Module 4: Diffusion Models

Diffusion Models: Introduction, Denoising Diffusion Models (DDM): The Flowers Dataset, The Forward Diffusion Process, The Reparameterization Trick, Diffusion Schedules, The Reverse Diffusion Process, The U-Net Denoising Model, Training the Diffusion Model, Training the Diffusion Model, Sampling from the Denoising Diffusion Model, Analysis of the Diffusion Model.

Text Book-1: Chapter 8.

Introduction to Diffusion Models

- **Diffusion models** are **generative AI models** that **learn to create new images by learning how to remove noise.**

- Imagine you have a **blurry picture covered with lots of random dots** (noise). At first, **you cannot recognize what the picture is.**

- A **diffusion model** is an AI model that **learns how to remove this noise little by little** until the **original clear image appears.**

- Instead of **drawing an image from scratch**, the **model starts** with **random noise** and **gradually converts it into a meaningful image**.

DiffuseTV Story

- Imagine you visit an electronics store and **hundreds of TVs placed one after another**.
- The **first TV** shows **only random static** (noise).
- The **second TV** removes a **little of the noise.**
- The **third TV** removes even more noise.
- As you **keep walking**, each **TV shows a slightly clearer picture.**
- Finally, the **last TV** displays a **perfect, clear image**.
- **Each TV performs one small step** of **removing noise**.
- This is **exactly how a diffusion model works**.



Figure 8-1. A long line of connected television sets stretching out along an aisle of a shop (created with Midjourney)

Denoising Diffusion Model (DDM)

What is a Denoising Diffusion Model?

- A **Denoising Diffusion Model (DDM)** is an AI model that learns **how to remove noise from an image step by step**.

- It starts with a **completely noisy image** and **gradually turns** it into a **clear and realistic image**.

Denoising Diffusion Model (DDM)

Example

- Imagine an old **photograph covered with dust**.
- At first, **the photo is difficult to see.**
- You **clean a small amount of dust**.
- You **keep cleaning little by little**.
- Finally, the **photo becomes completely clear.**

A **diffusion model works in the same way -** it removes **noise instead of dust**.

Denoising Diffusion Model (DDM)

How Does DDM Work?

It has **two main processes**.

1. Forward Diffusion (Adding Noise)

- Start with a **clear image**.
- **Add a small amount** of **random noise**.
- **Continue adding more noise**.
- **After many steps**, the **image becomes completely random noise**.

2. Reverse Diffusion (Removing Noise)

- Now the **AI learns to do the opposite**.
- **Start with** a **noisy image.**
- **Remove** a **little noise.**
- **Remove more noise**.
- **Repeat** many times.
- Finally, **obtain a clear image.**

The Flowers Dataset

The model is trained using the **Oxford 102 Flower Dataset**, which contains:

- More than **8,000 colour images**
- **102 different flower categories**

These **images help the model learn the appearance** of **different flowers.**

Before Training

The images are prepared using the following steps:

- **Resize:** Every image is resized to **64 × 64 pixels**.
- **Normalize:** Pixel values are changed from **0 - 255** into the smaller range **0 -1** by dividing every pixel value by **255**.
- **Repeat:** The **dataset is repeated** so the **model sees more examples**.
- **Batch: Images are given to the model** in groups of **64**.

These steps make the training process **faster and easier**.

The Flowers Dataset

Example 8-1: Downloading the Oxford 102 Flower Dataset

bash scripts/download_kaggle_data.sh nunenuh pytorch-challange-flower-dataset

- bash - runs a shell script.
- scripts/download_kaggle_data.sh - the **script that downloads** and **extracts the Kaggle dataset.**
- nunenuh - the Kaggle **username of the person** who **uploaded the dataset**.
- pytorch-challenge **- flower-dataset** - the dataset name.

The Flowers Dataset

Example 8-2: Loading the Oxford 102 Flower Dataset

train_data = utils.image_dataset_from_directory(

"/app/data/pytorch-challange-flower-dataset/dataset", // load the dataset

labels=None, image_size=(64, 64), batch_size=None, // Loads only images without loading class names, resizes them to 64×64 pixels, and loads them individually without creating batches.

shuffle=True, // arranging the training images in a **random order** before giving them to the model, model should not learn based on the order of images.

seed=42, // seed = 42; a random number ensures that the images are shuffled in the **same order every time** the program runs.

interpolation="bilinear",) // keeps the **image smooth when changing** its size to 64 × 64 **by calculating each new pixel** value **from the four nearest pixels.**

The Flowers Dataset

Preprocess and train the image

Defines a function to preprocess each image.

def preprocess(img):

Converts pixel values to float and normalizes them from 0 - 255 to 0 - 1. making the **numbers smaller and helping the model learn faster and more stably.**

img = tf.cast(img, "float32") / 255.0 [TensorFlow's cast() to change the data type of a value].

