
Nin  torch.Size([10, 128, 128, 128])
ResNet  torch.Size([10, 128, 128, 128])
```

U-Net ResNet: Implementation

```
t = (torch.rand(10)*10).float()
t = get_timestep_embedding(t,512)

downsample = Downsample(64)
img = torch.randn((10, 64, 16, 16))
img_down = downsample(img)
print("Original image: ", img.shape)
print("Down image: ", img_down.shape)

upsample = Upsample(64)
img_up = upsample(img_down)
print("Upsampling image: ", img_up.shape)

nin = Nin(64,128)
img = nin(img_up)
print("Nin ", img.shape)

resnet = ResNetBlock(128, 128, 0.1)
img = resnet(img, t)

resnet = ResNetBlock(128, 64, 0.1)
img = resnet(img, t)
print("ResNet ", img.shape)

att = AttentionBlock(64)
img = att(img)
print("Attention ", img.shape)
```




```
Original image:  torch.Size([10, 64, 16, 16])
Down image:  torch.Size([10, 64, 8, 8])
Upsampling image:  torch.Size([10, 64, 16, 16])
Nin  torch.Size([10, 128, 16, 16])
ResNet  torch.Size([10, 64, 16, 16])
Attention  torch.Size([10, 64, 16, 16])
```

U-Net ResNet: Fully Implementation

```
class UNet(nn.Module):

    def __init__(self, ch=128, in_ch=1):
        super(UNet, self).__init__()

        self.ch = ch
        self.linear1 = nn.Linear(ch, 4 * ch)
        self.linear2 = nn.Linear(4 * ch, 4 * ch)

        self.conv1 = nn.Conv2d(in_ch, ch, 3, stride=1, padding=1)

        self.down = nn.ModuleList([ResNetBlock(ch, 1 * ch),
                                   ResNetBlock(1 * ch, 1 * ch),
                                   Downsample(1 * ch),
                                   ResNetBlock(1 * ch, 2 * ch),
                                   AttentionBlock(2 * ch),
                                   ResNetBlock(2 * ch, 2 * ch),
                                   AttentionBlock(2 * ch),
                                   Downsample(2 * ch),
                                   ResNetBlock(2 * ch, 2 * ch),
                                   ResNetBlock(2 * ch, 2 * ch),
                                   Downsample(2 * ch),
                                   ResNetBlock(2 * ch, 2 * ch),
                                   ResNetBlock(2 * ch, 2 * ch)])

        self.middle = nn.ModuleList([ResNetBlock(2 * ch, 2 * ch),
                                      AttentionBlock(2 * ch),
                                      ResNetBlock(2 * ch, 2 * ch)])
```

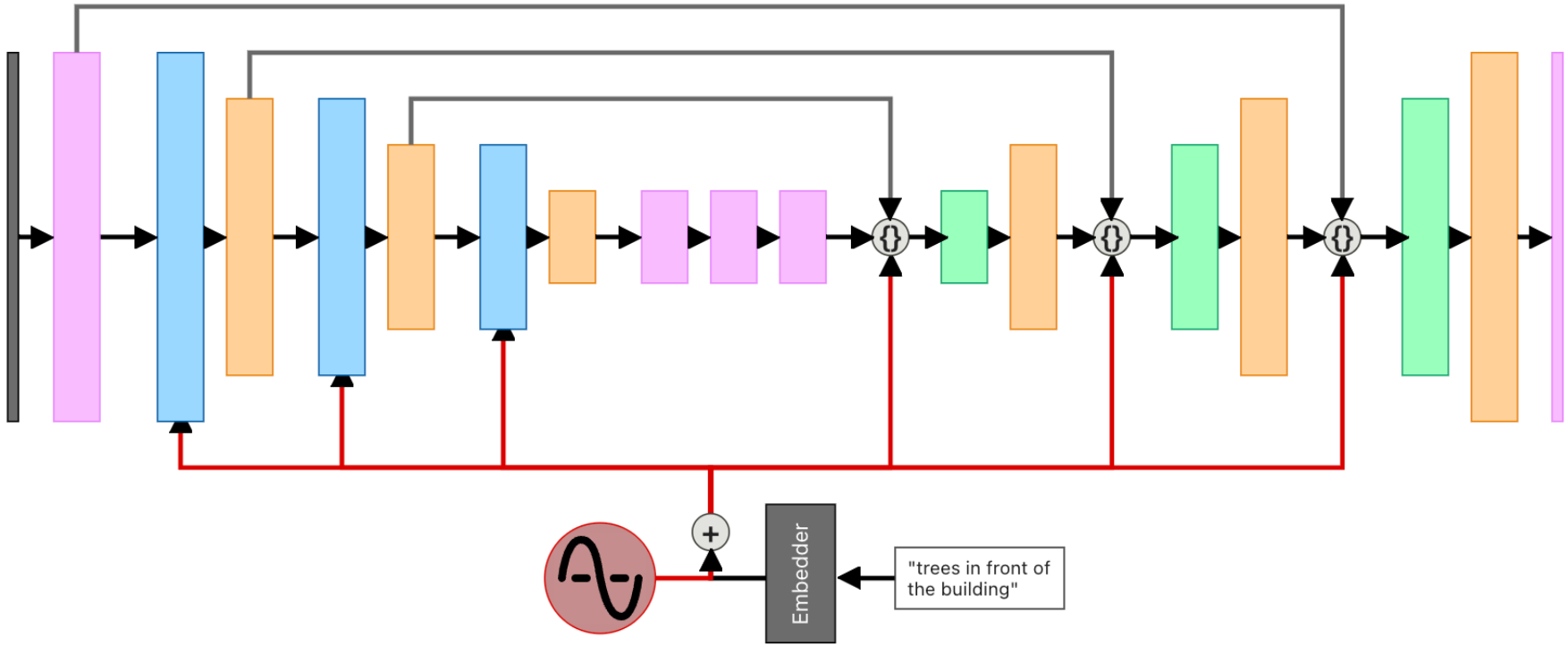
```
t = (torch.rand(10)*10).float()
img = torch.randn((10,1,32,32))
model = UNet()
img = model(img, t)
print(img.shape)
```

