Diffusion and flow matching tutorial: How we generate images, video, speech and protein structures

Nando de Freitas and Everyone

May 11, 2026

1 Introduction

Diffusion and flow matching are the standard ways for generating images, video, speech, music and even protein structures and molecular simulations.

This tutorial aims at deriving the basic equations needed to understand these applications and the code behind them. Regardless of the modality, the code will be pretty much the same. While there are other derivations using variational methods, here we will simply use a fundamental learning principle: match imagination to reality. That is, what the model imagines, predicts, or generates, must match the real data. This is the principle used to train LLMs too, but while LLMs typically use the Maximum Likelihood principle, here we will use the Score Matching principle.

After explaining diffusion, we will derive a simple but very powerful approach based on conditional expectation and ordinary differential equations (ODEs). This will allow us to arrive at flow matching, where very deep neural networks can be interpreted as running ODEs with transformer blocks to generate data.

To understand diffusion from first principles, we first need to derive a loss function, which will then be used to train the generative model. The loss function is often reparameterised to make it numerically stable. The data for the loss function will consist of the original image and noisy samples generated by a forward diffusion process, as shown in Figure 1. Using these data, we will *train* a neural network to undo the process of adding noise. Finally, such a network will enable us to start with any random sample and reverse it until we get an image. We will refer to this reverse process as *inference*.

In this tutorial, we will first derive the loss function. Once we have the loss function, we can minimize it with standard gradient descent approaches, such as Adam. For inference, we will derive a Gaussian distribution for sampling (generating) any type of data using the trained neural network. The generation can be unconditional or conditioned on signals such as past video frames, text, pose, camera view, quality score, and so on.

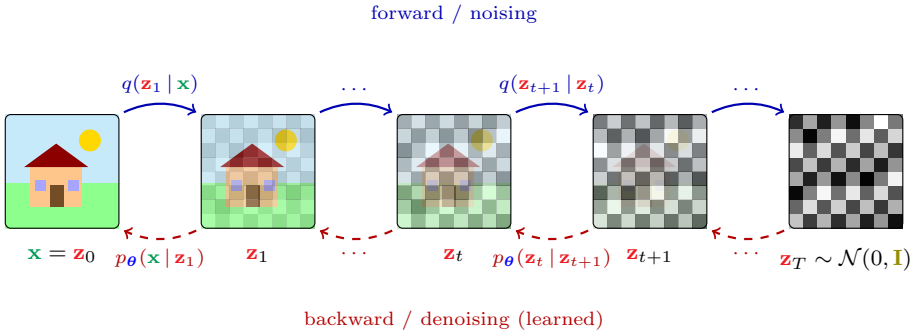

Figure 1: The two halves of a diffusion model. The forward process q (top, blue) takes a clean datapoint $\mathbf{x} = \mathbf{z}_0$ and gradually corrupts it into pure Gaussian noise $\mathbf{z}_T \sim \mathcal{N}(0, \mathbf{I})$ by adding a small amount of noise at each step. This direction is typically hand-designed: each transition is a Gaussian whose mean and variance are fixed by a schedule, with no learned parameters. The backward process $p_{\boldsymbol{\theta}}$ (bottom, red, dashed) goes the other way and is what the neural network learns: starting from pure noise it denoises step-by-step until it produces a sample. Sampling at inference time is just running the bottom row from right to left to produce new images, speech, videos or molecules.

2 Training

2.1 Matching imagination to reality

The data (images, proteins, videos, songs) will be represented with the generic vector $\mathbf{x}$. The real data is assumed to come from an unknown distribution $p_d(\mathbf{x})$. Since we don't have access to this distribution, we will try to approximate it using a model distribution $p_{\boldsymbol{\theta}}(\mathbf{x})$, with parameters $\boldsymbol{\theta}$. After learning the model distribution, also known as the generative model, we will be able to generate new data from it. Mathematically, the process of generation is represented as follows: $\mathbf{x} \sim p_{\boldsymbol{\theta}}(\mathbf{x})$.

The data distribution, that is the mechanism responsible for producing the images and videos we all see, is very complex. At first, it may seem that the task of generating data indistinguishable from real data might be impossible, and in fact 10 years ago most AI experts felt this was the case (no matter what they tell you - trust me, I've been doing this for 30 years and know almost everyone in the field). These days, however, we know that generative models using deep neural network representations are sufficiently flexible and powerful to solve the problem.

We want our model to assign the same probability as the world to all data configurations $\mathbf{x}$. That is, we want to minimize the difference between these two distributions on expectation over all the possible realizations of the data.

Mathematically, we express this with the following loss function:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{p_d(\mathbf{x})}\left[\frac{1}{2}\|p_{\boldsymbol{\theta}}(\mathbf{x}) - p_d(\mathbf{x})\|_2^2\right] \tag{1}$$

Learning, in this context, is all about finding a set of neural network parameters $\boldsymbol{\theta}$ such that this loss is minimized.

Matching what the model imagines (generates) to the data generated by the world seems like a natural goal for learning. However, this is hard because we cannot calculate probabilities for models directly (so we'll have to use auto-regression or, as we explain here, diffusion score matching). The reason we cannot calculate the probabilities has to do with the normalizing constant, which we explain in the following section.

2.2 Score matching

The model distribution $p_{\boldsymbol{\theta}}(\mathbf{x})$ can be expressed in a very general normalized exponential form:

$$p_{\boldsymbol{\theta}}(\mathbf{x}) = \frac{1}{Z(\boldsymbol{\theta})}e^{-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}$$

In this representation, Z is known as the normalizing constant or partition function. It ensures that over the whole set of values that the data can take, the model probability sums to 1 for all values of $\boldsymbol{\theta}$. The quantity in the exponent is known as the energy. Physicists often prefer to use the terminology of minimizing energy, but clearly this is equivalent to maximising the model probability. Maximising the probability of the data by modifying the model parameters is known as maximum likelihood.

The denominator sums over all possible images in the universe so that $p_{\boldsymbol{\theta}}(\mathbf{x})$ can be interpreted as a probability:

$$Z(\boldsymbol{\theta}) = \int_{\mathbf{x}} e^{-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}\mathrm{d}\mathbf{x}$$

This partition function is typically intractable because the sum is simply too large. It belongs to a complexity class known as sharp P, which in short means bloody hard if not impossible. For the practical applications we care about, we cannot do this sum. A decade ago we weren't too optimistic, but we have learned since then that it is actually possible to approximate this well for text, images, video, audio and other natural signals. One possible solution is to break the data x into small blocks and process each block auto-regressively (this is basically what LLMs do). An alternative is to do what we are about to learn to do in this document.

If we can't do the sum, let's get rid of the sum! We can do this by taking the log of the model probability and then computing its gradient (derivatives

with respect to the data) as follows:

$$
\begin{aligned}
p_{\boldsymbol{\theta}}(\mathbf{x}) &= \frac{1}{Z(\boldsymbol{\theta})} e^{-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})} \\
\log p_{\boldsymbol{\theta}}(\mathbf{x}) &= -\log Z(\boldsymbol{\theta}) - \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}) \\
\nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x}) &= -\nabla_{\mathbf{x}} \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})
\end{aligned}
$$

We will now reframe learning as matching the gradient of the model probability and the gradient of the distribution of the data. Intuitively, we want the rate of change in the modelled energy to match the rate of change of the real energy. This is known as score matching:

$$
\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{p_d(\mathbf{x})} \left[\| \nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x}) - \nabla_{\mathbf{x}} \log p_d(\mathbf{x}) \|_2^2 \right] \tag{2}
$$

Getting rid of Z is not enough. We still don't have an expression for the derivative of the data distribution: $\nabla_{\mathbf{x}} \log p_d(\mathbf{x})$. The model derivative $\nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x})$, known as the *score function*, can be easily calculated using backpropagation.

2.3 Denoised score matching

Assume instead that we can place a Gaussian $q(\cdot)$ concentrated on each data point $\mathbf{x}$ and then draw a sample $\mathbf{z}_t$. This Gaussian will have two scalar hyperparameters taking values between 0 and 1. A hyperparameter α_t will be used to scale the data, e.g. scale an image $\mathbf{x}$. A second hyperparameter σ_t^2 will determine the Gaussian variance.

When $\alpha_t = 0$ the Gaussian will have mean zero, and when $\alpha_t = 1$ the Gaussian will have mean $\mathbf{x}$. Later we will show how we can parameterise α_t so that by modifying the subindex t, α_t will vary from 1 to 0, and σ_t^2 in turn will vary from 0 to 1.

Mathematically, we express this process of adding noise to the data as follows:

$$
\mathbf{z}_t \sim q_t(\mathbf{z}_t|\mathbf{x}) = \mathcal{N}(\mathbf{z}_t|\alpha_t \mathbf{x}, \sigma_t^2 \mathbf{I})
$$

This sampling can be rewritten in the following equivalent form:

$$
\mathbf{z}_t = \alpha_t \mathbf{x} + \sigma_t \epsilon_t \tag{3}
$$

where

$$
\epsilon_t = \mathcal{N}(0, \mathbf{I}) \tag{4}
$$

In other words $\mathbf{z}_t$ is a bit like the image $\mathbf{x}$ and a bit like Gaussian noise ϵ_t.

We have introduced the index t because we will allow for sampling at multiple scales. At the very noisy scale, when $\alpha \approx 0$ and $\sigma \approx 1$, $\mathbf{z}_t$ will be basically Gaussian noise. At the other no noise extreme, when $\alpha \approx 1$ and $\sigma \approx 0$, $\mathbf{z}_t \approx \mathbf{x}$. We will choose a schedule to obtain samples between these two extremes.

All of the code that follows assumes the following imports and a single `DEVICE` variable that selects CUDA, Apple MPS, or CPU automatically:

```python
from __future__ import annotations
import math
from typing import Callable, Literal

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch import Tensor

if torch.cuda.is_available():
    DEVICE = torch.device("cuda")
elif torch.backends.mps.is_available():
    DEVICE = torch.device("mps")
else:
    DEVICE = torch.device("cpu")
```

The code for the denoising process is then:

```python
def add_noise(
    clean_sample: Tensor,
    alpha_t: Tensor,
    sigma_t: Tensor,
) -> tuple[Tensor, Tensor]:
    """Forward diffusion step: z_t = alpha_t * x + sigma_t * noise.

    Args:
        clean_sample: Clean data x, shape (batch, ...features).
        alpha_t: Per-sample signal coefficient, broadcastable to
        ↪  clean_sample.
        sigma_t: Per-sample noise coefficient, broadcastable to
        ↪  clean_sample.

    Returns:
        noisy_sample: z_t with the same shape as clean_sample.
        noise: The standard-normal noise that was added.
    """
    noise = torch.randn_like(clean_sample)
    noisy_sample = alpha_t * clean_sample + sigma_t * noise
    return noisy_sample, noise
```

The ratio of hyper-parameters is known as the signal-to-noise ratio:

$$SNR_t = \frac{\alpha_t^2}{\sigma_t^2}$$

We often use the log-SNR:

$$\lambda_t = \log SNR_t \tag{5}$$

With $\alpha_t = \cos(\pi t/2)$ and $\sigma_t = \sin(\pi t/2)$, we have $\alpha_t^2 + \sigma_t^2 = 1$, and hence $\lambda_t = -2 \log \tan(\pi t/2)$. This is known as the cosine schedule, and it is a very popular choice, e.g. by OpenAI. In code:

```python
class CosineNoiseSchedule:
    """Cosine log-SNR schedule (Nichol & Dhariwal, 2021), with optional shift.

    Maps a time variable t in [0, 1] to a log signal-to-noise ratio lambda_t,
    and from there to the diffusion coefficients alpha_t and sigma_t.

    The optional `shift` argument implements the "shifted cosine" from the
    Simple Diffusion paper, which adapts the schedule to the spatial
    resolution of the data when training on images. Concretely, set
    `shift = 2 * log(noise_d / image_d)`.
    """

    def __init__(
        self,
        log_snr_min: float = -15.0,
        log_snr_max: float = 15.0,
        shift: float = 0.0,
    ) -> None:
        self.log_snr_min = log_snr_min
        self.log_snr_max = log_snr_max
        self.shift = shift
        # Pre-compute the t-range bounds so log_snr is in [log_snr_min, log_snr_max].
        self._t_lo = math.atan(math.exp(-0.5 * log_snr_max))
        self._t_hi = math.atan(math.exp(-0.5 * log_snr_min))

    def log_snr(self, t: Tensor) -> Tensor:
        """Compute lambda_t = log(alpha_t^2 / sigma_t^2) for t in [0, 1]."""
        if torch.any((t < 0) | (t > 1)):
            raise ValueError("t must be in [0, 1].")
        clipped_t = self._t_lo + t * (self._t_hi - self._t_lo)
        cosine_log_snr = -2.0 * torch.log(torch.tan(clipped_t))
        return cosine_log_snr + self.shift

    def alpha_sigma(self, log_snr_t: Tensor) -> tuple[Tensor, Tensor]:
        """Convert log-SNR to (alpha_t, sigma_t) using the numerically stable
        identities alpha_t^2 = sigmoid(log_snr_t), sigma_t^2 = sigmoid(-log_snr_t)."""
        alpha_t = torch.sqrt(torch.sigmoid(log_snr_t))
        sigma_t = torch.sqrt(torch.sigmoid(-log_snr_t))
```

```
        return alpha_t, sigma_t
```

The simple diffusion paper argues for shifting the schedule by $2 \log(\text{noise\_d}/\text{image\_d})$ when training at higher spatial resolutions; the `shift` argument above implements this directly, so a single class covers both the standard and shifted-cosine cases.