Returns the normalized image.

return img

lambda x is used to take each image (x) **from the dataset and pass** it to the **preprocess()** function.

train = train_data.map(lambda x: preprocess(x))

The Flowers Dataset

Repeats the complete training dataset 5 times.

train = train.repeat(5)

Groups images into batches of 64 and removes the last incomplete batch.

train = train.batch(64, drop_remainder=True)

- First batch: 64 images
- Second batch: 64 images
- Last batch: only 2 images remove

model.fit(train) // Start training the model using the train dataset.

The Flowers Dataset

Why Do We Add Noise? Why make a clear image noisy?

The reason is :

- By **learning how** to **remove noise**, the **AI understands the structure of real images**.
- Later, it **can start** from **random noise** and **generate completely new images** that **look realistic.**

Example images from the dataset are shown in Figure 8-2.



Figure 8-2. Example images from the Oxford 102 Flower dataset

Forward Diffusion Process

What is Forward Diffusion?

- The **forward diffusion process** is the process of **gradually adding random noise** to a **clear image.**
- The **image becomes noisier** at **every step until** it finally becomes **pure random noise**.

Equation

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

- x_{t-1} = Image from the previous step
- x_t = New noisy image
- β_t = Amount of noise added at step t
- ϵ = Random Gaussian noise

Forward Diffusion Process

The equation says:

New Image = Small part of the previous image + Small amount of random noise

At every step,

- Keep **most of the previous image**.
- Add **a little random noise**.

Repeat many times.

Eventually, the image becomes completely noisy.

Why is $\sqrt{1-\beta_t}$ Used?

a pixel value changes from **200 to 100**,

The original image is multiplied by $\sqrt{1-\beta_t}$ so its intensity is slightly reduced.

This creates space for adding noise while keeping the overall image values stable.

Forward Diffusion Process

The forward diffusion process changes a clear image into random noise, step by step.

Suppose $\mathbf{x_0}$ is the original flower image.

1. A **small amount of random noise** is **added** to $\mathbf{x_0}$ to produce $\mathbf{x_1}$**.**
2. More **noise** is **added** to $\mathbf{x_1}$ to **produce** $\mathbf{x_2}$**.**
3. This **process continues for many steps**.
4. At the final step $\mathbf{x_t}$, the **flower is no longer visible** and the **image looks like complete random noise.**

Clear image $x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow$ Complete noise x_t

Forward Diffusion Process

FORMULA: $\mathbf{x_t = \sqrt{(1 - \beta_t)} \times x_{t-1} + \sqrt{\beta_t} \times \varepsilon}$

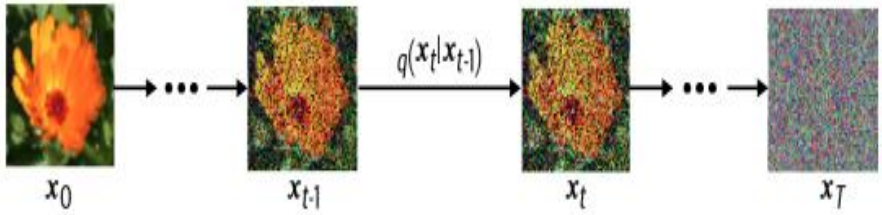


Figure 8-3. The forward diffusion process q

EACH TERM:

x_{t-1} = Image from the previous step.

x_t = New image after adding noise.

t = Current step number.

β_t = Controls the amount of noise added at the current step.

ε = Random Gaussian noise (**it adds random black, white, or grey dots to the image**.)

$\sqrt{(1 - \beta_t)}$ = Slightly reduces the previous image.

$\sqrt{\beta_t}$ = Controls the strength of the new noise(h**ow much noise is added to the image at that step**.

Forward Diffusion Process

EXAMPLE:

Suppose the previous image contributes 90% and random noise contributes 10%.

New image = 90% of the previous image + 10% random noise

After one step, the flower is still visible. After many steps, the noise gradually covers the flower completely.

WHY IS THE PREVIOUS IMAGE REDUCED?

If **noise is repeatedly added without reducing the previous image**, the **pixel values may become too large.**

Therefore, the **previous image is slightly reduced before adding new noise**. This **keeps the image values stable.**

Forward Diffusion Process

WHY IS THIS PROCESS USED?

During training, the **model observes images containing different amounts of noise** and **learns how to remove that noise.**

After training, the process is reversed:

Random noise → Less noise → Clear flower image

Forward process = Adds noise to an image.

Reverse process = Removes noise and generates a new image.

Reparameterization Trick

Normally, noise is added step by step: $\mathbf{x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_t}$

The **reparameterization trick** allows us to **directly create $\mathbf{x_t}$** from the **original image $\mathbf{x_0}$** without calculating the intermediate images.

Formula: $\mathbf{x_t = \sqrt{\bar{\alpha}_t} \times x_0 + \sqrt{(1 - \bar{\alpha}_t)} \times \varepsilon}$

x_0 = Original image

x_t = Noisy image at step t

$\bar{\alpha}_t$ = Amount of the original image remaining

$1 - \bar{\alpha}_t$ = Amount of noise added

ε = Random noise

The **equation can calculate the noisy** image **directly for any chosen time step t, without calculating all the previous steps one by one**. This makes the training process faster.