```
torch.Size([10, 1, 32, 32])
```

```
print("# parameters: ", sum([p.numel() for p in model.parameters()])/1e6, "M")
```

```
# parameters:  35.713281 M
```

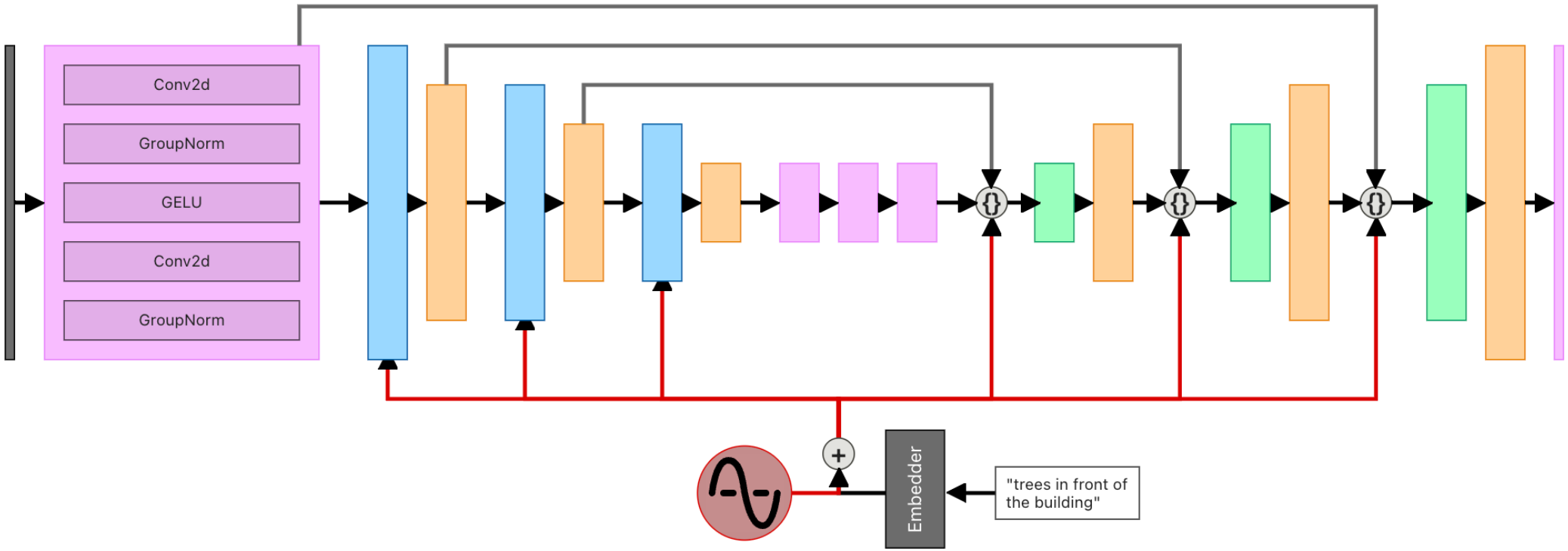
Diffusion Model Architecture



Input
Conv Block
Downsample Block
Self-Attention Block
Upsample Block
Embedding vector

To make the network conditional (dependent on the external input), we're adding a **text embedder**. It will create embedding for given text and **add output vector to the timestep embedding**.

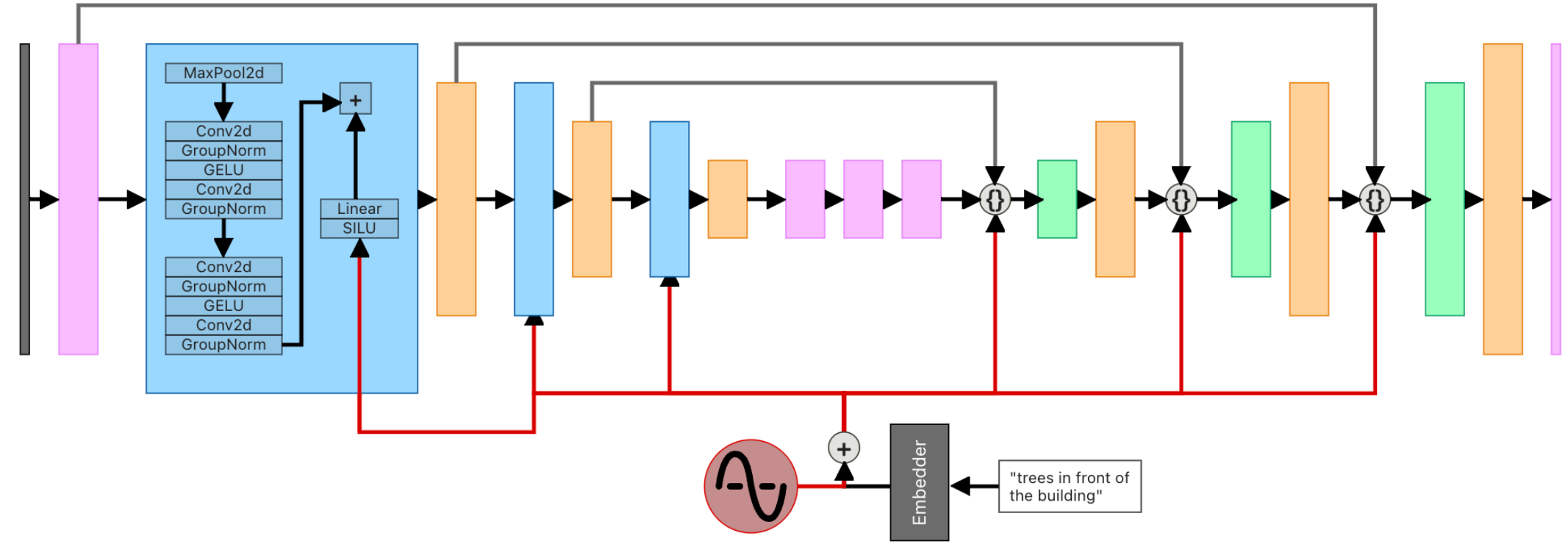
Diffusion Model Architecture



Input
Conv Block
Downsample Block
Self-Attention Block
Upsample Block
Embedding vector

Detailed exploded view of the ResNet/Conv Block. All blocks are structured in the same way, only tensor shapes are different.

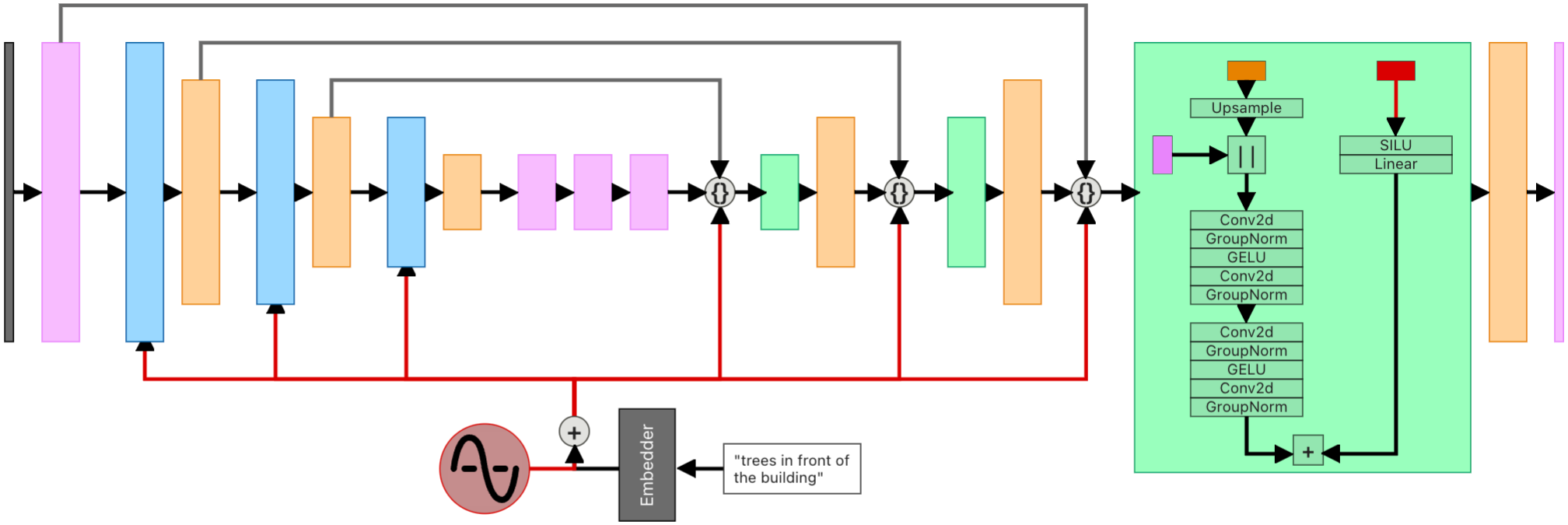
Diffusion Model Architecture



Input
Conv Block
Downsample Block
Self-Attention Block
Upsample Block
Embedding vector

Detailed exploded view of the Downsample Block. All blocks are structured in the same way, only tensor shapes are different.

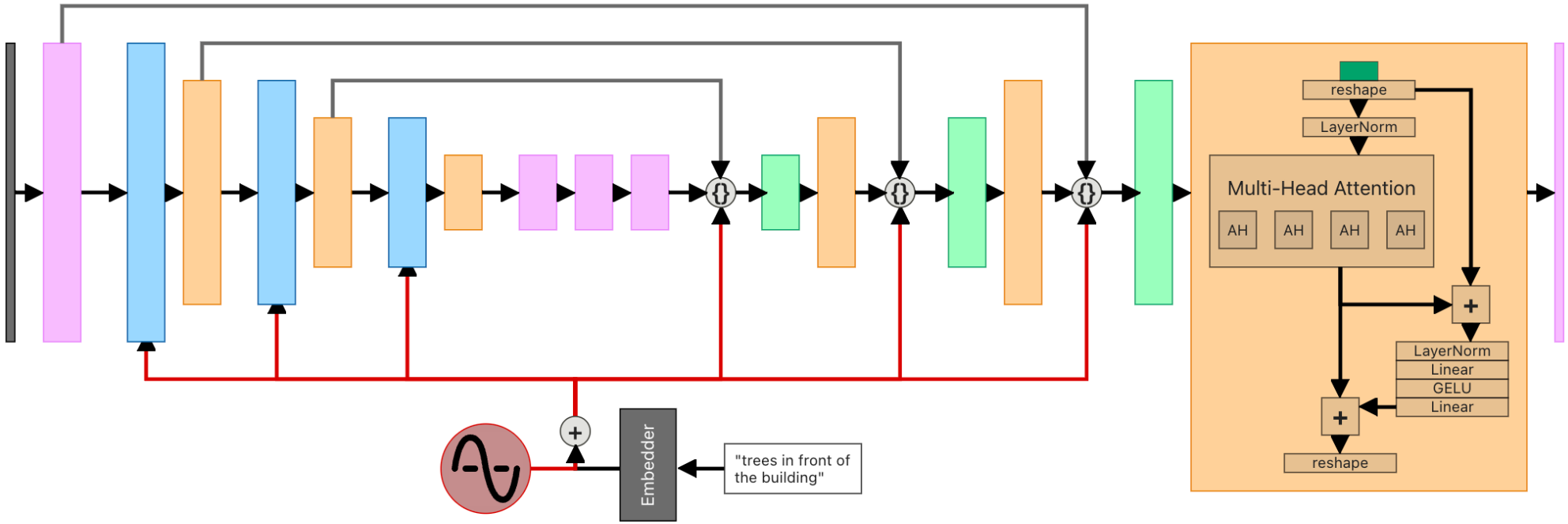
Diffusion Model Architecture



Input
Conv Block
Downsample Block
Self-Attention Block
Upsample Block
Embedding vector

Detailed exploded view of the Upsample Block. All blocks are structured in the same way, only tensor shapes are different.