The hyper-parameters α and σ can also be obtained from the log signal-to-noise ratio as follows:

$$\alpha_t^2 = \frac{1}{1 + e^{-\lambda_t}} \tag{6}$$

$$\sigma_t^2 = \frac{1}{1 + e^{\lambda_t}} \tag{7}$$

With this, we still have $\alpha_t^2 + \sigma_t^2 = 1$. We can thus express the code in terms of λ_t.

2.4 The score matching objective function

Let us define the score as follows:

$$\mathbf{s}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) = \nabla_{\mathbf{z}} \log p_{\boldsymbol{\theta}}(\mathbf{z}_t | \lambda_t) \tag{8}$$

The score is the function that we will approximate with a neural network with parameters $\boldsymbol{\theta}$. That is, the neural net would take the tuple $(\mathbf{z}_t, \lambda_t)$ as input and output $\mathbf{s}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t)$.

In terms of the samples $\mathbf{z}_t$, the denoising score matching objective is:

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\theta}; \lambda_t) &= \mathbb{E}_{q_{\lambda_t}(\mathbf{z}_t)} \left[\left\| \mathbf{s}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) - \nabla_{\mathbf{z}_t} \log q_{\lambda_t}(\mathbf{z}_t) \right\|_2^2 \right] \\
&= \mathbb{E}_{q_{\lambda_t}(\mathbf{z}_t)} \left[\left\| \mathbf{s}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) - \frac{\nabla_{\mathbf{z}_t} \int_{\mathbf{x}} q_{\lambda_t}(\mathbf{z}_t|\mathbf{x}) p_d(\mathbf{x}) d\mathbf{x}}{q_{\lambda_t}(\mathbf{z}_t)} \right\|_2^2 \right] \\
&= \mathbb{E}_{q_{\lambda_t}(\mathbf{z}_t)} \left[\left\| \mathbf{s}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) - \frac{\int_{\mathbf{x}} q_{\lambda_t}(\mathbf{z}_t|\mathbf{x}) \nabla_{\mathbf{z}_t} \log q_{\lambda_t}(\mathbf{z}_t|\mathbf{x}) p_d(\mathbf{x}) d\mathbf{x}}{q_{\lambda_t}(\mathbf{z}_t)} \right\|_2^2 \right] \\
&= \mathbb{E}_{q_{\lambda_t}(\mathbf{z}_t)} \left[\left\| \mathbf{s}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) - \mathbb{E}_{q_{\lambda_t}(\mathbf{x}|\mathbf{z}_t)} \nabla_{\mathbf{z}_t} \log q_{\lambda_t}(\mathbf{z}_t|\mathbf{x}) \right\|_2^2 \right] \\
&= \mathbb{E}_{q_{\lambda_t}(\mathbf{z}_t|\mathbf{x}) p_d(\mathbf{x})} \left[\left\| \mathbf{s}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) - \nabla_{\mathbf{z}_t} \log q_{\lambda_t}(\mathbf{z}_t|\mathbf{x}) \right\|_2^2 \right] + \text{const} \\
&= \mathbb{E}_{p_d(\mathbf{x})} \mathbb{E}_{\mathbf{z}_t \sim \mathcal{N}(\mathbf{z}_t | \alpha_t \mathbf{x}, \sigma_t^2 \mathbf{I})} \left\{ \left\| \mathbf{s}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) + \frac{1}{\sigma_t^2} (\mathbf{z}_t - \alpha_t \mathbf{x}) \right\|_2^2 \right\} + \text{const}
\end{aligned}$$

where in the fourth equation, we swapped the gradient and integral operator and then use the equality $\frac{p(\mathbf{x})}{p(\mathbf{z}_t)} = \frac{p(\mathbf{x}|\mathbf{z}_t)}{p(\mathbf{z}_t|\mathbf{x})}$. In the fifth equation, we move the conditional expectation $\mathbb{E}_{q_{\lambda_t}(\mathbf{x}|\mathbf{z}_t)}$ out of the square and the constant term does

not depend on $\mathbf{s}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t)$. In the last equation, we used the fact that, for a Gaussian, the score is given by

$$\nabla_{\mathbf{z}_t} \log \mathcal{N}(\mathbf{z}_t | \alpha_t \mathbf{x}, \sigma_t^2 \mathbf{I}) = -\nabla_{\mathbf{z}_t} \frac{1}{2\sigma_t^2} (\mathbf{z}_t - \alpha_t \mathbf{x})^T (\mathbf{z}_t - \alpha_t \mathbf{x}) = \frac{1}{\sigma_t^2} (\mathbf{z}_t - \alpha_t \mathbf{x})$$

Assuming that we have T different noise scales, that is $t = 0, 1, \ldots, T$, we can combine the losses in a weighted fashion using

$$\mathcal{L}(\boldsymbol{\theta}; \lambda_1, \lambda_2, \ldots, \lambda_T) = \sum_{t=1}^{T} \rho_t \mathcal{L}(\boldsymbol{\theta}; \lambda_t) \tag{9}$$

where we choose $\sigma_1 > \sigma_2 > \cdots > \sigma_T$, and the weighting term satisfies $\rho_t > 0$.

2.5 Re-Parameterisations of the Loss

From before, we know that:

$$\boldsymbol{\epsilon}_t = \frac{1}{\sigma_t} (\mathbf{z}_t - \alpha_t \mathbf{x})$$

so the loss can be rewritten directly in terms of the noise:

$$\begin{aligned} \mathcal{L}(\boldsymbol{\theta}; \lambda_t) &= \mathbb{E}_{\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \mathbf{x} \sim p_d(\mathbf{x}), t \sim \mathcal{U}_{[0,T]}} \left\{ \| \mathbf{s}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) + \frac{1}{\sigma_t} \boldsymbol{\epsilon}_t \|_2^2 \right\} \\ &= \mathbb{E}_{\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \mathbf{x} \sim p_d(\mathbf{x}), t \sim \mathcal{U}_{[0,T]}} \left\{ \frac{1}{\sigma_t^2} \| \boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) - \boldsymbol{\epsilon}_t \|_2^2 \right\} \end{aligned}$$

where we replaced the (arbitrary) neural network model $\mathbf{s}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t)$ with another neural network

$$\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) = -\sigma_t \mathbf{s}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) = -\sigma_t \nabla_{\mathbf{z}} \log p_{\boldsymbol{\theta}}(\mathbf{z}_t | \lambda_t)$$

That is, the neural network now predicts $\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t)$ directly by taking $\mathbf{z}_t$ and the log SNR as input. Papers often choose the scaling term $\rho_t = \sigma_t^2$, so the loss takes the following simple form:

$$\mathcal{L}(\boldsymbol{\theta}; \lambda_t) = \mathbb{E} \left\{ \| \boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) - \boldsymbol{\epsilon}_t \|_2^2 \right\} \tag{10}$$

to obtain the total loss. Note that we are now using the shorthand notation $\mathbb{E}$ to summarize $\mathbb{E}_{\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \mathbf{x} \sim p_d(\mathbf{x}), t \sim \mathcal{U}_{[0,T]}}$.

The neural network here is most often a *Diffusion Transformer (DiT)* (Peebles & Xie, 2022). At each call, $\mathbf{z}_t$ enters as a sequence of patch embeddings; the scalar log signal-to-noise ratio λ_t is first lifted to a vector by a small embedding network (sinusoidal/Fourier features followed by an MLP), and the text caption $\mathbf{c}$ is encoded into a sequence of feature vectors by a text-only transformer encoder (e.g. a frozen T5 or CLIP text encoder, in the style of Imagen and

Stable Diffusion 3). Both conditioning signals are then injected into the DiT's transformer blocks — through adaptive layer-norm or through cross-attention. The network outputs the predicted noise $\hat{\epsilon}_{\boldsymbol{\theta}}$ at this noise level. This is exactly one call of the network as used inside `ancestral_sampling_step` which will be described later.

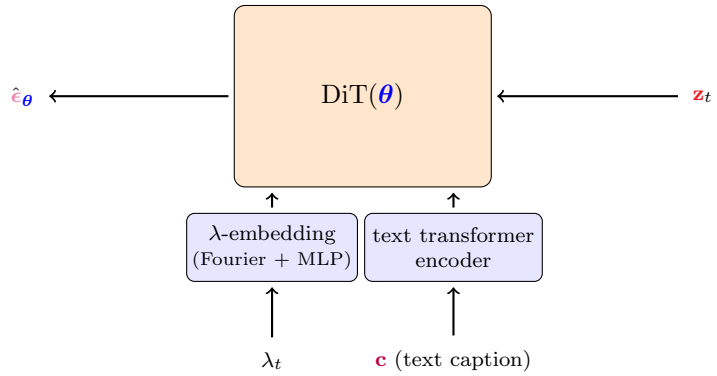

Figure 2: One call of the diffusion network, as used inside `ancestral_sampling_step`. The noisy latent $\mathbf{z}_t$ enters from the right; the scalar log-SNR λ_t and the text caption $\mathbf{c}$ enter from below, after each is lifted to a feature vector / sequence by its own encoder (λ-embedding for the scalar, text transformer encoder for the caption). The DiT consumes all three and emits the predicted noise $\hat{\epsilon}_{\boldsymbol{\theta}}$ on the left, which is exactly the `network_output` consumed by the sampling step. At training time, $\hat{\epsilon}_{\boldsymbol{\theta}}$ is compared against the true ϵ_t via MSE; at inference time, it feeds the next sampling step. The same box, with $\hat{\epsilon}_{\boldsymbol{\theta}}$ swapped for $\hat{\mathbf{v}}_{\boldsymbol{\theta}}$ or $\hat{\mathbf{x}}_{\boldsymbol{\theta}}$, describes the $\mathbf{v}$- and $\mathbf{x}$-parameterisations introduced below.

If the models were consistent, we would expect the same relation that we used for sampling ($\mathbf{z}_t = \alpha_t \mathbf{x} + \sigma_t \epsilon_t$) to be true for the models:

$$\mathbf{x}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) = \frac{1}{\alpha_t}\left(\mathbf{z}_t - \sigma_t \epsilon_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t)\right) = \frac{1}{\alpha_t}\left(\mathbf{z}_t - \sigma_t^2 \nabla_{\mathbf{z}} \log p_{\boldsymbol{\theta}}(\mathbf{z}_t|\lambda_t)\right) \tag{11}$$

where $\mathbf{x}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t)$ may be interpreted as the generated data. We can simply start from this assumption by construction.

By predicting $\epsilon_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t)$ we can reconstruct $\mathbf{x}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t)$. Substituting the expressions for $\mathbf{x}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t)$ and $\mathbf{x}$ into the loss expression yields the following

$$\mathcal{L}(\boldsymbol{\theta}; \lambda_t) = \mathbb{E}\left\{\frac{\alpha_t^2}{\sigma_t^4}\left\|\mathbf{x}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) - \mathbf{x}\right\|_2^2\right\}$$

This loss enables us to see more clearly that we are after a model that can reconstruct the data from the noise samples $\mathbf{z}$.