Reparameterization Trick

Normally, noise is added step by step: $x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_t$

- Take the **original image** x_0.
- Choose **how much noise you want using** time step t.
- Create **random noise** ε of the **same image size.**
- **Keep** some **part of the original image** because noise is added **gradually**, not all at once. clear image → slightly noisy image → highly noisy image
- **Add some random noise** to it.
- The result x_t is the noisy image.

It can **directly create the noisy image** at **any selected step t** from the **original image** x_0, without calculating all earlier noisy images.

Diffusion Schedule

What is a Diffusion Schedule?

A **diffusion schedule** decides **How much noise should be added at each step?**

Different schedules add noise differently.

1. Linear Diffusion Schedule

Noise is added **slowly and evenly** at **each step**.

In the beginning, **the image changes very little**.

As the steps increase, **more noise is added**.

At the **final step**, the **image becomes almost complete random noise**.

Example:

Step 1 → Very little noise

Step 200 → More noise

Step 500 → Even more noise

Step 1000 → Maximum noise (image is almost pure noise)

Diffusion Schedule

The noise value β_t **gradually increases** at the **same rate from 0.0001 to 0.1.**

- At each step, the noise increases equally.
- At the final step, the noise value reaches 0.01.
- Advantage: It is simple, easy to understand and easy to implement.

The code uses **1,000 steps** to **slowly add noise to an image**.

- **T = 1000:** Total number of steps.
- **min_rate = 0.0001:** A **very small amount of noise** is added at the first step.
- **max_rate = 0.1:** More **noise is added at the final step**.
- **betas: Amount of noise added** at each step.
- **alphas = 1 − betas: how much** of the **original image remains after adding noise in one step**.
- **alpha_bars:** Total **image information remaining** after **all previous steps**.
- **signal_rates:** Amount of the original image present.
- **noise_rates:** Amount of noise present.

Diffusion Schedule

Example 8-3. The linear diffusion schedule

Minimum noise = 0.0001

Maximum noise = 0.02

1. The **code sets** the **starting noise** to **0.0001**.
2. It **sets the ending noise** to **0.01**.
3. This formula calculates the noise at each step:

beta = minimum noise + diffusiontime × (maximum noise − minimum noise)

Time is a value between **0 and 1** that **shows the progress of adding noise**:

- **0** → beginning of the process
- **0.5** → halfway through the process
- **1** → end of the process

For example, diffusion_time = 0.5 means the model is at approximately the **500th step out of 1,000 steps**.

Diffusion Schedule

Example 8-3. The linear diffusion schedule

- As **time increases**, the **noise slowly increases** from **0.0001 to 0.01**.
- This formula **calculates how much image information is kept**: ***alpha = 1 − beta***
- **If the noise is 0.1, the image information kept** is: **1 − 0.01 = 0.90 (90% image)**

  Alpha bar multiplies all previous alpha values to **find how much of the original image is still remaining.**
  Step 1 alpha = **0.98**
  Step 2 alpha = **0.97**
  Alpha bar = 0.98 × 0.97 = 0.9506 -> which is signal rate

- The **signal rate tells us how much of the original image remains**.
- The **noise rate is calculated** as: **noise rate = 1 − signal rate => 1- 0.9506= 0.494 is the noise**
- The **code repeats this process** for **1,000 steps**. In every step, **the image information decreases and the noise increases.**

Diffusion Schedule

Example 8-3. The linear diffusion schedule

This creates the diffusion times for all steps: **diffusion_times = [x/T for x in range(T)]**

diffusion time = current step / total steps
Since T = 1000:
Step 0 → 0 / 1000 = 0
Step 1 → 1 / 1000 = 0.001
Step 500 → 500 / 1000 = 0.5
Step 999 → 999 / 1000 = 0.999

linear_noise_rates, linear_signal_rates = linear_diffusion_schedule(diffusion_times)

calculates and stores the noise amount and the remaining image amount for all 1,000 steps.

Diffusion Schedule

Example 8-3. The linear diffusion schedule

```
def linear_diffusion_schedule(diffusion_times):
min_rate = 0.0001
max_rate = 0.1
betas = min_rate + tf.convert_to_tensor(diffusion_times) * (max_rate - min_rate)
alphas = 1 - betas
alpha_bars = tf.math.cumprod(alphas)
signal_rates = alpha_bars
noise_rates = 1 - alpha_bars
return noise_rates, signal_rates
T = 1000
diffusion_times = [x/T for x in range(T)]
linear_noise_rates, linear_signal_rates = linear_diffusion_schedule(
diffusion_times
)
```

Diffusion Schedule

2. Cosine Diffusion Schedule

The **Cosine Diffusion Schedule** is used to **decide how much of the original image to keep and how much noise to add at each diffusion step**.

- **Beginning:** Very little noise is added.
- **Middle:** Noise increases gradually.
- **End:** Noise increases faster.