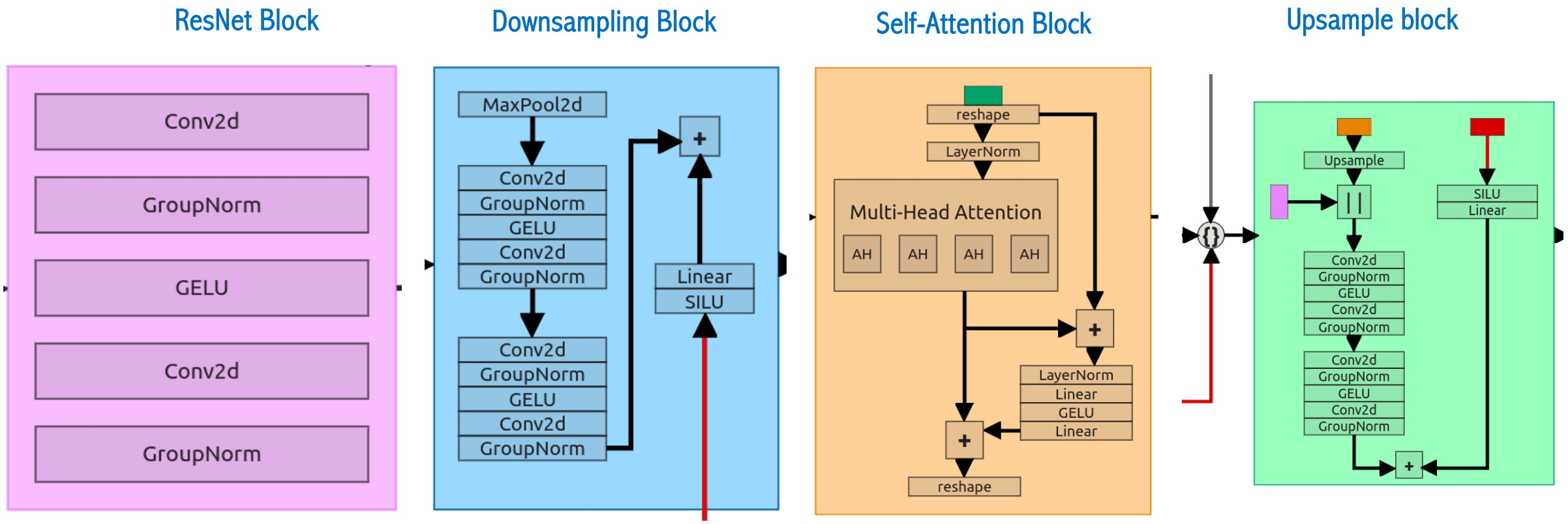
Diffusion Model Architecture



Input
Conv Block
Downsample Block
Self-Attention Block
Upsample Block
Embedding vector

Detailed exploded view of the Attention Block. All blocks are structured in the same way, only tensor shapes are different.

Achitecture: Detailed Discussion




QUIZ TIME

Outline

- Objective
- What is Denoising Probability Diffusion Model
- Forward Diffusion Process: Review
- Reverse Diffusion Process: Explain and Implementation
- Denoise Probability Diffusion Model: Implementation
- Summary

Diffusion Model: Training

Workflow

1 : **repeat**
2 : $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3 : $t \sim \text{Uniform}(\{1, \dots, T\})$
4 : $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5 : Take gradient descent step on

$$\nabla_\theta \left\| \epsilon - \epsilon_\theta \left(\sqrt{\bar{\alpha}_t} \mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon, t \right) \right\|^2$$

6 : **until** converged

Explaination

First, we sample an image from our dataset (2:). Then we sample timestep **t** (3:) and finally sample noise from the normal distribution (4:). As you remember, we've defined how to apply noise at **t** without going through the iterative process:

$$q(x_t|x_0) = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon$$