As pointed out by Tim Salimans and Jonathan Ho in their Progressive Distillation work for diffusion models, when the SNR is low, that is when σ is large compared to α, any small changes to the neural net prediction $\epsilon_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t)$ will result in big changes in the reconstructions because

$$\mathbf{x}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) = \frac{1}{\alpha_t}\mathbf{z}_t - \frac{\sigma_t}{\alpha_t}\epsilon_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t). \tag{12}$$

The less diffusion steps (lower T), the bigger the problem. To overcome this, they propose the following re-parameterisation:

$$\mathbf{v}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) = \alpha_t\epsilon_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) - \sigma_t\mathbf{x}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) \tag{13}$$

Under this parameterisation, and with $\alpha_t^2 + \sigma_t^2 = 1$, one can easily show that:

$$\mathbf{x}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) = \alpha_t\mathbf{z}_t - \sigma_t\mathbf{v}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) \tag{14}$$

$$\epsilon_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) = \sigma_t\mathbf{z}_t + \alpha_t\mathbf{v}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) \tag{15}$$

In the high-noise regime, we have

$$\mathbf{x}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) \approx -\mathbf{v}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) \quad \textit{and} \quad \epsilon_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) \approx \mathbf{z}_t. \tag{16}$$

On the other hand, in the low-noise regime, we have

$$\mathbf{x}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) \approx \mathbf{z}_t \quad \textit{and} \quad \epsilon_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) \approx \mathbf{v}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t). \tag{17}$$

The first equation for $\mathbf{x}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t)$ will be useful for generating new data, while the one for $\epsilon_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t)$ is necessary for minimizing the score matching loss.

The pytorch code for computing the loss function, using a neural network model in the same class is as follows:

```
Parameterisation = Literal["v", "eps"]


def diffusion_loss(
    score_network: nn.Module,
    clean_batch: Tensor,
    schedule: CosineNoiseSchedule,
    parameterisation: Parameterisation = "v",
    condition: Tensor | None = None,
) -> Tensor:
    """Denoising score-matching loss for a diffusion model.

    For each sample in the batch we draw an independent time t ~ U[0,
    ↪  1],
    add the corresponding noise, ask the network for either the noise
    ↪  or
```

```
    the velocity, convert to a noise prediction, and compute MSE against
    the true noise.
    """
    batch_size = clean_batch.shape[0]
    device = clean_batch.device

    # Sample one t per example, then look up alpha_t, sigma_t, lambda_t.
    t = torch.rand(batch_size, device=device)
    log_snr_t = schedule.log_snr(t)
    alpha_t, sigma_t = schedule.alpha_sigma(log_snr_t)

    # Reshape the schedule scalars so they broadcast over the feature dims.
    feature_shape = (-1,) + (1,) * (clean_batch.ndim - 1)
    alpha_t = alpha_t.view(feature_shape)
    sigma_t = sigma_t.view(feature_shape)

    # Forward noising.
    noisy_sample, true_noise = add_noise(clean_batch, alpha_t, sigma_t)

    # Network forward pass.
    network_output = score_network(noisy_sample, log_snr_t, condition)

    # Convert whatever the network predicts into a noise prediction so we can
    # compare against `true_noise` with the same loss in both parameterisations.
    if parameterisation == "v":
        predicted_noise = sigma_t * noisy_sample + alpha_t * network_output
    elif parameterisation == "eps":
        predicted_noise = network_output
    else:
        raise ValueError(f"Unknown parameterisation: {parameterisation!r}")

    return F.mse_loss(predicted_noise, true_noise)
```

The code above allows one to use the epsilon-parameterisation or the v-parameterisation. This is all we need to train the model, which can be a UViT, a UNet or any transformer. We simply need to minimize this loss to learn the neural net parameters.

The model can also have *control inputs*, which can be a text prompt, quality score, camera motion, previous or future frames of a video, and so on.

Next, we discuss inference, that is how to start at noise and then progressively sample new images in the reverse process.

3 Inference

Our way of sampling, can been seen as a forward noising mechanism:

$$\mathbf{z}_t \sim q(\mathbf{z}_t|\mathbf{x}) = \mathcal{N}(\mathbf{z}_t|\alpha_t\mathbf{x}, \sigma_t^2\mathbf{I})$$

Using Bayes' rule, we can obtain a way of sampling backward, to eliminate the noise as follows:

$$q(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{x}) = \frac{q(\mathbf{z}_{t+1}|\mathbf{z}_t, \mathbf{x})q(\mathbf{z}_t|\mathbf{x})}{q(\mathbf{z}_{t+1}|\mathbf{x})} = \frac{q(\mathbf{z}_{t+1}|\mathbf{z}_t)q(\mathbf{z}_t|\mathbf{x})}{q(\mathbf{z}_{t+1}|\mathbf{x})}$$

With a Markovian forward process, we have $q(\mathbf{z}_{t+1}|\mathbf{z}_t, \mathbf{x}) = q(\mathbf{z}_{t+1}|\mathbf{z}_t)$. The distributions $q(\mathbf{z}_t|\mathbf{x})$ and $q(\mathbf{z}_{t+1}|\mathbf{x})$ are Gaussian by definition. It can be shown that the distribution $q(\mathbf{z}_{t+1}|\mathbf{z}_t)$ is also Gaussian. Specifically,

$$q(\mathbf{z}_{t+1}|\mathbf{z}_t) = \mathcal{N}\left(\mathbf{z}_{t+1}; \frac{\alpha_{t+1}}{\alpha_t}\mathbf{z}_t, \sigma_{t+1}^2 - \frac{\alpha_{t+1}^2}{\alpha_t^2}\sigma_t^2\mathbf{I}\right) \tag{18}$$

To see this, note that unrolling this expression we have:

$$\begin{aligned}
\mathbf{z}_{t+1} &= \frac{\alpha_{t+1}}{\alpha_t}\mathbf{z}_t + \sqrt{\sigma_{t+1}^2 - \frac{\alpha_{t+1}^2}{\alpha_t^2}\sigma_t^2}\epsilon_{t+1} \\
&= \frac{\alpha_{t+1}}{\alpha_t}\frac{\alpha_t}{\alpha_{t-1}}\mathbf{z}_{t-1} + \frac{\alpha_{t+1}}{\alpha_t}\sqrt{\sigma_t^2 - \frac{\alpha_t^2}{\alpha_{t-1}^2}\sigma_{t-1}^2}\epsilon_t + \sqrt{\sigma_{t+1}^2 - \frac{\alpha_{t+1}^2}{\alpha_t^2}\sigma_t^2}\epsilon_{t+1} \\
&= \frac{\alpha_{t+1}}{\alpha_{t-1}}\mathbf{z}_{t-1} + \sqrt{\sigma_{t+1}^2 - \frac{\alpha_{t+1}^2}{\alpha_{t-1}^2}\sigma_{t-1}^2}\tilde{\epsilon}_{t+1}
\end{aligned}$$

where to get the last line we have used that the sum of two normal random variables is normal with the appropriate parameters. Intuitively, if we continue expanding this recursion we will arrive at our expression for $q(\mathbf{z}_t|\mathbf{x}) = \mathcal{N}(\mathbf{z}_t|\alpha_t\mathbf{x}, \sigma_t^2\mathbf{I})$ as expected:

$$\begin{aligned}
\mathbf{z}_{t+1} &= \frac{\alpha_{t+1}}{\alpha_0}\mathbf{z}_0 + \sqrt{\sigma_{t+1}^2 - \frac{\alpha_{t+1}^2}{\alpha_0^2}\sigma_0^2}\tilde{\epsilon}_{t+1} \\
&= \alpha_{t+1}\mathbf{z}_0 + \sigma_{t+1}\tilde{\epsilon}_{t+1} \\
&= \alpha_{t+1}\mathbf{x} + \sigma_{t+1}\tilde{\epsilon}_{t+1}
\end{aligned}$$

We have used the fact that by construction, when the noise vanishes, we have $\alpha_0 = 1$, $\sigma_0 = 0$ and $\mathbf{z}_0 = \mathbf{x}$.

We can now apply Bayes' rule to obtain an expression for the normal distribution $q(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{x})$. The algebra involved in this is about completing the square of the exponent of the Gaussian. There is a derivation of this in the Appendix.

The resulting Gaussian distribution is:

$$
\begin{aligned}
q(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{x}) &= \mathcal{N}\left(\mathbf{z}_t; \frac{\alpha_{t+1}}{\alpha_t}\frac{\sigma_t^2}{\sigma_{t+1}^2}\mathbf{z}_{t+1} + \left(\sigma_{t+1}^2 - \frac{\alpha_{t+1}^2}{\alpha_t^2}\sigma_t^2\right)\frac{\alpha_t}{\sigma_{t+1}^2}\mathbf{x}, \left(\sigma_{t+1}^2 - \frac{\alpha_{t+1}^2}{\alpha_t^2}\sigma_t^2\right)\frac{\sigma_t^2}{\sigma_{t+1}^2}\mathbf{I}\right) \\
&= \mathcal{N}\left(\mathbf{z}_t; \frac{\alpha_t}{\alpha_{t+1}}\frac{\alpha_{t+1}^2}{\sigma_{t+1}^2}\frac{\sigma_t^2}{\alpha_t^2}\mathbf{z}_{t+1} + \left(1 - \frac{\alpha_{t+1}^2}{\sigma_{t+1}^2}\frac{\sigma_t^2}{\alpha_t^2}\right)\alpha_t\mathbf{x}, \left(1 - \frac{\alpha_{t+1}^2}{\sigma_{t+1}^2}\frac{\sigma_t^2}{\alpha_t^2}\right)\sigma_t^2\mathbf{I}\right) \\
&= \mathcal{N}\left(\mathbf{z}_t; \frac{\alpha_t}{\alpha_{t+1}}e^{\lambda_{t+1}-\lambda_t}\mathbf{z}_{t+1} + \left(1 - e^{\lambda_{t+1}-\lambda_t}\right)\alpha_t\mathbf{x}, \left(1 - e^{\lambda_{t+1}-\lambda_t}\right)\sigma_t^2\mathbf{I}\right) \\
&= \mathcal{N}\left(\mathbf{z}_t; \alpha_t\left[\frac{(1-c)}{\alpha_{t+1}}\mathbf{z}_{t+1} + c\mathbf{x}\right], c\sigma_t^2\mathbf{I}\right) \qquad (19)
\end{aligned}
$$

where we used the notation $\lambda_t = \log \alpha_t^2/\sigma_t^2$ for the log SNR. The above distribution is what we use for drawing samples. We use the score matching loss to learn the approximation

$$
\mathbf{x}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) = \alpha_t\mathbf{z}_t - \sigma_t\mathbf{v}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) \qquad (20)
$$

and replace $\mathbf{x}_t$ with $\mathbf{x}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t)$ in $q(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{x})$ in order to draw novel samples. That is,

$$
q(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{x}) = \mathcal{N}\left(\mathbf{z}_t; \alpha_t\left[\frac{(1-c)}{\alpha_{t+1}}\mathbf{z}_{t+1} + c\mathbf{x}_{\boldsymbol{\theta}}(\mathbf{z}_{t+1}, \lambda_{t+1})\right], c\sigma_t^2\mathbf{I}\right) \qquad (21)
$$