Diffusion Schedule

2. Cosine Diffusion Schedule: To calculate the original image information remaining:

$$\bar{\alpha}_t = \cos^2\left(\frac{t}{T} \cdot \frac{\pi}{2}\right)$$

- $\bar{\alpha}_t$ = amount of original image information remaining at step t
- **t** = current step
- **T** = total number of steps
- **t/T** = progress of the diffusion process
- **π/2** = 90 degrees
- **cos²** means calculate the cosine value and multiply it by itself

If t = 500 and T = 1000:

t/T = 500/1000 = 0.5

Angle = 0.5 × 90° = 45°

cos(45°) = 0.707

$\bar{\alpha}_t = 0.707^2 = 0.5$

Therefore, at step 500:

- **50% of the original image information remains**
- **50% noise is present**

It only **calculates how much original image information should remain** at **step t**.

It **does not create a noisy image**.

Diffusion Schedule

2. Cosine Diffusion Schedule - To create the noisy image:

$$x_t = \cos\left(\frac{t}{T}\frac{\pi}{2}\right)x_0 + \sin\left(\frac{t}{T}\frac{\pi}{2}\right)\epsilon$$

This formula **creates the noisy image at step t**.

Noisy image = cos value multiplied by the original image + sin value multiplied by random noise.

- t = current step
- T = total number of steps
- $\frac{\pi}{2}$ = 90 degrees
- **cos** decides **how much original image is kept**
- **sin** decides **how much noise is added**

It uses the calculated amounts to create the noisy image.

The first formula calculates the proportion, and the second formula uses it to create the noisy image.

Diffusion Schedule

Linear diffusion schedule:

- **Linear schedule** adds a fixed amount to the noise rate at each step.

- Example:

0.001 → 0.002 → 0.003 → 0.004

Cosine schedule:

- **Cosine schedule** calculates noise using a curved sine/cosine function.
- It usually:
  - increases slowly at the beginning,
  - changes more quickly in the middle,
  - increases slowly again near the end

Example of a **cosine schedule** using 5 stages:
Step 0: 0% noise
Step 1: 15% noise
Step 2: 50% noise
Step 3: 85% noise
Step 4: 100% noise

Diffusion Schedule

Offset Cosine Schedule

The cosine schedule is slightly modified.

Why?

Because the first few noise steps in the pure cosine schedule are **too small**.

The offset version adds a small adjustment so that the model learns more effectively from the beginning.

Diffusion Schedule

Example 8-4. The cosine and offset cosine diffusion schedules

```
def cosine_diffusion_schedule(diffusion_times):
signal_rates = tf.cos(diffusion_times * math.pi / 2)
noise_rates = tf.sin(diffusion_times * math.pi / 2)
return noise_rates, signal_rates
def offset_cosine_diffusion_schedule(diffusion_times):
min_signal_rate = 0.02
max_signal_rate = 0.95
start_angle = tf.acos(max_signal_rate)
end_angle = tf.acos(min_signal_rate)
diffusion_angles = start_angle + diffusion_times * (end_angle - start_angle)
signal_rates = tf.cos(diffusion_angles)
noise_rates = tf.sin(diffusion_angles)
return noise_rates, signal_rates
```

Diffusion Schedule

COSINE DIFFUSION SCHEDULE

This function receives the current diffusion time between 0 and 1. It uses the cosine function to calculate how much of the original image should remain and the sine function to calculate how much noise should be added. Finally, it returns the noise rate and signal rate.

OFFSET COSINE DIFFUSION SCHEDULE

This function starts with 95% of the original image and keeps at least 2% of the original image at the final step. It converts the maximum and minimum signal rates into starting and ending angles. It then calculates the angle for each diffusion step. Using this angle, it calculates how much of the original image remains and how much noise is present. Finally, it returns the noise rate and signal rate.

Diffusion Schedule

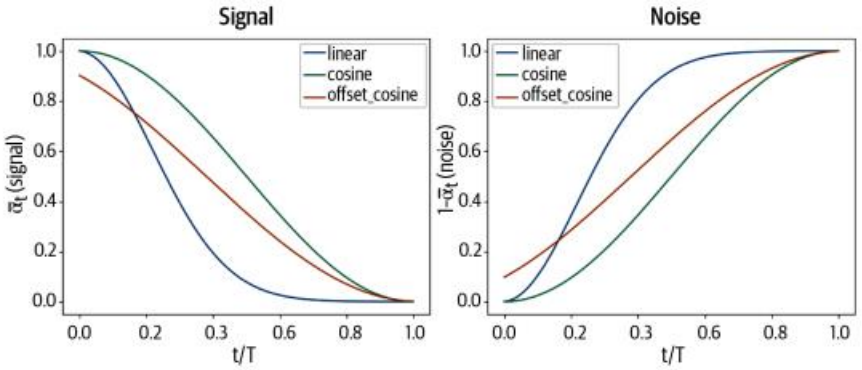


Figure 8-4. The signal and noise at each step of the noising process, for the linear, cosine, and offset cosine diffusion schedules



Figure 8-5. An image being corrupted by the linear (top) and cosine (bottom) diffusion schedules, at equally spaced values of t from 0 to T (source: Ho et al., 2020)

We can compute the αtvalues for each $t$to show how much signal (αt) and noise (1 - αt) is let through at each stage of the process for the linear, cosine, and offset Cosine diffusion schedules, as shown in Figure 8-4.