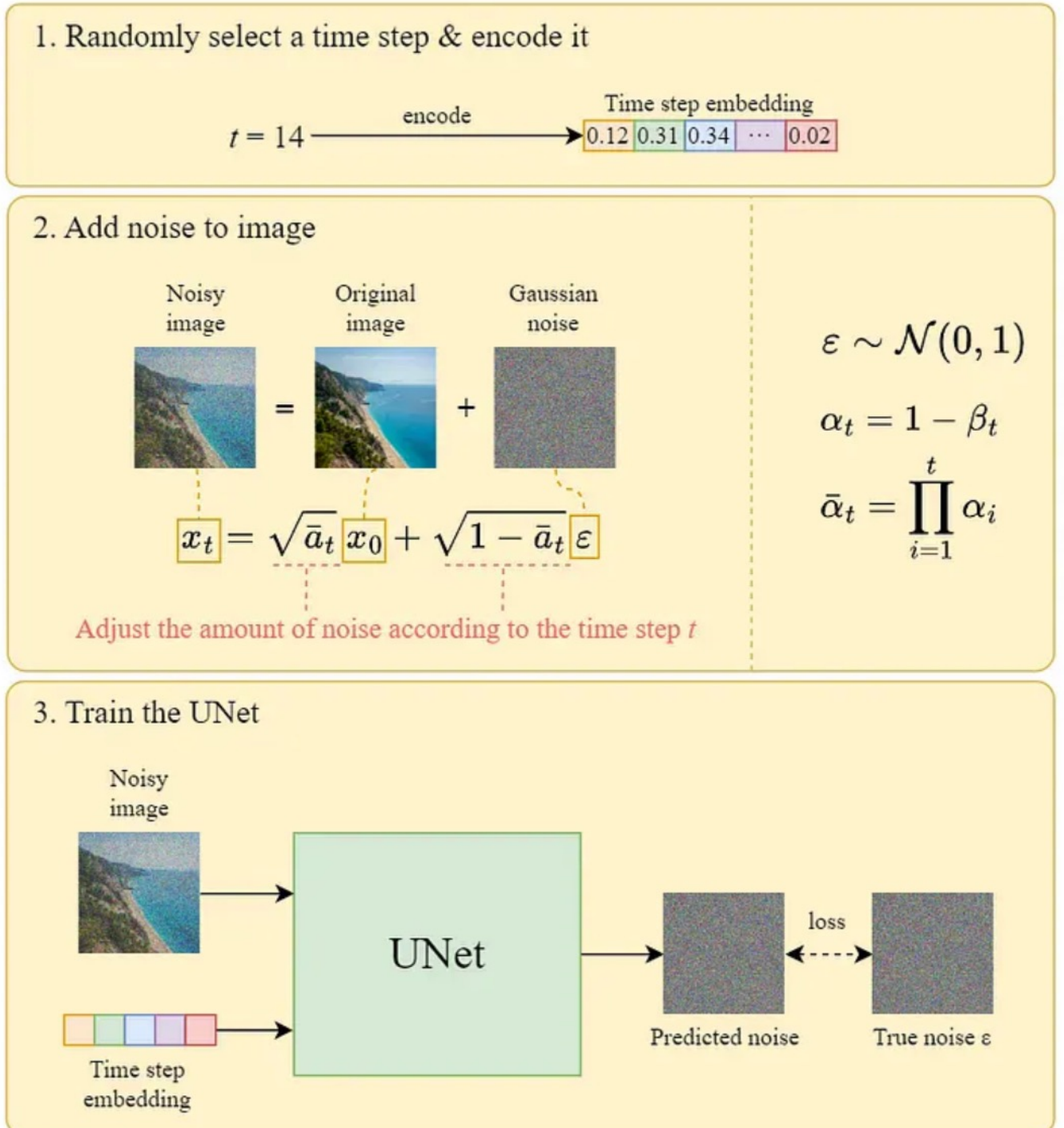
Now we optimize the objective via the gradient descent (5:). That's it, now we repeat the entire process until it converges and the model is ready.

AI VIETNAM
All-in-One Course

UNet

In each epoch:
- A random time step t will be selected for each training sample (image).
- Apply the Gaussian noise (corresponding to t) to each image.
- Convert the time steps to embeddings (vectors).

$$
\begin{aligned}
&\textbf{repeat} \\
&\quad \mathbf{x}_0 \sim q(\mathbf{x}_0) \\
&\quad t \sim \text{Uniform}(\{1, \dots, T\}) \\
&\quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \\
&\quad \text{Take gradient descent step on} \\
&\qquad \nabla_\theta \left\| \epsilon - \epsilon_\theta \left(\sqrt{\bar{\alpha}_t} \mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon, t \right) \right\|^2 \\
&\textbf{until} \text{ converged}
\end{aligned}
$$



Diffusion Model: Training Implementation

Workflow

1 : **repeat**

2 : $\mathbf{x}_0 \sim q(\mathbf{x}_0)$

3 : $t \sim \text{Uniform}(\{1, \dots, T\})$

4 : $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$

5 : Take gradient descent step on

$$\nabla_\theta \left\| \epsilon - \epsilon_\theta \left(\sqrt{\bar{\alpha}_t} \mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon, t \right) \right\|^2$$

6 : **until** converged