In code, we implement this with the following two functions:

```python
@torch.no_grad()
def ancestral_sampling_step(
    noisy_sample: Tensor,
    network_output: Tensor,
    log_snr_t: Tensor,
    log_snr_s: Tensor,
    parameterisation: Parameterisation,
    clip_to: tuple[float, float] | None = (-1.0, 1.0),
) -> tuple[Tensor, Tensor]:
    """One reverse step of ancestral sampling, from noise level t down to s < t.

    Returns the mean and (scalar) variance of q(z_s | z_t, x_hat), where
    x_hat is reconstructed from the network output.
    """
    # -expm1(x) = 1 - exp(x); positive when log_snr_t < log_snr_s.
    c = -torch.expm1(log_snr_t - log_snr_s)

    alpha_t = torch.sqrt(torch.sigmoid(log_snr_t))
    alpha_s = torch.sqrt(torch.sigmoid(log_snr_s))
```

```
    sigma_t = torch.sqrt(torch.sigmoid(-log_snr_t))
    sigma_s = torch.sqrt(torch.sigmoid(-log_snr_s))

    # Decode the network output into a prediction of the clean sample
    ↪  x.
    if parameterisation == "v":
        predicted_clean = alpha_t * noisy_sample - sigma_t *
        ↪  network_output
    elif parameterisation == "eps":
        predicted_clean = (noisy_sample - sigma_t * network_output) /
        ↪  alpha_t
    else:
        raise ValueError(f"Unknown parameterisation:
        ↪  {parameterisation!r}")

    if clip_to is not None:
        predicted_clean = predicted_clean.clamp(*clip_to)

    posterior_mean = alpha_s * (
        noisy_sample * (1.0 - c) / alpha_t + c * predicted_clean
    )
    posterior_variance = (sigma_s ** 2) * c
    return posterior_mean, posterior_variance
```

The corresponding code is just a Python `for` loop over timesteps, calling the same network at each iteration:

```
@torch.no_grad()
def generate_samples(
    score_network: nn.Module,
    schedule: CosineNoiseSchedule,
    sample_shape: tuple[int, ...],
    num_steps: int = 100,
    parameterisation: Parameterisation = "v",
    condition: Tensor | None = None,
    clip_to: tuple[float, float] | None = None,
    device: torch.device = DEVICE,
) -> Tensor:
    """Generate a batch of samples by running ancestral sampling
    ↪  end-to-end.

    Starts from pure Gaussian noise at t=1 and iterates the reverse
    ↪  step
    down to t=0 on a uniform grid of `num_steps` points.
    """
    sample = torch.randn(sample_shape, device=device)

    for step in reversed(range(1, num_steps + 1)):
        t_now = torch.tensor(step / num_steps, device=device)
        t_next = torch.tensor((step - 1) / num_steps, device=device)
```

```
        log_snr_t = schedule.log_snr(t_now)
        log_snr_s = schedule.log_snr(t_next)

        # The network expects a per-sample log-SNR; broadcast the
        ↪  scalar.
        log_snr_t_batched = log_snr_t.expand(sample.shape[0])
        network_output = score_network(sample, log_snr_t_batched,
        ↪  condition)

        mean, variance = ancestral_sampling_step(
            noisy_sample=sample,
            network_output=network_output,
            log_snr_t=log_snr_t,
            log_snr_s=log_snr_s,
            parameterisation=parameterisation,
            clip_to=clip_to,
        )

        if step > 1:
            # Stochastic step: add noise scaled by sqrt(variance).
            sample = mean + torch.randn_like(mean) *
            ↪  torch.sqrt(variance)
        else:
            # Last step: take the mean (deterministic) to avoid noisy
            ↪  outputs.
            sample = mean

    return sample
```

During generation, it is helpful to clip the predicted clean sample $\hat{\mathbf{x}}$ in $\mathbf{x}$-space, which the `ancestral_sampling_step` function above does via the `clip_to` argument (e.g. `clip_to=(-1, 1)` when the data is normalised to $[-1, 1]$).

The structure of the loop in `generate_samples` above is exactly that of an unrolled recurrent network with tied parameters: the same DiT (same $\boldsymbol{\theta}$) is called at every step, the only things that change between steps are the latent $\mathbf{z}_t$ and the noise level λ_t. Figure 3 unrolls four iterations of the loop.

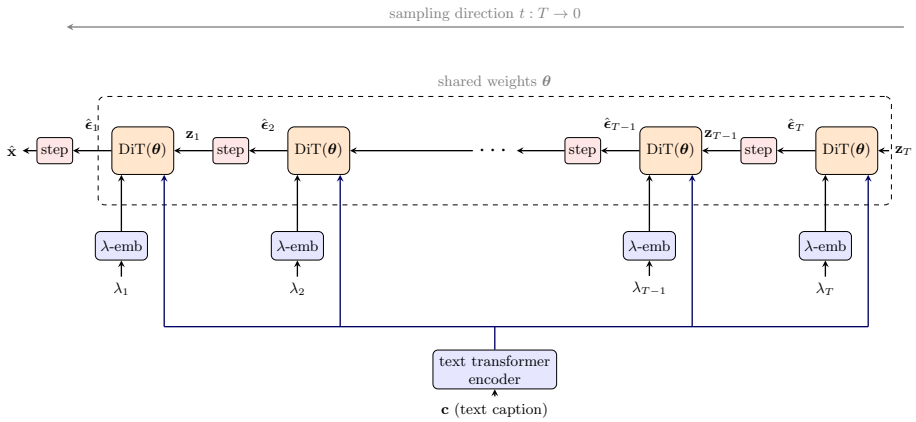

Figure 3: Inference (`generate_samples`) unrolled as a tied-parameter recurrent neural network (RNN). The *same* DiT (same weights $\boldsymbol{\theta}$) is applied at every step; what changes between steps is (i) the input latent $\mathbf{z}_t$ on the right of each block and (ii) the noise level λ_t injected from below through the λ-embedding (one fresh λ_t per step). The text caption $\mathbf{c}$ is encoded *once* by a text transformer encoder and the same conditioning is fanned out (blue bus) to every DiT call. Each DiT emits a noise prediction $\hat{\boldsymbol{\epsilon}}_t$, which is consumed by the small red `step` box. This is exactly `ancestral_sampling_step`. Time runs right-to-left: we start from pure Gaussian noise $\mathbf{z}_T$ on the right and, after T DiT calls, read off the generated sample $\hat{\mathbf{x}} = \mathbf{z}_0$ on the left.

Connection to looped and feedback language models. Figure 3 is also a useful way to view recent recurrent-depth LLMs. The diffusion sampler is a *loop over latent state*: the same transformer weights are reused, and each pass refines a hidden object, here the noisy latent $\mathbf{z}_t$. In Looped Language Models (Zhu et al., 2025), the same idea is applied inside an LLM: a shared recurrent block performs several latent "thinking" updates before emitting tokens, and the model can allocate more recurrent depth to harder examples. Earlier, the Feedback Transformer (Fan et al., 2020) used a related feedback idea along the sequence direction: future token representations can read high-level representations computed for previous tokens, rather than only lower-layer states. The analogy is not exact, but it is instructive. In Figure 3, the recurrence index is diffusion time and the state is an external sample $\mathbf{z}_t$; in looped LLMs the recurrence index is latent reasoning depth; in feedback transformers the recurrent signal is memory from earlier sequence positions. In all three cases, the core move is parameter sharing across repeated computation, which turns a feed-forward transformer block into an iterative refinement machine.

4 The important details

The preceding two sections have all the code needed to train a diffusion model, and to generate new images with it at inference time. Some details are missing. Sometimes we use encoders and decoders to project the images and videos to lower dimensional spaces. We have abstracted all the details of the neural network model, and in particular how we condition. We have not discussed the effect of scale. Most importantly, we have not discussed data. Getting good captions for the images and conditioning on these is of paramount importance to attain SoTA models.

4.1 Classifier free guidance

Typically we have text captions c and want to generate images or videos that match such captions. In *classifier-free guidance*, we use Bayes' rule to compute $p(\mathbf{y}|\mathbf{x}) = p(\mathbf{x}|\mathbf{y})p(\mathbf{y})/p(\mathbf{x})$, and hence

$$\log p(\mathbf{y}|\mathbf{x}) = \log p(\mathbf{x}|\mathbf{y}) + \log p(\mathbf{y}) - \log p(\mathbf{x}) \tag{22}$$

so the score of the induced classifier becomes

$$\nabla_{\mathbf{x}} \log p(\mathbf{y}|\mathbf{x}) = \nabla_{\mathbf{x}} \log p(\mathbf{x}|\mathbf{y}) - \nabla_{\mathbf{x}} \log p(\mathbf{x}) \tag{23}$$

Plugging in the scaled guidance signal gives

$$\nabla_{\mathbf{x}} \log p_w(\mathbf{x}|\mathbf{y}) = \nabla_{\mathbf{x}} \log p(\mathbf{x}) + w(\nabla_{\mathbf{x}} \log p(\mathbf{x}|\mathbf{y}) - \nabla_{\mathbf{x}} \log p(\mathbf{x})) \tag{24}$$

$$= (1-w)\nabla_{\mathbf{x}} \log p(\mathbf{x}) + w\nabla_{\mathbf{x}} \log p(\mathbf{x}|\mathbf{y}) \tag{25}$$

For $w = 0$ we recover the unconditional model, and for $w = 1$ we recover the standard conditional model.

In practice, we can implement this method with a small change to the ancestral sampling routine. At each step t, we predict the guided error term, which has the form

$$\tilde{\mathbf{v}}_t = w\mathbf{v}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t, \mathbf{y}) + (1-w)\mathbf{v}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t, \emptyset) \tag{26}$$

That is, instead of training a conditional and unconditional model, we can just train a single conditional model, provided we set $\mathbf{y} = \emptyset$ with probability $1 - w$ to emulate unconditional sampling. The advantage of this approach is that we have a single, self-consistent model, where the guidance signal is derived from a generative model, for which gradients in input space are more meaningful.

4.2 Classifier guidance

An alternative to introduce guidance, is to replace $\mathbf{x}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t, \mathbf{y}_t)$ with the corresponding score parameterisation:

$$
\begin{aligned}
q(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{x}) &= \mathcal{N}\left(\mathbf{z}_t; \alpha_t \left[\frac{(1-c)}{\alpha_{t+1}}\mathbf{z}_{t+1} + c\mathbf{x}_{\boldsymbol{\theta}}(\mathbf{z}_{t+1}, \lambda_{t+1}, \mathbf{y})\right], c\sigma_t^2\mathbf{I}\right) \\
&= \mathcal{N}\left(\mathbf{z}_t; \left[\frac{\alpha_t(1-c)}{\alpha_{t+1}}\mathbf{z}_{t+1} + c\left(\mathbf{z}_{t+1} - \sigma_t^2\nabla_{\mathbf{z}}\log p_{\boldsymbol{\theta}}(\mathbf{z}_{t+1}|\lambda_{t+1}, \mathbf{y})\right)\right], c\sigma_t^2\mathbf{I}\right) \\
&\equiv \mathcal{N}\left(\mathbf{z}_t; \left[\frac{\alpha_t(1-c)}{\alpha_{t+1}}\mathbf{z}_{t+1} + c\left(\mathbf{z}_{t+1} - \sigma_t^2(\nabla_{\mathbf{z}}\log p_{\boldsymbol{\theta}}(\mathbf{z}_{t+1}|\lambda_{t+1}) + \nabla_{\mathbf{z}}\log p_{\boldsymbol{\theta}}(\mathbf{y}|\mathbf{z}_{t+1}))\right)\right], c\sigma_t^2\mathbf{I}\right)
\end{aligned}
$$

The challenge with this approach is that we need to train a classifier $p_{\boldsymbol{\theta}}(\mathbf{y}|\mathbf{z}_{t+1})$ that works well for each level of noise. This is a lot like the old inception work of Google – look for those inception patterns in generated images to reveal whether this technique has been used.