Notice how the noise level ramps up more slowly in the cosine diffusion schedule. A cosine diffusion schedule adds noise to the image more gradually than a linear diffusion schedule, which improves training efficiency and generation quality. This can also be seen in images that have been corrupted by the linear and cosine schedules (Figure 8-5).

Reverse Diffusion Process

What is Reverse Diffusion?

The **reverse diffusion process** is the process of **removing noise step by step** to recover a clear image.

It is the **opposite of the forward diffusion process**.

What Does the Neural Network Do?

The neural network learns to answer one question:

"What noise was added to this image?"

Once it predicts the noise, it removes it to make the image clearer.

This process is repeated many times until a realistic image is generated.

Reverse Diffusion Process

Training Process

Start with a **clear image** (x_0).

Add noise to get a noisy image (x_t).

Give the noisy image to the neural network.

The network predicts the added noise (ϵ).

Compare the predicted noise with the actual noise.

Update the network to improve its prediction.

Reverse Diffusion Process

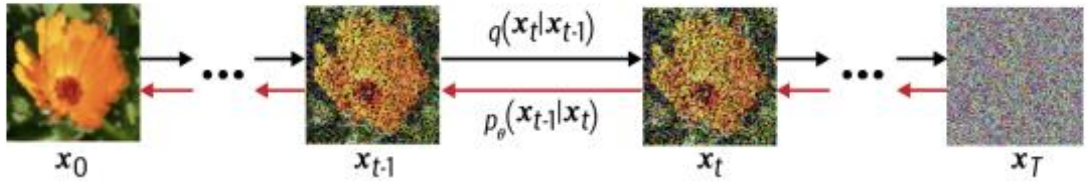


Figure 8-6. The reverse diffusion process $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ tries to undo the noise produced by the forward diffusion process

Equation

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

x_0= Original image
x_t= Noisy image
ϵ= Added noise
The network learns to **predict** ϵ.

Reverse Diffusion Process

Two Networks Used

1. Training Network

Learns by updating its weights.

Used during training.

2. EMA Network (Exponential Moving Average)

Keeps a smooth average of the trained weights.

Used to generate final images.

Produces **more stable and better-quality images**.

Algorithm 1 Training

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
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

Reverse Diffusion Process

The diffusion model training works as follows:

1. Select an original image from the dataset.
2. Randomly select a time step or noise level.
3. Generate random noise.
4. Add the random noise to the original image to create a noisy image.
5. Give the noisy image and time step to the model.
6. The model tries to predict the noise added to the image.
7. Compare the actual noise with the noise predicted by the model.
8. Update the model to reduce the difference between them.
9. Repeat these steps until the model learns to predict and remove noise correctly.

Algorithm 1 Training

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
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

The U-Net Denoising Model

What is a U-Net?

A **U-Net** is a deep learning model used to **remove noise from an image**.

In diffusion models, it learns how to **convert a noisy image into a clearer image**.

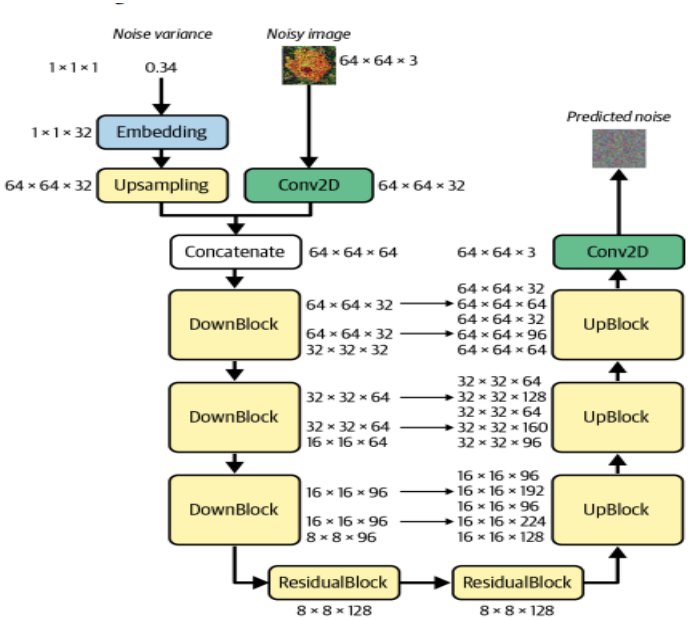


Figure 8-8. U-Net architecture diagram

The U-Net Denoising Model

A **U-Net** is a type of neural network commonly used in **diffusion models** to remove noise from images.

It has **two main parts**:

1. Downsampling Path (Encoder)

The input image enters the network.

The image becomes **smaller step by step**, while the number of channels increases.

Image size decreases → the image becomes compressed.

Channels increase → the network learns more features.

This part learns the **important information** present in the image.