```python
class DiffusionModel():

    def __init__(self, T : int, model : nn.Module, device : str):

        self.T = T
        self.function_approximator = model.to(device)
        self.device = device

        self.beta = torch.linspace(1e-4, 0.02, T).to(device)
        self.alpha = 1. - self.beta
        self.alpha_bar = torch.cumprod(self.alpha, dim=0)

    def training(self, batch_size, optimizer):
        """
        Algorithm 1 in Denoising Diffusion Probabilistic Models
        """

        x0 = sample_batch(batch_size, self.device)

        t = torch.randint(1, self.T + 1, (batch_size,), device=self.device, dtype=torch.long)

        eps = torch.randn_like(x0)

        # Take one gradient descent step
        alpha_bar_t = self.alpha_bar[t-1].unsqueeze(-1).unsqueeze(-1).unsqueeze(-1)
        eps_predicted = self.function_approximator(torch.sqrt(
            alpha_bar_t) * x0 + torch.sqrt(1 - alpha_bar_t) * eps, t-1)
        loss = nn.functional.mse_loss(eps, eps_predicted)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        return loss.item()
```

Diffusion Model: Sampling Implementation

Workflow

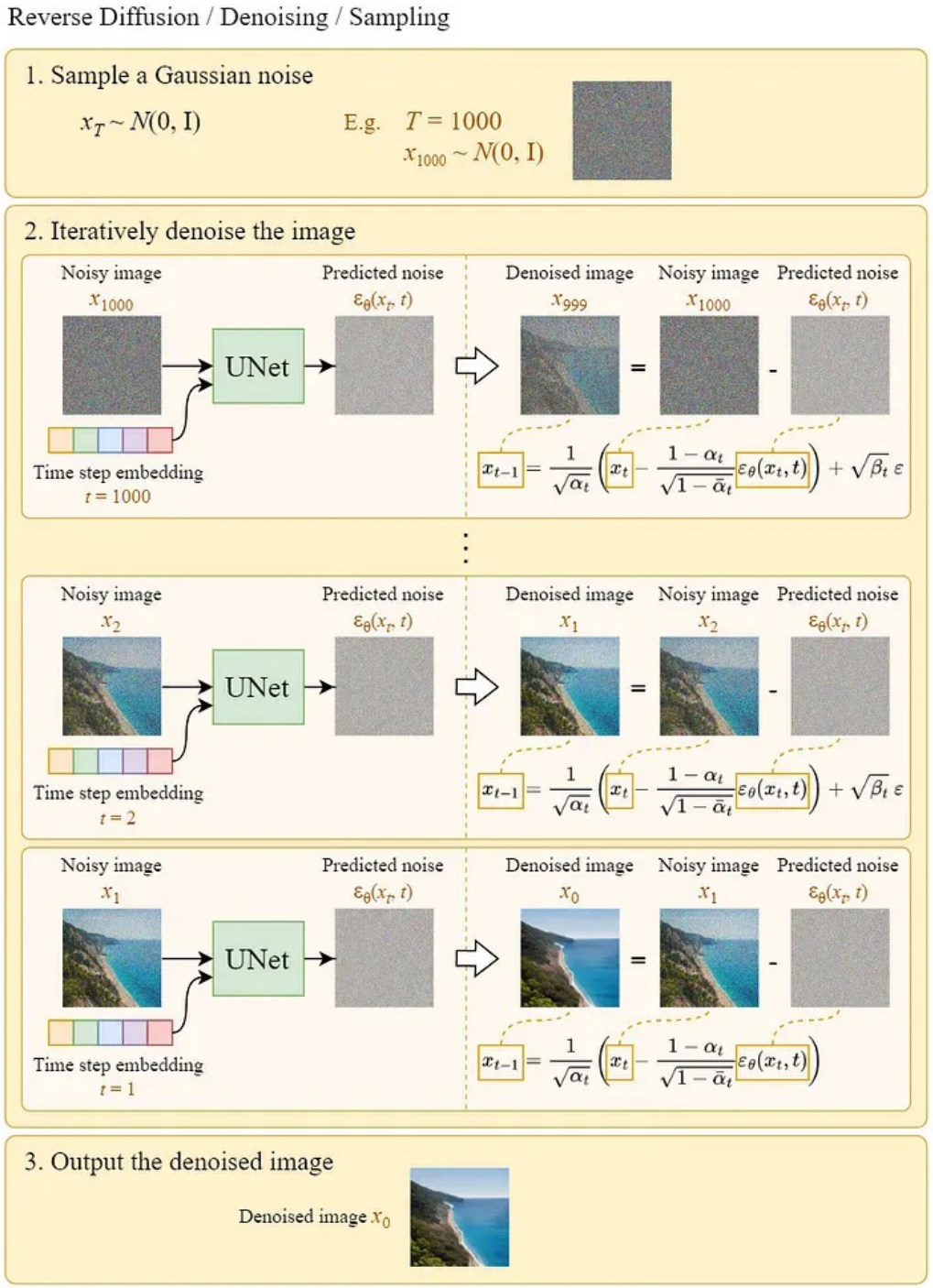
$\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
for $t = T, \ldots, 1$ **do**
  $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
  $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
end for
return $\mathbf{x}_0$