5 Connection with DDPM parameterisation

We started with the forward equation

$$\mathbf{z}_t = \alpha_t\mathbf{x} + \sigma_t\epsilon_t$$

and derived the following two expressions:

$$q(\mathbf{z}_{t+1}|\mathbf{z}_t) = \mathcal{N}\left(\frac{\alpha_{t+1}}{\alpha_t}\mathbf{z}_t, \left(\sigma_{t+1}^2 - \frac{\alpha_{t+1}^2}{\alpha_t^2}\sigma_t^2\right)\mathbf{I}\right)$$

and

$$q(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{x}) = \mathcal{N}\left(\frac{\alpha_{t+1}}{\alpha_t}\frac{\sigma_t^2}{\sigma_{t+1}^2}\mathbf{z}_{t+1} + \left(\sigma_{t+1}^2 - \frac{\alpha_{t+1}^2}{\alpha_t^2}\sigma_t^2\right)\frac{\alpha_t}{\sigma_{t+1}^2}\mathbf{x}, \left(\sigma_{t+1}^2 - \frac{\alpha_{t+1}^2}{\alpha_t^2}\sigma_t^2\right)\frac{\sigma_t^2}{\sigma_{t+1}^2}\mathbf{I}\right)$$

A lot of denoising diffusion probabilistic model (DDPM) papers, including Dall-e start with the forward process:

$$\mathbf{z}_t = \sqrt{\rho_t}\mathbf{x} + \sqrt{1-\rho_t}\epsilon_t$$

and derive

$$q(\mathbf{z}_{t+1}|\mathbf{z}_t) = \mathcal{N}\left(\sqrt{1-\beta_{t+1}}\mathbf{z}_t, \beta_{t+1}\mathbf{I}\right)$$

and

$$q(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{x}) = \mathcal{N}\left(\frac{1-\rho_{t-1}}{1-\rho_t}\sqrt{1-\beta_t}\mathbf{z}_t + \frac{\sqrt{\rho_{t-1}}\beta_t}{1-\rho_t}\mathbf{x}, \frac{\beta_t(1-\rho_{t-1})}{1-\rho_t}\mathbf{I}\right)$$

However, these two derivations are the same, when one uses the following correspondences:

$$\sqrt{\rho_t} = \alpha_t, \sigma_t = \sqrt{1-\rho_t}, \sqrt{1-\beta_t} = \frac{\alpha_t}{\alpha_{t-1}}, \beta_t = \sigma_t^2 - \frac{\alpha_t^2}{\alpha_{t-1}^2}\sigma_{t-1}^2 \tag{27}$$

6 The conditional expectation trick

One of the most widely used tricks in the theory of diffusion and flow matching is the definition of conditional expectation. Specifically, if

$$\mathbf{y} = f(\mathbf{u}) = \mathbb{E}[\mathbf{x}|\mathbf{u}] \tag{28}$$

Then the optimal solution f is the one that minimizes

$$\hat{f} = \arg\min_f \mathbb{E}[\|\mathbf{x} - f(\mathbf{u})\|^2] \tag{29}$$

and vice versa.

For example, at the start, we derived the following loss:

$$\mathcal{L}(\boldsymbol{\theta}; \lambda_t) = \mathbb{E}\left\{ \left\| \mathbf{s}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) + \frac{1}{\sigma_t}\boldsymbol{\epsilon}_t \right\|_2^2 \right\}$$

From our conditional expectation result, we obtain:

$$\mathbf{s}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t) = -\frac{1}{\sigma_t}\mathbb{E}[\boldsymbol{\epsilon}_t|\mathbf{z}_t] \tag{30}$$

This trick will turn out to be super helpful to derive probability flow matching in the next section.

7 Flow matching

7.1 The key insight

Flow matching is based on the following important theoretical result, which we prove in Appendix B. The marginal probability distribution $p_t(\mathbf{z})$ associated with the samples generated according to our forward diffusion process

$$\mathbf{z}_t = \alpha_t \mathbf{x} + \sigma_t \boldsymbol{\epsilon}_t$$

is the same as the distribution of the probability flow ODE with velocity field:

$$\frac{d\mathbf{z}(t)}{dt} = \dot{\mathbf{z}}_t = \mathbf{v}(\mathbf{z}, t) \tag{31}$$

where the vector field $\mathbf{v}(\mathbf{z}, t)$ (*not to be confused by the v-parameterisation in diffusion - sorry!*) is given by the conditional expectation

$$\begin{aligned} \mathbf{v}(\mathbf{z}, t) &= \mathbb{E}[\dot{\mathbf{z}}_t|\mathbf{z}_t = \mathbf{z}] \\ &= \dot{\alpha}_t \mathbb{E}[\mathbf{x}|\mathbf{z}_t = \mathbf{z}] + \dot{\sigma}_t \mathbb{E}[\boldsymbol{\epsilon}|\mathbf{z}_t = \mathbf{z}]. \end{aligned} \tag{32}$$

This velocity field governs the deterministic transport of the probability density, ensuring that the evolution of the unique $p_t(\mathbf{z})$ backward from $t = T$, (which is governed by an ODE with known initial conditions at $t = T$) matches that of

the forward diffusion process. By solving the probability flow ODE backward in time from $\mathbf{z}_T = \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, we can generate samples from $p_0(\mathbf{x})$, which approximates the true data distribution $p(\mathbf{x})$.

In short, the distribution over the images generated by the forward diffuse process is the same as the distribution over the images generated by following the ODE. We simply need to run an ODE numerical technique with a parameterised field $\mathbf{v}_{\boldsymbol{\theta}}(\mathbf{z}, t)$ in order to draw samples. This is how we will do the inference step. That is, instead of deriving Gaussian distributions as done for diffusion, we will simply use an available ODE solver.

The ODE can be discretised with the Euler method, resulting in a simple algorithm over a discrete grid:

$$\mathbf{z}_{i+1} = \mathbf{z}_i + \eta \mathbf{v}_{\boldsymbol{\theta}}(\mathbf{z}_i, t_i) \tag{33}$$

In practice, we simply use one of the many existing ODE solver packages.

7.2 Flow matching training

For training, we use the conditional expectation trick from the previous section. Since $\mathbf{v}(\mathbf{z}, t) = \dot{\alpha}_t \mathbb{E}[\mathbf{x}|\mathbf{z}_t = \mathbf{z}] + \dot{\sigma}_t \mathbb{E}[\epsilon|\mathbf{z}_t = \mathbf{z}]$, we can obtain the neural network $\mathbf{v}_{\boldsymbol{\theta}}(\mathbf{z}, t)$ by minimizing:

$$\mathcal{L}_v(\boldsymbol{\theta}; t) = \mathbb{E}\left\{ \left\| \mathbf{v}_{\boldsymbol{\theta}}(\mathbf{z}_t, t) - \dot{\alpha}_t \mathbf{x} - \dot{\sigma}_t \epsilon_t \right\|_2^2 \right\}$$

Choosing a linear schedule $\alpha_t = 1 - t$ and $\beta_t = t$ between 0 and 1, we have $\dot{\alpha}_t = -1$ and $\dot{\sigma}_t = 1$, so $\dot{\alpha}_t \mathbf{x} + \dot{\sigma}_t \epsilon_t = \epsilon_t - \mathbf{x}$. The loss then becomes the usual expression also found in the rectified flow papers:

$$\mathcal{L}_v(\boldsymbol{\theta}) = \mathbb{E}\left\{ \left\| \mathbf{v}_{\boldsymbol{\theta}}(\mathbf{z}_t, t, \mathbf{c}) - (\epsilon_t - \mathbf{x}) \right\|_2^2 \right\}$$

We could have also used $\alpha_t = \cos(1/2\pi t)$ and $\beta_t = \sin(1/2\pi t)$, but the linear parameterisation results in a very simple objective function.

For the code, I will use the conditional flow matching (CFM) variant which Meta uses. It is the same as what I derived but, annoyingly, the image is defined as

$$\mathbf{x} = \mathbf{z}_1$$

and noise

$$\epsilon = \mathbf{z}_0$$

with $\alpha_t = t$ and $\beta_t = 1 - t$. That is, the direction is inverted. I could have undone this, but it's rather painful so from now on in this section and the flow matching inference section only we reverse the direction of denoising (to backward) and generation (to forward).

CFM uses the intermediate definition:

$$\varphi_t(\mathbf{z}_0, \mathbf{z}_1) = (1 - (1 - \sigma_{min})t)\mathbf{z}_0 + t\mathbf{z}_1 \tag{34}$$

with $\epsilon = \mathbf{z}_0$. Here, σ_{min} is a tiny number. Then the loss becomes

$$\mathcal{L}_v(\boldsymbol{\theta}) = \mathbb{E}_t \mathbb{E}_{\mathbf{x}\sim p_d(\mathbf{x})} \mathbb{E}_{\mathbf{z}_0\sim\mathcal{N}(\mathbf{0},\mathbf{I})} \left\{ \left\| \mathbf{v}_{\boldsymbol{\theta}}(\varphi_t(\mathbf{z}_0, \mathbf{z}_1)) - (\mathbf{z}_1 - (1-\sigma_{min})\mathbf{z}_0) \right\|_2^2 \right\}$$

As a very simple demo, I will use a notebook by Georges Le Bellier, which I find to be very clear. The loss can then be implemented as follows:

```python
class OptimalTransportFlowMatching:
    """Conditional Flow Matching with the linear / OT-style interpolant.

    psi_t(z_0, z_1, t) = (1 - (1 - sigma_min) * t) * z_0 + t * z_1

    so the target velocity along this straight-line path is the constant
    direction `z_1 - (1 - sigma_min) * z_0`.
    """

    def __init__(self, sigma_min: float = 1e-3, time_eps: float = 1e-5) -> None:
        self.sigma_min = sigma_min
        self.time_eps = time_eps

    def interpolant(self, noise: Tensor, data: Tensor, t: Tensor) -> Tensor:
        """Compute psi_t(noise, data) on the straight line between them."""
        return (1.0 - (1.0 - self.sigma_min) * t) * noise + t * data

    def target_velocity(self, noise: Tensor, data: Tensor) -> Tensor:
        """The constant-along-the-path target velocity."""
        return data - (1.0 - self.sigma_min) * noise

    def loss(
        self,
        velocity_network: nn.Module,
        data_batch: Tensor,
    ) -> Tensor:
        """Conditional flow matching loss.

        Uses low-discrepancy time sampling within the batch (one uniform t,
        offset by k/B) to reduce variance, as in the original FM paper.
        """
        batch_size = data_batch.shape[0]
        device = data_batch.device

        # Stratified t in [0, 1 - eps].
        base_t = torch.rand(1, device=device)
```

```
        offsets = torch.arange(batch_size, device=device) / batch_size
        t = (base_t + offsets) % (1.0 - self.time_eps)
        # Broadcast t over the feature dims.
        feature_shape = (-1,) + (1,) * (data_batch.ndim - 1)
        t = t.view(feature_shape)

        noise = torch.randn_like(data_batch)
        interpolated = self.interpolant(noise, data_batch, t)
        target = self.target_velocity(noise, data_batch)

        predicted = velocity_network(interpolated, t.view(batch_size),
        ↪  condition=None)
        return F.mse_loss(predicted, target)
```

The vector field is implemented in the following class. It defines encode and decode functions:

- `encode` solves the ODE from $t = 1$ to $t = 0$, mapping the data distribution $p_d(\mathbf{z}_1)$ to the Gaussian noise distribution $p(\mathbf{z}_0)$.
- `decode` maps the Gaussian distribution $p(\mathbf{z}_0)$ to the data distribution $p_d(\mathbf{z}_1)$ from $t = 0$ to $t = 1$.