2. Upsampling Path (Decoder)

The compressed image is expanded step by step until it reaches its original size.

The number of channels decreases, and the network reconstructs a **clear image by removing noise**.

The U-Net Denoising Model

1. Sinusoidal Embedding

In a **diffusion model**, the amount of noise added to an image changes at every step.

Example:

Step 1 → Very little noise

Step 500 → Medium noise

Step 1000 → Maximum noise

The model needs to know the **current noise level** so that it can remove the correct amount of noise.

However, the noise level is represented by only **one number**.

Example: Noise level = 0.5

A single number may not provide enough information to the neural network. Therefore, **sinusoidal embedding** converts this single number into a **larger vector containing multiple values**.

This vector helps the model understand the current noise level more clearly and remove the noise correctly.

The U-Net Denoising Model

Why Do We Need Sinusoidal Embedding?

Instead of giving the model only one noise value:

Noise level = 0.5

- We convert it into a vector containing many values: [0.21, 0.85, -0.34, 0.76, ...]
- This gives the network more information about the noise level and helps it remove noise correctly.
- Sinusoidal embedding uses sine and cosine functions with different frequencies.

Equation: $\gamma(x) = [\sin(2\pi f_0 x), \sin(2\pi f_1 x), ..., \sin(2\pi f_{l-1} x), \cos(2\pi f_0 x), \cos(2\pi f_1 x), ..., \cos(2\pi f_{l-1} x)]$

Where:

x = Current noise level

f = Different frequencies used in the sine and cosine functions

L = Number of frequencies

The U-Net Denoising Model

Residual Block

- A residual block is a group of layers inside the U-Net.
- It has a shortcut connection called a **skip connection**.
- The input is directly added to the output of the layers.
- This helps information pass through the network easily.
- It helps train deeper neural networks.
- It reduces the problem of vanishing gradients.
- It also prevents performance from decreasing when the network becomes deeper.
- Residual blocks help the U-Net learn complex image features better.

The U-Net Denoising Model

Why Use Residual Blocks?

Deep neural networks can face two problems:

1. Vanishing Gradient Problem

During training:

Input Layer
↓
Layer 1
↓
Layer 2
↓
Layer 50, The learning signal becomes very small as it travels backward.

Result:

Early layers learn very slowly.
Training becomes difficult.
Residual connections help the gradient flow directly.

The U-Net Denoising Model

Why Use Residual Blocks?

2. Degradation Problem

Adding more layers does not always improve accuracy.

Example:

10 layers → 95% accuracy

50 layers → 92% accuracy

A deeper network may perform worse.

Residual blocks reduce this problem by allowing the network to easily learn an identity function.

The U-Net Denoising Model

DownBlocks and UpBlocks

Each successive DownBlock increases the number of channels via block_depth (=2 in our example) ResidualBlocks, while also applying a final AveragePooling2D layer in order to halve the size of the image. Each ResidualBlock is added to a list for use later by the UpBlock layers as skip connections across the U-Net.

An UpBlock first applies an UpSampling2D layer that doubles the size of the image, through bilinear interpolation. Each successive UpBlock decreases the number of channels via block_depth (=2) ResidualBlocks, while also concatenating the outputs from the DownBlocks through skip connections across the U-Net. A diagram of this process is shown in Figure 8-11.

The U-Net Denoising Model

DownBlocks and UpBlocks

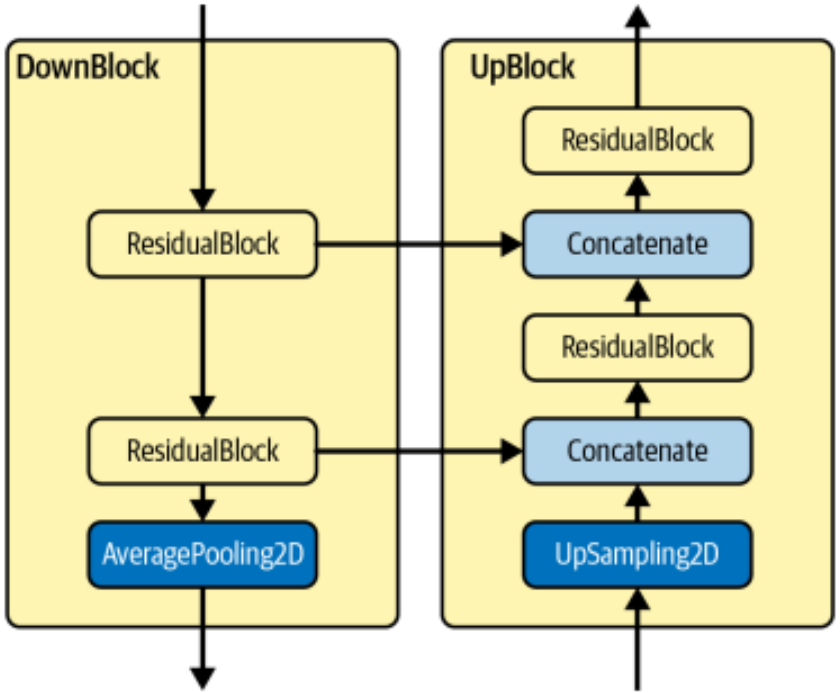


Figure 8-11. The `DownBlock` and corresponding `UpBlock` in the U-Net

Training the Diffusion Model

1. Training the Model

The goal of training a diffusion model is to teach the network **how to remove noise from an image**.