```python
@torch.no_grad()
def sampling(self, n_samples=1, image_channels=1, img_size=(32, 32), use_tqdm=True):

    x = torch.randn((n_samples, image_channels, img_size[0], img_size[1]),
                    device=self.device)

    progress_bar = tqdm if use_tqdm else lambda x : x
    for t in progress_bar(range(self.T, 0, -1)):
        z = torch.randn_like(x) if t > 1 else torch.zeros_like(x)

        t = torch.ones(n_samples, dtype=torch.long, device=self.device) * t

        beta_t = self.beta[t-1].unsqueeze(-1).unsqueeze(-1).unsqueeze(-1)
        alpha_t = self.alpha[t-1].unsqueeze(-1).unsqueeze(-1).unsqueeze(-1)
        alpha_bar_t = self.alpha_bar[t-1].unsqueeze(-1).unsqueeze(-1).unsqueeze(-1)

        mean = 1 / torch.sqrt(alpha_t) * (x - ((1 - alpha_t) / torch.sqrt(
            1 - alpha_bar_t)) * self.function_approximator(x, t-1))
        sigma = torch.sqrt(beta_t)
        x = mean + sigma * z

    return x
```

UNet

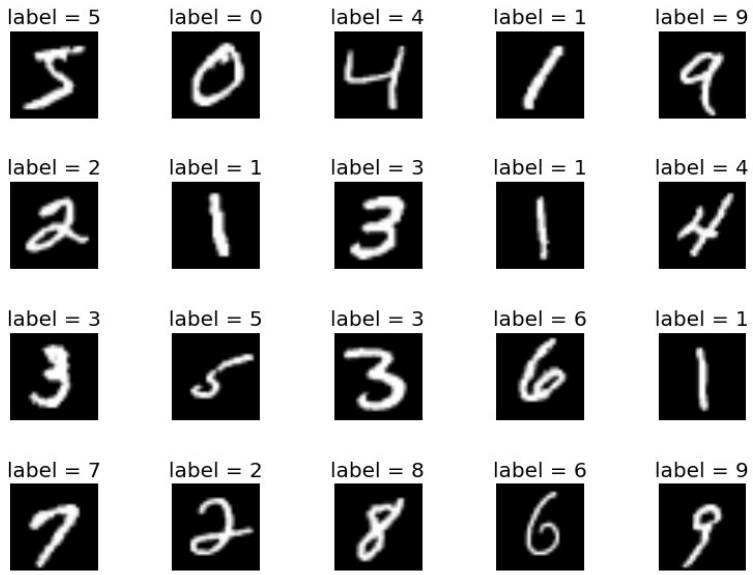
Workflow

1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \ldots, 1$ **do**
3: $\quad \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4: $\quad \mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$



Preparing MNIST Dataset

```python
from keras.datasets.mnist import load_data

(trainX, trainy), (testX, testy) = load_data()
trainX = np.float32(trainX) / 255.
testX = np.float32(testX) / 255.