For the integrator we use a small in-house RK4 stepper to keep the dependencies minimal. In practice you would reach for *odeint* from torchdiffeq or Zuko; both are drop-in replacements for our `integrate` method below.

```
def runge_kutta_4_step(
    velocity_fn: Callable[[Tensor, Tensor], Tensor],
    state: Tensor,
    t: Tensor,
    dt: Tensor,
) -> Tensor:
    """One classic RK4 step for state' = velocity_fn(state, t)."""
    k1 = velocity_fn(state, t)
    k2 = velocity_fn(state + 0.5 * dt * k1, t + 0.5 * dt)
    k3 = velocity_fn(state + 0.5 * dt * k2, t + 0.5 * dt)
    k4 = velocity_fn(state + dt * k3, t + dt)
    return state + (dt / 6.0) * (k1 + 2 * k2 + 2 * k3 + k4)


class ConditionalVelocityField(nn.Module):
    """Wraps a velocity-prediction network as an ODE field, and
    ↪  provides
    convenient encode / decode methods.
    """

    def __init__(self, network: nn.Module) -> None:
        super().__init__()
```

```
        self.network = network

    def forward(
        self,
        state: Tensor,
        t: Tensor,
        condition: Tensor | None = None,
    ) -> Tensor:
        return self.network(state, t, condition)

    @torch.no_grad()
    def integrate(
        self,
        initial_state: Tensor,
        t_start: float,
        t_end: float,
        num_steps: int = 100,
    ) -> Tensor:
        """Integrate the ODE from t_start to t_end using RK4."""
        t = torch.tensor(t_start, device=initial_state.device,
        ↪  dtype=initial_state.dtype)
        dt = torch.tensor(
            (t_end - t_start) / num_steps,
            device=initial_state.device,
            dtype=initial_state.dtype,
        )

        def velocity_at(state: Tensor, t_scalar: Tensor) -> Tensor:
            return self(state, t_scalar.expand(state.shape[0]))

        state = initial_state
        for _ in range(num_steps):
            state = runge_kutta_4_step(velocity_at, state, t, dt)
            t = t + dt
        return state

    @torch.no_grad()
    def decode(self, noise: Tensor, num_steps: int = 100) -> Tensor:
        """Map noise (t=0) to data (t=1)."""
        return self.integrate(noise, t_start=0.0, t_end=1.0,
        ↪  num_steps=num_steps)

    @torch.no_grad()
    def encode(self, data: Tensor, num_steps: int = 100) -> Tensor:
        """Map data (t=1) back to noise (t=0)."""
        return self.integrate(data, t_start=1.0, t_end=0.0,
        ↪  num_steps=num_steps)
```

With these two classes, and the standard data loaders and model (say, trans-

former) classes, the training proceeds as follows:

```python
def train_flow_matching_model(
    velocity_field: ConditionalVelocityField,
    flow_matcher: OptimalTransportFlowMatching,
    dataloader,
    *,
    num_epochs: int = 5_000,
    learning_rate: float = 1e-3,
) -> list[float]:
    """Train `velocity_field` to minimise the CFM loss."""
    optimizer = torch.optim.Adam(velocity_field.parameters(),
    ↪  lr=learning_rate)
    loss_history: list[float] = []

    velocity_field.train()
    for _ in range(num_epochs):
        for batch in dataloader:
            data_batch = batch[0]
            loss = flow_matcher.loss(velocity_field, data_batch)
            optimizer.zero_grad(set_to_none=True)
            loss.backward()
            optimizer.step()
            loss_history.append(loss.item())
    return loss_history
```

7.3 Flow matching inference

Using the *ConditionalVelocityField* class with the RK4 integrator, inference is straightforward and much simpler than diffusion. To draw a batch of samples, we just decode pure-noise initial conditions through the learned flow:

```python
velocity_field.eval()
initial_noise = torch.randn(num_samples, data_dim, device=DEVICE)
generated_samples = velocity_field.decode(initial_noise, num_steps=100)
```

7.4 Relation between the score and velocity field

Using the score trick and the definition of the vector field, it can be shown easily that the score is related to the vector field by the following relation:

$$\mathbf{s}_{\boldsymbol{\theta}}(\mathbf{z}, t) = \frac{1}{\sigma_t} \frac{\alpha_t \mathbf{v}_{\boldsymbol{\theta}}(\mathbf{z}, t) - \dot{\alpha}_t \mathbf{z}}{\dot{\alpha}_t \sigma_t - \alpha_t \dot{\sigma}_t} \tag{35}$$

Thus, one can use either the diffusion loss to estimate the score or the flow matching loss to estimate the vector field, apply this relation, and then use either a numerical integrator or the inference equations we derived to generate data. That is, we can mix and match.

The only remaining thing is implement a solid transformer, do all the scaling analysis carefully, and above all focus on the datasets.

Conditioning on different signals is also an interesting topic, which we haven't covered here.

8 Training LLMs with diffusion losses jointly over text and images

There are two natural ways to combine an autoregressive LLM with a diffusion loss for the continuous (image, audio, video) parts of a sequence. They differ in *where the noise enters the network*. We describe each in turn and then contrast them.

8.1 Approach 1: Noise into a separate diffusion head

Let τ denote the time index for an autoregressive model over N image patches:

$$p(\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N) = \prod_{\tau=1}^{N} p(\mathbf{x}^\tau | \mathbf{x}^1, \ldots, \mathbf{x}^{\tau-1}) \tag{36}$$

We can use a transformer LLM to predict an embedding $\mathbf{h}^\tau$ autoregressively as follows:

$$\mathbf{h}^\tau = f_{\boldsymbol{\theta}}(\mathbf{x}^0, \mathbf{x}^1, \ldots, \mathbf{x}^{\tau-1}) \tag{37}$$

and then generate the next patch via:

$$\mathbf{x}^\tau \sim p(\mathbf{x}^\tau | \mathbf{h}^\tau) \tag{38}$$

We can model this conditional distribution over patches with diffusion, instead of the standard softmax used for text and hard quantized embeddings. To achieve this, we proceed as before. Let the forward diffusion be:

$$\mathbf{z}_t^\tau = \alpha_t \mathbf{x}^\tau + \sigma_t \boldsymbol{\epsilon}_t \tag{39}$$

Note again that t is the diffusion time index, whereas τ is the index over image patches (autoregressive time index). The loss can be any of the losses we've covered thus far, for example:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}\left[\|\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\mathbf{z}_t^\tau, \lambda_t, \mathbf{h}^\tau) - \boldsymbol{\epsilon}_t\|_2^2\right] \tag{40}$$

The crucial detail: the noisy latent $\mathbf{z}_t^\tau$ is fed to a *separate* small denoiser head $\boldsymbol{\epsilon}_{\boldsymbol{\theta}}$ (often an MLP or a tiny U-Net), with the LLM hidden state $\mathbf{h}^\tau$ acting only as conditioning. The LLM itself never sees the noise. This approach is described in detail in autoregressive image generation without vector quantization (MAR). See the figure below.

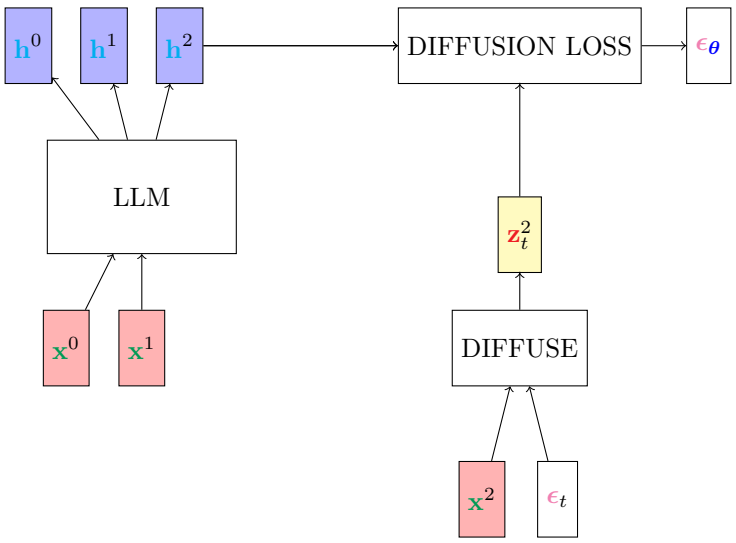

Figure 4: MAR-style architecture. The LLM consumes clean past patches $\mathbf{x}^0, \mathbf{x}^1$ and emits a hidden state $\mathbf{h}^2$. The forward diffusion (DIFFUSE) corrupts the target patch $\mathbf{x}^2$ into $\mathbf{z}_t^2$ using a fresh noise sample ϵ_t. A separate diffusion head $\epsilon_{\boldsymbol{\theta}}$ takes $\mathbf{z}_t^2$ and the LLM's hidden state $\mathbf{h}^2$ as inputs. Crucially, the LLM never sees the noise: the noisy input only enters the head.

8.2 Approach 2: noise into the LLM (Transfusion)

Transfusion (Zhou et al., 2024) makes the opposite design choice from MAR. Instead of keeping the noisy image latent outside the LLM and sending it to a small auxiliary denoising head, Transfusion puts the noisy image patches directly into the transformer's input sequence. The same transformer therefore plays two roles at once: at text positions it is a normal next-token predictor, and at image positions it is a diffusion denoiser over continuous vectors.

A good way to understand the method is to separate *representation*, *attention*, and *loss*.

1. Representation: one sequence, two kinds of elements. Text is represented as discrete tokens exactly as in a language model. An image is first encoded into a continuous latent space, usually with a VAE, and then split into N patch vectors

$$\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N.$$

The image patch vectors are not quantised into a codebook. They remain real-valued vectors. To splice an image into a text sequence, Transfusion surrounds

the image block with special marker tokens:

$$\texttt{A cat <BOI>}\ \mathbf{x}^1, \ldots, \mathbf{x}^N\ \texttt{<EOI> is cute.}$$

The marker tokens tell the model when it should switch from language modelling to image diffusion and back again.

2. Noising: corrupt the image block, not the text. During training, choose a diffusion time t for each image and corrupt every patch in that image at the same noise level:

$$\mathbf{z}_t^\tau = \alpha_t \mathbf{x}^\tau + \sigma_t \epsilon_t^\tau, \qquad \epsilon_t^\tau \sim \mathcal{N}(0, \mathbf{I}), \qquad \tau = 1, \ldots, N. \tag{41}$$

The transformer input is therefore a mixed sequence: clean text token embeddings, marker-token embeddings, and noisy continuous image-patch embeddings. A time embedding $\phi(t)$ (or equivalently a log-SNR embedding $\phi(\lambda_t)$) is added to the image-patch inputs so the transformer knows how much noise it must remove.

3. Attention: causal outside the image, bidirectional inside it. Text remains autoregressive: a text token can only attend to previous elements. Image patches, however, have no natural left-to-right generation order, so patches within the same image block are allowed to attend to one another bidirectionally. Across image boundaries the sequence remains causal: an image can attend to the prefix before it, and text after an image can attend to the image, but an image patch cannot attend to future text.

4. Loss: cross-entropy for text, diffusion MSE for images. The transformer output at a text position goes through the usual vocabulary head and is trained with cross-entropy. The transformer output at an image-patch position goes through a continuous denoising head and is trained to predict the noise:

$$\mathcal{L}_{\text{diff}}(\boldsymbol{\theta}) = \mathbb{E}\left[\sum_{\tau=1}^{N} \| \epsilon_{\boldsymbol{\theta}}^\tau(\mathbf{z}_t^1, \ldots, \mathbf{z}_t^N, t, \text{prefix}) - \epsilon_t^\tau \|_2^2\right], \tag{42}$$

$$\mathcal{L}_{\text{Transfusion}}(\boldsymbol{\theta}) = \mathcal{L}_{\text{LM}}(\boldsymbol{\theta}) + \lambda_{\text{img}} \mathcal{L}_{\text{diff}}(\boldsymbol{\theta}). \tag{43}$$

The important point is that the *same transformer activations* support both losses. Only the small input/output adapters are modality-specific: text uses token embeddings and a softmax head; images use patch embeddings, time embeddings, and a continuous denoising output.

Inference. Decoding becomes a simple state machine. In *LM mode*, the model samples tokens one at a time from the softmax. When it samples `<BOI>`, it switches to *diffusion mode*: it appends N pure-noise image patches $\mathbf{z}_T^1, \ldots, \mathbf{z}_T^N$ to the current sequence and runs the diffusion sampler. At each denoising step, the full transformer is called on the current mixed sequence, its image-position

outputs predict $\hat{\epsilon}_t^1, \ldots, \hat{\epsilon}_t^N$, and the sampler overwrites the image block with the next latent $\mathbf{z}_{t-1}^1, \ldots, \mathbf{z}_{t-1}^N$. After the final step, the model appends `<EOI>` and switches back to LM mode.

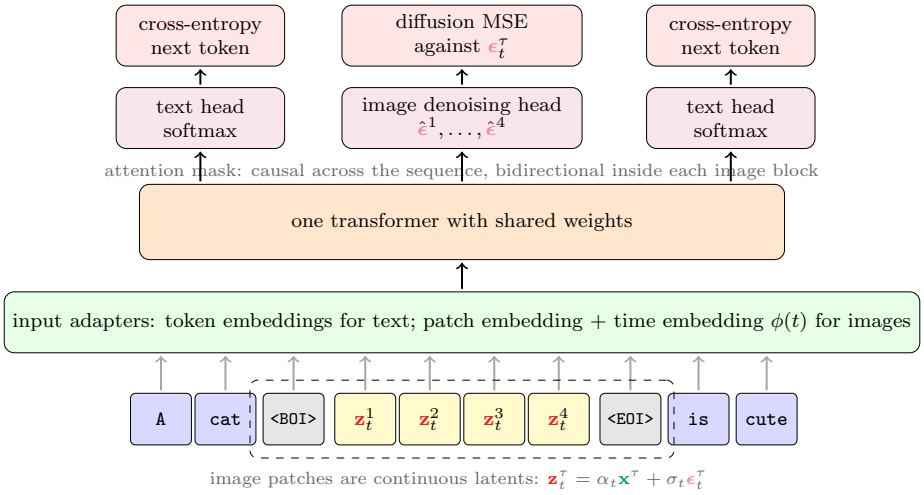

Figure 5: Transfusion-style training picture. Text tokens (blue), marker tokens (gray), and noisy continuous image patches (yellow) are placed into one sequence. A small set of modality-specific adapters maps them into the shared transformer space. The same transformer then feeds two kinds of output heads: a vocabulary head trained with cross-entropy at text positions, and a denoising head trained with diffusion MSE at image positions. The image block receives time conditioning and uses bidirectional attention within the block, while the mixed text-image sequence remains causal across blocks.