During training:

We take a normal image (for example, a flower image).

We add different levels of noise to the image.

The model learns to **predict the noise added to the image**.

After learning, the model can remove noise and generate new images.

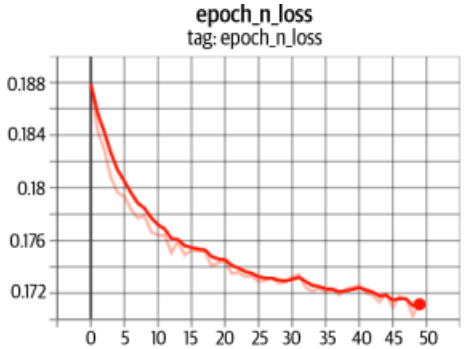


Figure 8-12. The noise mean absolute error loss curve, by epoch

Example 8-10. Code for training the DiffusionModel

```
model = DiffusionModel()
model.compile(
optimizer=optimizers.experimental.AdamW(learning_rate=1e-3, weight_decay=1e-4),
loss=losses.mean_absolute_error,
)
model.normalizer.adapt(train)
model.fit(
train,
epochs=50,

)
```

Sampling from the Diffusion Model (Generating Images)

After training, we can create new images.
The process is opposite to training.

Training (Forward Process)

Image → Add Noise → Random Noise

$$\mathbf{x}_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\underbrace{\left(\frac{\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\epsilon_\theta^{(t)}(\mathbf{x}_t)}{\sqrt{\bar{\alpha}_t}}\right)}_{\text{predicted } \mathbf{x}_0} + \underbrace{\sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\cdot\epsilon_\theta^{(t)}(\mathbf{x}_t)}_{\text{direction pointing to } \mathbf{x}_t} + \underbrace{\sigma_t\epsilon_t}_{\text{random noise}}$$

Generation (Reverse Process)

Random Noise → Remove Noise Step-by-Step → New Image

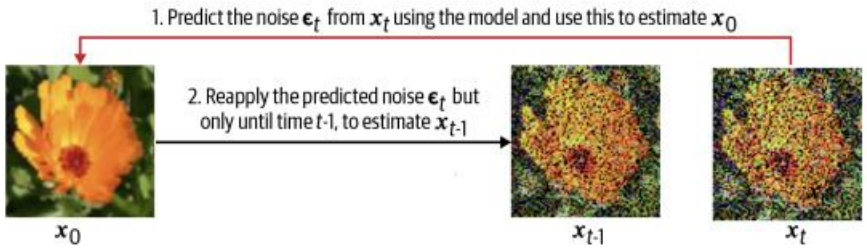


Figure 8-13. One step of the sampling process for our diffusion model

Analysis of the Diffusion Model

After training the diffusion model, its performance and capabilities can be analyzed in three main ways:

Generating New Images

Effect of Diffusion Steps on Image Quality

Interpolating Between Images

1. Generating New Images Using the Diffusion Model

A trained diffusion model can generate new images by applying the **reverse diffusion process**.

Process:

The model starts with a random noise image.

The reverse diffusion process gradually removes noise.

The model converts random noise into a meaningful image.

Finally, the generated image is denormalized to bring pixel values back to the original range **[0,1]**.

Analysis of the Diffusion Model

Generating New Images

Steps:

Step 1:

Create random noise.

Example:

Random pixels

↓

Step 2:

Apply reverse diffusion.

The model removes noise step-by-ste

↓

Step 3:

Convert values back to normal image range.

↓

Output:

A new generated image.

Example 8-12. Generating images using the diffusion model

```
class DiffusionModel(models.Model):

...

    def denormalize(self, images):
        images = self.normalizer.mean + images * self.normalizer.variance**0.5 ❶
        return tf.clip_by_value(images, 0.0, 1.0)

    def generate(self, num_images, diffusion_steps):
        initial_noise = tf.random.normal(shape=(num_images, 64, 64, 3)) ❶
        generated_images = self.reverse_diffusion(initial_noise, diffusion_steps) ❷
        generated_images = self.denormalize(generated_images) ❸
        return generated_images
```

Analysis of the Diffusion Model

2. Effect of Number of Diffusion Steps on Image Quality

The number of reverse diffusion steps affects the quality of generated images.

Fewer Diffusion Steps:

Noise is removed quickly.

The generated image may be unclear or blurry.

Details are not properly captured.

More Diffusion Steps:

Noise is removed gradually.

The model gets more chances to refine the image.

Image quality and details improve.

Example:

1 step → Blurred image with fewer details.

20 steps → Good quality image.

100 steps → Slight improvement but requires more time.

Therefore, **20 diffusion steps provide a good balance between quality and speed**.