def sample_batch(batch_size, device):
    indices = torch.randperm(trainX.shape[0])[:batch_size]
    data = torch.from_numpy(trainX[indices]).unsqueeze(1).to(device)
    return torch.nn.functional.interpolate(data, 32)
```

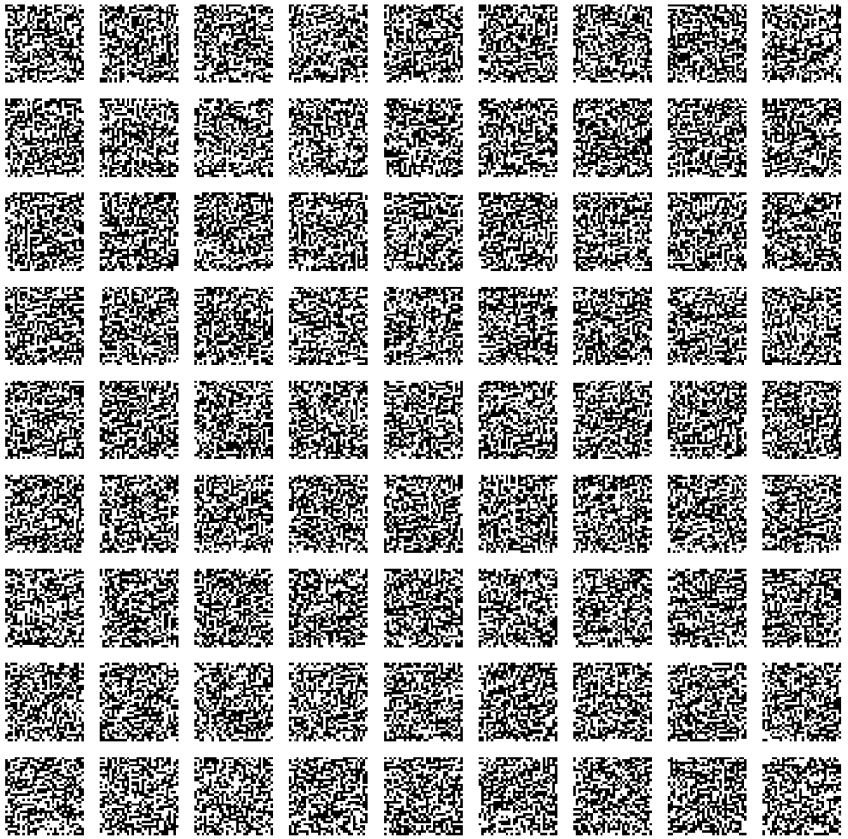



```python
device = 'cpu'
batch_size = 64
model = UNet()
optimizer = torch.optim.Adam(model.parameters(), lr=2e-5)
diffusion_model = DiffusionModel(1000, model, device)
```

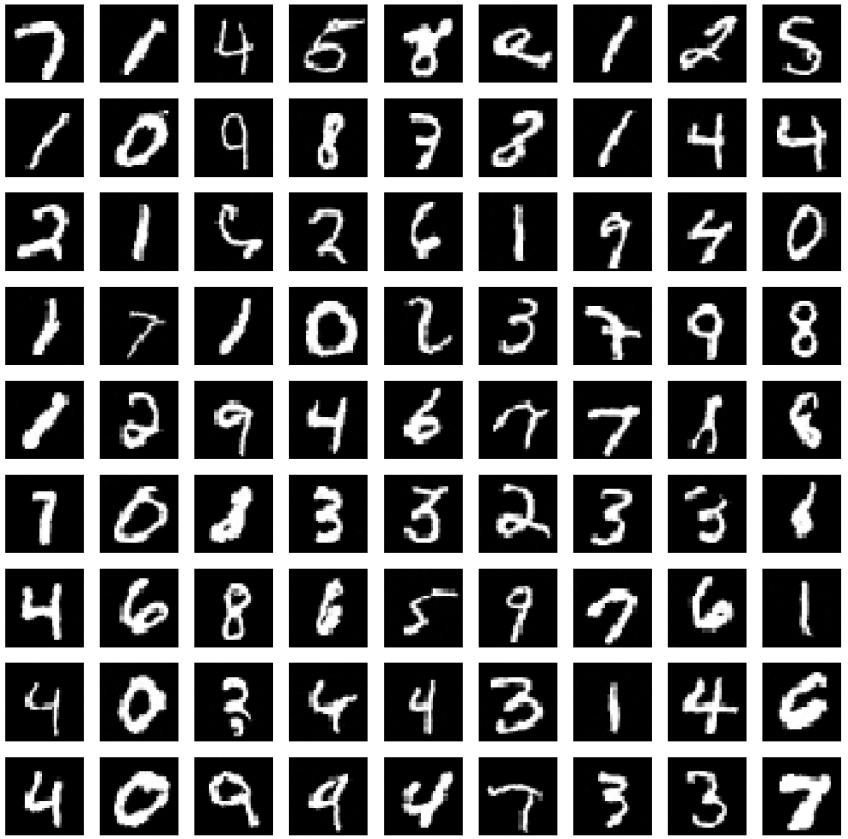
```python
from tqdm import tqdm
import matplotlib.pyplot as plt

training_loss = []
for epoch in tqdm(range(40_000)):
    loss = diffusion_model.training(batch_size, optimizer)
    training_loss.append(loss)

    if epoch % 100 == 0:
        plt.plot(training_loss)
        plt.savefig('training_loss.png')
        plt.close()