8.3 Contrasting the two approaches

Both approaches train one big transformer plus a diffusion-style MSE-on-noise loss for image patches, and both can be combined with cross-entropy on text. The differences are:

- **Where the noise lives.** In MAR, the noise is contained inside a small auxiliary head and the LLM only sees clean tokens / clean past patches. In Transfusion, the noise is mixed directly into the LLM's input sequence; the LLM itself learns to denoise.
- **Number of forward passes per image.** MAR needs one LLM forward pass to compute $\mathbf{h}^\tau$ plus, for each diffusion step, a forward pass through the diffusion head; the LLM activations can be cached. Transfusion runs the full transformer at every diffusion step, since the noisy patches change between steps. This makes Transfusion sampling more expensive per step

but means the same transformer learns multi-modal representations end-to-end.

- **Attention pattern.** MAR uses standard causal attention end-to-end. Transfusion uses causal between-element attention with a *bidirectional* window inside each image, exploiting the fact that image patches have no natural left-to-right order.
- **Token-versus-patch parallelism.** MAR is autoregressive over patches: patch τ is generated only after patches $1, \dots, \tau - 1$. Transfusion generates all N patches of an image *in parallel* via diffusion, which can be substantially faster for high-resolution images.
- **Information bottleneck.** In MAR, everything the diffusion head knows about the past must fit through the single conditioning vector $\mathbf{h}^\tau$. In Transfusion, every image-patch position attends back to the entire prefix, so the conditioning is much richer.

The Transfusion paper compares this recipe head-to-head with the alternative of *quantising* images into discrete tokens and treating them like text (the Chameleon recipe). Keeping image patches continuous and using diffusion is reported to scale better and reach matching FID at less than a third of the compute. The same trade-offs we have studied throughout this tutorial therefore carry over: cross-entropy on quantised tokens is simple but throws information away, while a noise-prediction loss on continuous patches preserves it at the cost of a more elaborate sampling procedure.

A natural variant, also discussed by the Transfusion authors as future work, is to swap the diffusion loss for the flow-matching loss of the previous section, or to replace it with the $\mathbf{v}$- or $\mathbf{x}$-parameterised losses earlier in this tutorial. Everything we have said about schedules, parameterisations, and inference samplers applies unchanged: only the input wiring of the LLM differs.

Appendix A: Derivation of the backward sampling Gaussian

We need to derive the expression for the mean and variance of $q(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{x})$. The first step is to apply Bayes' rule:

$$\begin{aligned} q(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{x}) &= \frac{q(\mathbf{z}_{t+1}|\mathbf{z}_t, \mathbf{x})\, q(\mathbf{z}_t|\mathbf{x})}{q(\mathbf{z}_{t+1}|\mathbf{x})} \\ &\propto q(\mathbf{z}_{t+1}|\mathbf{z}_t)\, q(\mathbf{z}_t|\mathbf{x}). \end{aligned} \tag{44}$$

The prior follows from the way we diffuse:

$$q(\mathbf{z}_t|\mathbf{x}) = \mathcal{N}(\mathbf{z}_t; \alpha_t \mathbf{x}, \sigma_t^2 \mathbf{I}). \tag{45}$$

We also know that the one-step forward transition can be written as

$$q(\mathbf{z}_{t+1}|\mathbf{z}_t) = \mathcal{N}\left(\mathbf{z}_{t+1}; \frac{\alpha_{t+1}}{\alpha_t}\mathbf{z}_t,\, \left(\sigma_{t+1}^2 - \frac{\alpha_{t+1}^2}{\alpha_t^2}\sigma_t^2\right)\mathbf{I}\right). \tag{46}$$

Multiplying two Gaussians gives another Gaussian. Define

$$A = \frac{\alpha_{t+1}}{\alpha_t}, \qquad B = \sigma_{t+1}^2 - A^2\sigma_t^2. \tag{47}$$

Here $B > 0$ is the variance of the forward step from t to $t+1$, assuming the log signal-to-noise ratio decreases as we add noise. Up to constants that do not depend on $\mathbf{z}_t$, the posterior is

$$q(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{x}) \propto \exp\left\{-\frac{1}{2}\left[\frac{1}{B}\|\mathbf{z}_{t+1} - A\mathbf{z}_t\|_2^2 + \frac{1}{\sigma_t^2}\|\mathbf{z}_t - \alpha_t\mathbf{x}\|_2^2\right]\right\}. \tag{48}$$

Now complete the square in $\mathbf{z}_t$. The quadratic terms give the precision

$$\begin{aligned} \Sigma^{-1} &= \left(\frac{A^2}{B} + \frac{1}{\sigma_t^2}\right)\mathbf{I} \\ &= \frac{A^2\sigma_t^2 + B}{B\sigma_t^2}\mathbf{I} = \frac{\sigma_{t+1}^2}{B\sigma_t^2}\mathbf{I}, \end{aligned} \tag{49}$$

where the last equality uses $A^2\sigma_t^2 + B = \sigma_{t+1}^2$. Inverting gives

$$\begin{aligned} \Sigma &= \frac{B\sigma_t^2}{\sigma_{t+1}^2}\mathbf{I} \\ &= \left(\sigma_{t+1}^2 - \frac{\alpha_{t+1}^2}{\alpha_t^2}\sigma_t^2\right)\frac{\sigma_t^2}{\sigma_{t+1}^2}\mathbf{I}. \end{aligned} \tag{50}$$

The linear terms give the natural-parameter vector

$$\frac{A}{B}\mathbf{z}_{t+1} + \frac{\alpha_t}{\sigma_t^2}\mathbf{x}.$$

Therefore the mean is

$$\begin{aligned}\boldsymbol{\mu} &= \Sigma\left(\frac{A}{B}\mathbf{z}_{t+1} + \frac{\alpha_t}{\sigma_t^2}\mathbf{x}\right) \\ &= \frac{A\sigma_t^2}{\sigma_{t+1}^2}\mathbf{z}_{t+1} + \frac{B\alpha_t}{\sigma_{t+1}^2}\mathbf{x}. \end{aligned} \tag{51}$$

Putting the mean and covariance together, we can write the posterior in a form that stays within the page margins:

$$q(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{x}) = \mathcal{N}\left(\mathbf{z}_t; \boldsymbol{\mu}_{t|t+1}, \Sigma_{t|t+1}\right), \tag{52}$$

$$\boldsymbol{\mu}_{t|t+1} = \frac{\alpha_{t+1}}{\alpha_t}\frac{\sigma_t^2}{\sigma_{t+1}^2}\mathbf{z}_{t+1} + \left(\sigma_{t+1}^2 - \frac{\alpha_{t+1}^2}{\alpha_t^2}\sigma_t^2\right)\frac{\alpha_t}{\sigma_{t+1}^2}\mathbf{x}, \tag{53}$$

$$\Sigma_{t|t+1} = \left(\sigma_{t+1}^2 - \frac{\alpha_{t+1}^2}{\alpha_t^2}\sigma_t^2\right)\frac{\sigma_t^2}{\sigma_{t+1}^2}\mathbf{I}. \tag{54}$$

It is often more convenient to express the same result in terms of the log signal-to-noise ratio

$$\lambda_t = \log\frac{\alpha_t^2}{\sigma_t^2}.$$

Using

$$\frac{\alpha_{t+1}^2}{\sigma_{t+1}^2}\frac{\sigma_t^2}{\alpha_t^2} = e^{\lambda_{t+1}-\lambda_t},$$

we obtain

$$\boldsymbol{\mu}_{t|t+1} = \frac{\alpha_t}{\alpha_{t+1}}e^{\lambda_{t+1}-\lambda_t}\mathbf{z}_{t+1} + \left(1 - e^{\lambda_{t+1}-\lambda_t}\right)\alpha_t\mathbf{x}, \tag{55}$$

$$\Sigma_{t|t+1} = \left(1 - e^{\lambda_{t+1}-\lambda_t}\right)\sigma_t^2\mathbf{I}. \tag{56}$$

Finally, setting

$$c = 1 - e^{\lambda_{t+1}-\lambda_t} \in (0, 1),$$

gives the compact sampler form:

$$q(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{x}) = \mathcal{N}\left(\mathbf{z}_t;\, \alpha_t\left[\frac{1-c}{\alpha_{t+1}}\mathbf{z}_{t+1} + c\mathbf{x}\right], c\,\sigma_t^2\mathbf{I}\right). \tag{57}$$

This is the form used by the sampler.

Appendix B

The marginal probability distribution $p_t(\mathbf{z})$ associated with the samples generated according to our forward diffusion process

$$\mathbf{z}_t = \alpha_t \mathbf{x} + \sigma_t \boldsymbol{\epsilon}_t$$

coincides with the distribution of the probability flow ODE with velocity field:

$$\dot{\mathbf{z}}_t = \mathbf{v}(\mathbf{z}, t) \tag{58}$$

where

$$\mathbf{v}(\mathbf{z}, t) = \mathbb{E}[\dot{\mathbf{z}}_t | \mathbf{z}_t = \mathbf{z}] = \dot{\alpha}_t \mathbb{E}[\mathbf{x} | \mathbf{z}_t = \mathbf{z}] + \dot{\sigma}_t \mathbb{E}[\boldsymbol{\epsilon} | \mathbf{z}_t = \mathbf{z}]. \tag{59}$$

The proof consists of two steps.

- STEP 1: Prove that $p_t(\mathbf{z})$ satisfies the transport equation

$$\partial_t p_t(\mathbf{z}) + \nabla_{\mathbf{z}} \cdot (\mathbf{v}(\mathbf{z}, t) p_t(\mathbf{z})) = 0. \tag{60}$$

- STEP 2: Solve the transport equation by the method of characteristics to prove that $p_t(\mathbf{z})$ associated with the samples generated according to our forward diffusion process

$$\mathbf{z}_t = \alpha_t \mathbf{x} + \sigma_t \boldsymbol{\epsilon}_t$$

coincides with the distribution of the probability flow ODE with velocity field:

$$\dot{\mathbf{z}}_t = \mathbf{v}(\mathbf{z}, t). \tag{61}$$

I will use the SiT paper to prove STEP 1 (their Appendix A.1). I will then use ChatGPT with the following prompt to prove STEP 2.