Trade-off:

More steps → Better image quality but slower generation.

Fewer steps → Faster generation but lower quality.

Analysis of the Diffusion Model

2. Effect of Number of Diffusion Steps on Image Quality

.

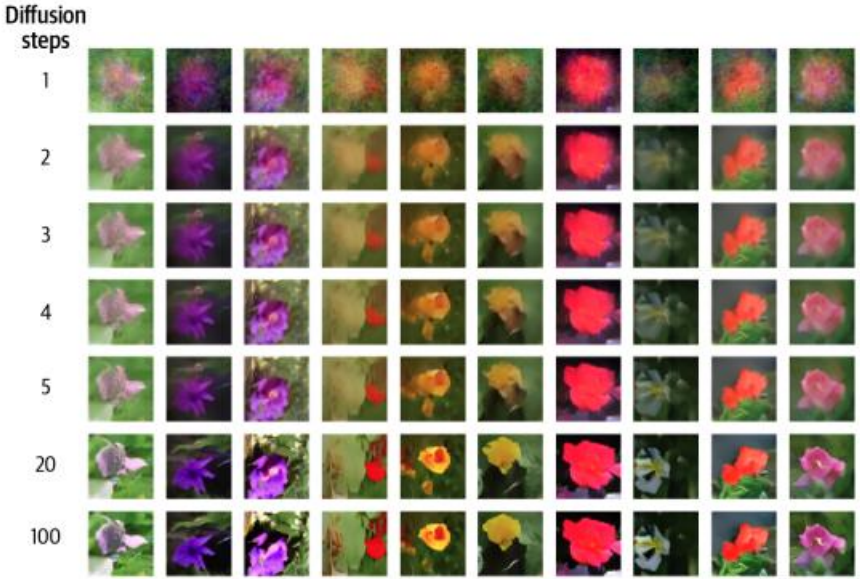


Figure 8-15. Image quality improves with the number of diffusion steps

Analysis of the Diffusion Model

3. Image Interpolation Using Diffusion Model

Image interpolation means creating a smooth transition between two images.

Process:

Two random noise patterns are selected.

The noise patterns are gradually combined.

The diffusion model generates intermediate images between them.

Example:

Image A:

Flower Image

↓

Intermediate Images:

Mixed features of Flower and another image

↓

Image B:

Another Image

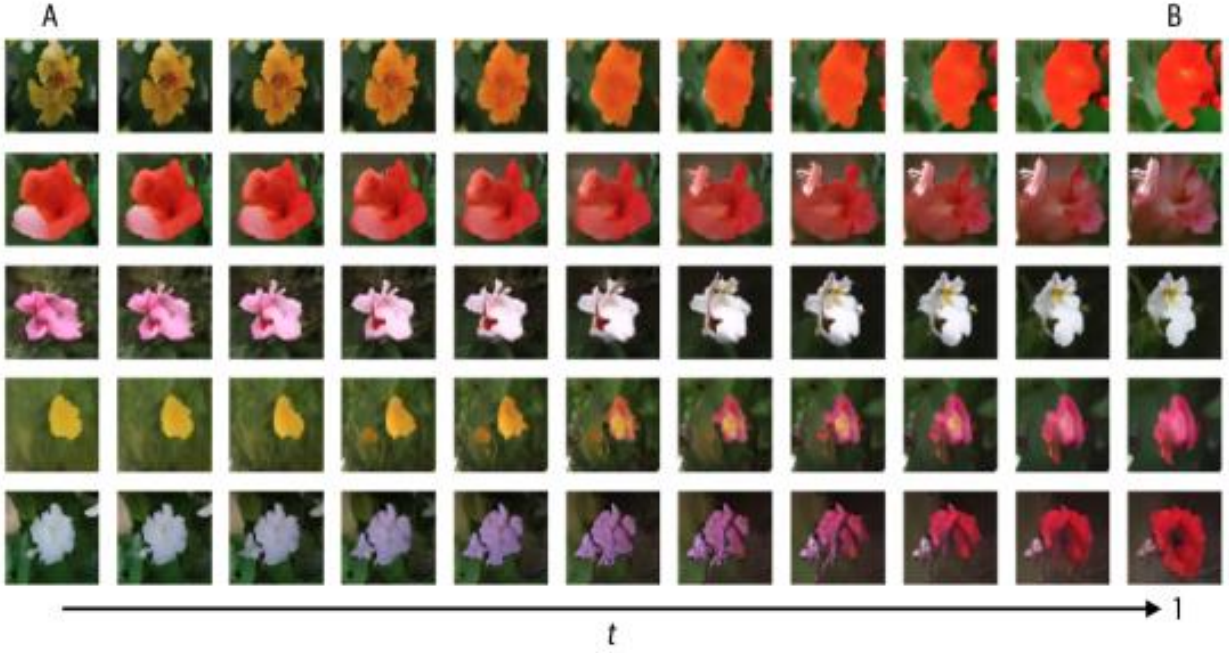


Figure 8-16. Interpolating between images using the denoising diffusion model