        plt.plot(training_loss[-1000:])
        plt.savefig('training_loss_cropped.png')
        plt.close()
```

Diffusion Model: Sampling Implementation

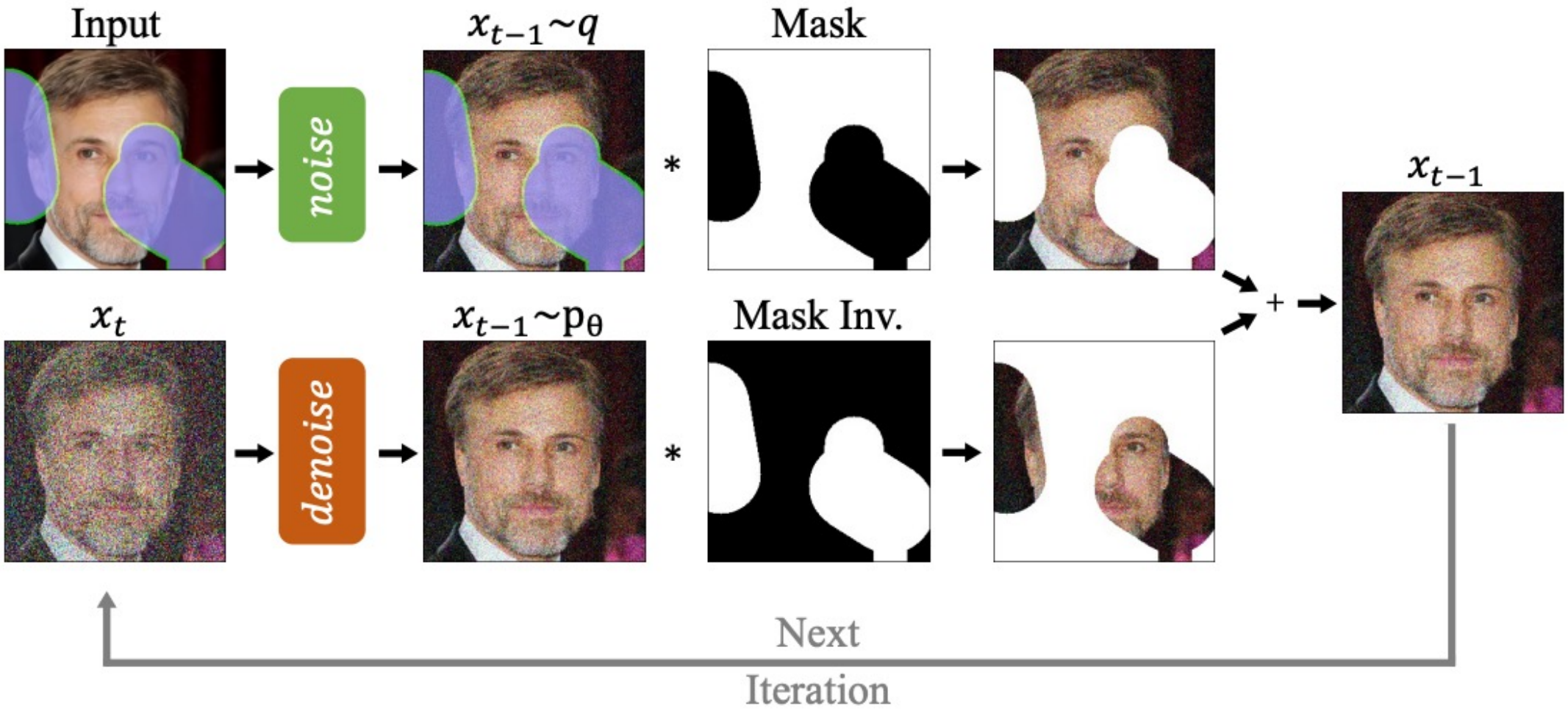
Epoch 1



Epoch 39999



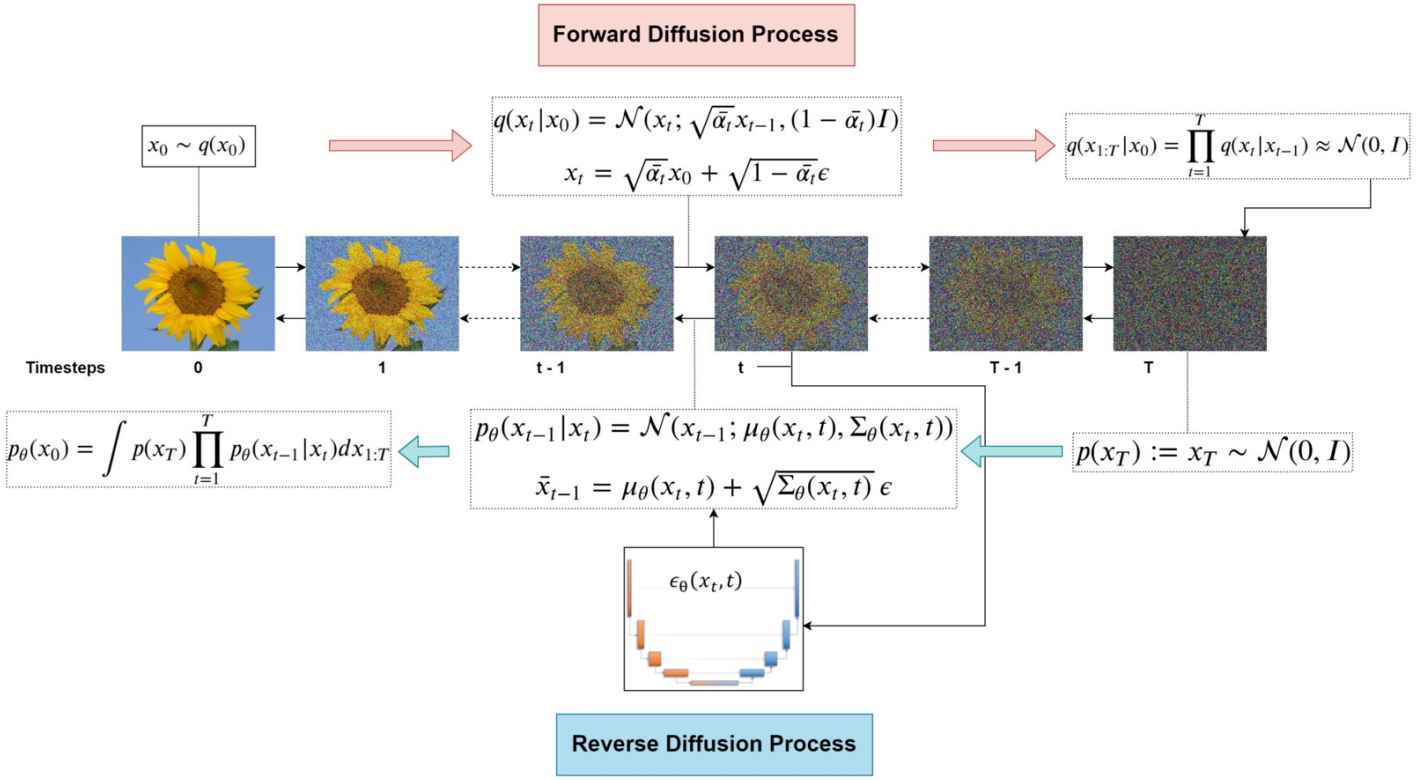
Inpainting with Diffusion Model



Outline

- Objective
- Application of Diffusion Models
- Why Do We Need Diffusion Model?
- Diffusion Model Detail Explanation and Implementation
- Summary

Summary



1. The Diffusion model is divided into two parts: forward diffusion and reverse diffusion.
2. The forward diffusion can be done using the closed-form formula.
3. The backward diffusion can be done using a trained neural network.
4. To approximate the desired denoising step q, we just need to approximate the noise ε_t using a neural network $\varepsilon\theta$.
5. Training on the simplified loss function yields better sample quality.


Thank you