Prompt: *Using the above statement, please prove STEP 2 in detail. Assume that I only have basic knowledge of calculus so explain all steps clearly and in detail.*

STEP 1: Derive the transport equation

This part of the proof is from the Appendix of the SiT paper. Consider the time-dependent probability density function (PDF) $p_t(\mathbf{z})$ of $\mathbf{z}_t = \alpha_t \mathbf{x} + \sigma_t \boldsymbol{\epsilon}$. By definition, its characteristic function $\hat{p}_t(\mathbf{k})$ is given by

$$\hat{p}_t(\mathbf{k}) = \mathbb{E}[e^{i\mathbf{k} \cdot \mathbf{z}_t}] \tag{62}$$

where $\mathbb{E}$ denotes expectation over $\mathbf{x}$ and ϵ. Taking time derivative on both sides, and using the tower property of conditional expectation, we have

$$\partial_t \hat{p}_t(\mathbf{k}) = i\mathbf{k} \cdot \mathbb{E}[\dot{\mathbf{z}}_t e^{i\mathbf{k}\cdot\mathbf{z}_t}] \tag{63}$$

$$= i\mathbf{k} \cdot \mathbb{E}_{\mathbf{z}\sim p_t}[\mathbb{E}[\dot{\mathbf{z}}_t e^{i\mathbf{k}\cdot\mathbf{z}_t}|\mathbf{z}_t = \mathbf{z}]] \tag{64}$$

$$= i\mathbf{k} \cdot \mathbb{E}_{\mathbf{z}\sim p_t}[\mathbb{E}[(\dot{\alpha}_t\mathbf{z}_* + \dot{\sigma}_t\epsilon)e^{i\mathbf{k}\cdot\mathbf{z}_t}|\mathbf{z}_t = \mathbf{z}]] \tag{65}$$

$$= i\mathbf{k} \cdot \mathbb{E}_{\mathbf{z}\sim p_t}[\mathbb{E}[(\dot{\alpha}_t\mathbf{z}_* + \dot{\sigma}_t\epsilon)|\mathbf{z}_t = \mathbf{z}]e^{i\mathbf{k}\cdot\mathbf{z}}] \tag{66}$$

$$= i\mathbf{k} \cdot \mathbb{E}_{\mathbf{z}\sim p_t}[\mathbf{v}(\mathbf{z}, t)e^{i\mathbf{k}\cdot\mathbf{z}}] \tag{67}$$

where $\mathbf{v}(\mathbf{z}, t) = \mathbb{E}[(\dot{\alpha}_t\mathbf{z}_* + \dot{\sigma}_t\epsilon)|\mathbf{z}_t = \mathbf{z}] = \dot{\alpha}_t\mathbb{E}[\mathbf{z}_*|\mathbf{z}_t = \mathbf{z}] + \dot{\sigma}_t\mathbb{E}[\epsilon|\mathbf{z}_t = \mathbf{z}]$ is the velocity defined in Eq. (3). Explicitly, Eq. (67) reads

$$\partial_t \int_{\mathbb{R}^d} e^{i\mathbf{k}\cdot\mathbf{z}} p_t(\mathbf{z})d\mathbf{z} = i\mathbf{k} \cdot \int_{\mathbb{R}^d} \mathbf{v}(\mathbf{z}, t)e^{i\mathbf{k}\cdot\mathbf{z}} p_t(\mathbf{z})d\mathbf{z} \tag{68}$$

from which we deduce

$$\int_{\mathbb{R}^d} e^{i\mathbf{k}\cdot\mathbf{z}} \partial_t p_t(\mathbf{z})d\mathbf{z} = \int_{\mathbb{R}^d} \mathbf{v}(\mathbf{z}, t) \cdot \nabla_{\mathbf{z}}[e^{i\mathbf{k}\cdot\mathbf{z}}] p_t(\mathbf{z})d\mathbf{z} \tag{69}$$

$$= -\int_{\mathbb{R}^d} \nabla_{\mathbf{z}} \cdot [\mathbf{v}(\mathbf{z}, t)p_t(\mathbf{z})]e^{i\mathbf{k}\cdot\mathbf{z}} d\mathbf{z} \tag{70}$$

where $\nabla_{\mathbf{z}} \cdot [\mathbf{v}p_t] = \sum_{i=1}^d \frac{\partial}{\partial z_i}[v_i p_t]$ is the divergence operator and we used integration by parts to get the second equality. By the properties of the Fourier transform, uniqueness for all frequencies, the last equation implies that $p_t(\mathbf{z})$ satisfies the transport equation (a partial differential equation for p in terms of t and $\mathbf{z}$):

$$\partial_t p_t(\mathbf{z}) + \nabla_{\mathbf{z}} \cdot (\mathbf{v}(\mathbf{z}, t)p_t(\mathbf{z})) = 0. \tag{71}$$

Solving this equation by the method of characteristics leads to probability flow ODE, as shown by ChatGPT in the following section.

STEP 2: Solution by method of characteristics

We wish to prove that the marginal probability density

$$p_t(\mathbf{z})$$

of the forward diffusion process

$$\mathbf{z}_t = \alpha_t\mathbf{x} + \sigma_t\epsilon_t,$$

which satisfies the transport equation

$$\partial_t p_t(\mathbf{z}) + \nabla_{\mathbf{z}} \cdot \Big(\mathbf{v}(\mathbf{z}, t)\, p_t(\mathbf{z})\Big) = 0,$$

with the velocity field

$$\mathbf{v}(\mathbf{z}, t) = \dot{\alpha}_t\, \mathbb{E}[\mathbf{x} \mid \mathbf{z}_t = \mathbf{z}] + \dot{\sigma}_t\, \mathbb{E}[\epsilon \mid \mathbf{z}_t = \mathbf{z}],$$

is equivalent to the density obtained by solving the probability flow ODE

$$\dot{\mathbf{z}}_t = \mathbf{v}(\mathbf{z}, t).$$

Part A: Expanding the Transport Equation

First, we rewrite the transport equation:

$$\partial_t p_t(\mathbf{z}) + \nabla_{\mathbf{z}} \cdot \Big(\mathbf{v}(\mathbf{z}, t)\, p_t(\mathbf{z})\Big) = 0.$$

Expanding the divergence term by the product rule gives

$$\nabla_{\mathbf{z}} \cdot \Big(\mathbf{v}(\mathbf{z}, t)\, p_t(\mathbf{z})\Big) = \mathbf{v}(\mathbf{z}, t) \cdot \nabla_{\mathbf{z}} p_t(\mathbf{z}) + p_t(\mathbf{z})\, \nabla_{\mathbf{z}} \cdot \mathbf{v}(\mathbf{z}, t).$$

Thus, the transport equation becomes

$$\partial_t p_t(\mathbf{z}) + \mathbf{v}(\mathbf{z}, t) \cdot \nabla_{\mathbf{z}} p_t(\mathbf{z}) + p_t(\mathbf{z})\, \nabla_{\mathbf{z}} \cdot \mathbf{v}(\mathbf{z}, t) = 0.$$

Part B: Introducing the Characteristic Curves

The method of characteristics reduces the PDE to an ODE along paths (or *characteristics*) defined by

$$\frac{d\mathbf{z}(t)}{dt} = \mathbf{v}\big(\mathbf{z}(t), t\big).$$

For any starting point $\mathbf{z}_{t_0}$ at time t_0 (for example, at $t = T$ where the density is known), we denote the trajectory by

$$\mathbf{z}(t) = \phi_{t,t_0}(\mathbf{z}_{t_0}).$$

Part C: Computing the Total Derivative Along a Characteristic

Consider $p_t\big(\mathbf{z}(t)\big)$, which is the density along the trajectory. By the chain rule,

$$\frac{d}{dt} p_t\big(\mathbf{z}(t)\big) = \partial_t p_t\big(\mathbf{z}(t)\big) + \frac{d\mathbf{z}(t)}{dt} \cdot \nabla_{\mathbf{z}} p_t\big(\mathbf{z}(t)\big).$$

Since $\frac{d\mathbf{z}(t)}{dt} = \mathbf{v}\big(\mathbf{z}(t), t\big)$, we obtain

$$\frac{d}{dt} p_t\big(\mathbf{z}(t)\big) = \partial_t p_t\big(\mathbf{z}(t)\big) + \mathbf{v}\big(\mathbf{z}(t), t\big) \cdot \nabla_{\mathbf{z}} p_t\big(\mathbf{z}(t)\big).$$

Comparing this with our expanded transport equation at the point $\mathbf{z}(t)$,

$$\partial_t p_t\big(\mathbf{z}(t)\big) + \mathbf{v}\big(\mathbf{z}(t), t\big) \cdot \nabla_{\mathbf{z}} p_t\big(\mathbf{z}(t)\big) = -p_t\big(\mathbf{z}(t)\big)\, \nabla_{\mathbf{z}} \cdot \mathbf{v}\big(\mathbf{z}(t), t\big),$$

we find that along a characteristic,

$$\frac{d}{dt} p_t\big(\mathbf{z}(t)\big) = -p_t\big(\mathbf{z}(t)\big)\, \nabla_{\mathbf{z}} \cdot \mathbf{v}\big(\mathbf{z}(t), t\big).$$

Part D: Solving the ODE for the Density

The ODE

$$\frac{d}{dt}p_t\big(\mathbf{z}(t)\big) = -p_t\big(\mathbf{z}(t)\big)\,\nabla_{\mathbf{z}}\cdot\mathbf{v}\big(\mathbf{z}(t),t\big)$$

is separable. We rewrite it as

$$\frac{1}{p_t\big(\mathbf{z}(t)\big)}\frac{d}{dt}p_t\big(\mathbf{z}(t)\big) = -\nabla_{\mathbf{z}}\cdot\mathbf{v}\big(\mathbf{z}(t),t\big).$$

Integrate both sides from an initial time t_0 (e.g., T) to time t:

$$\int_{t_0}^{t}\frac{1}{p_s\big(\mathbf{z}(s)\big)}\frac{d}{ds}p_s\big(\mathbf{z}(s)\big)\,ds = -\int_{t_0}^{t}\nabla_{\mathbf{z}}\cdot\mathbf{v}\big(\mathbf{z}(s),s\big)\,ds.$$

The left-hand side integrates to

$$\ln p_t\big(\mathbf{z}(t)\big) - \ln p_{t_0}\big(\mathbf{z}(t_0)\big) = \ln\left(\frac{p_t\big(\mathbf{z}(t)\big)}{p_{t_0}\big(\mathbf{z}(t_0)\big)}\right).$$

Thus, we have

$$\ln\left(\frac{p_t\big(\mathbf{z}(t)\big)}{p_{t_0}\big(\mathbf{z}(t_0)\big)}\right) = -\int_{t_0}^{t}\nabla_{\mathbf{z}}\cdot\mathbf{v}\big(\mathbf{z}(s),s\big)\,ds,$$

and exponentiating both sides yields

$$p_t\big(\mathbf{z}(t)\big) = p_{t_0}\big(\mathbf{z}(t_0)\big)\exp\left(-\int_{t_0}^{t}\nabla_{\mathbf{z}}\cdot\mathbf{v}\big(\mathbf{z}(s),s\big)\,ds\right).$$

Part E: Interpreting the Result as a Change of Variables

The above expression shows how the density changes along the flow. In fact, if we consider the change of variables induced by the mapping

$$\mathbf{z}(t) = \phi_{t,t_0}\big(\mathbf{z}(t_0)\big),$$

the density transforms as

$$p_t(\mathbf{z}) = p_{t_0}(\mathbf{z}_0)\left|\det\frac{\partial\mathbf{z}(t)}{\partial\mathbf{z}_0}\right|^{-1}.$$

It turns out that the derivative of the log-determinant of the Jacobian matrix satisfies

$$\frac{d}{dt}\ln\left|\det\frac{\partial\mathbf{z}(t)}{\partial\mathbf{z}_0}\right| = \nabla_{\mathbf{z}}\cdot\mathbf{v}\big(\mathbf{z}(t),t\big).$$

Thus, the exponential factor in our solution exactly accounts for the change in volume (i.e., the local expansion or contraction) induced by the flow.

Part F: Concluding the Proof

Since the forward diffusion process is defined by

$$\mathbf{z}_t = \alpha_t \mathbf{x} + \sigma_t \epsilon_t,$$

its marginal density is given by

$$p_t(\mathbf{z}) = \int p(\mathbf{x})\, \mathcal{N}\Big(\mathbf{z}; \alpha_t \mathbf{x},\, \sigma_t^2 \mathbf{I}\Big)\, d\mathbf{x}.$$

At the final time T, we typically have

$$\mathbf{z}_T = \alpha_T \mathbf{x} + \sigma_T \epsilon_T,$$

with ϵ_T chosen so that $p_T(\mathbf{z})$ is a simple distribution (often a standard Gaussian).

By solving the probability flow ODE backward in time from $\mathbf{z}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, the density evolves as

$$p_t\big(\mathbf{z}(t)\big) = p_T\big(\mathbf{z}(T)\big) \exp\Big(-\int_T^t \nabla_{\mathbf{z}} \cdot \mathbf{v}\big(\mathbf{z}(s), s\big)\, ds\Big).$$

Since this evolution is unique given the initial condition at T, the density obtained by *pushing forward* the Gaussian through the probability flow ODE exactly matches the marginal density obtained from the forward diffusion process. This completes the detailed proof of STEP 2.