Diffusion and flow matching tutorial

Nando de Freitas

November 27, 2026

1 Introduction

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

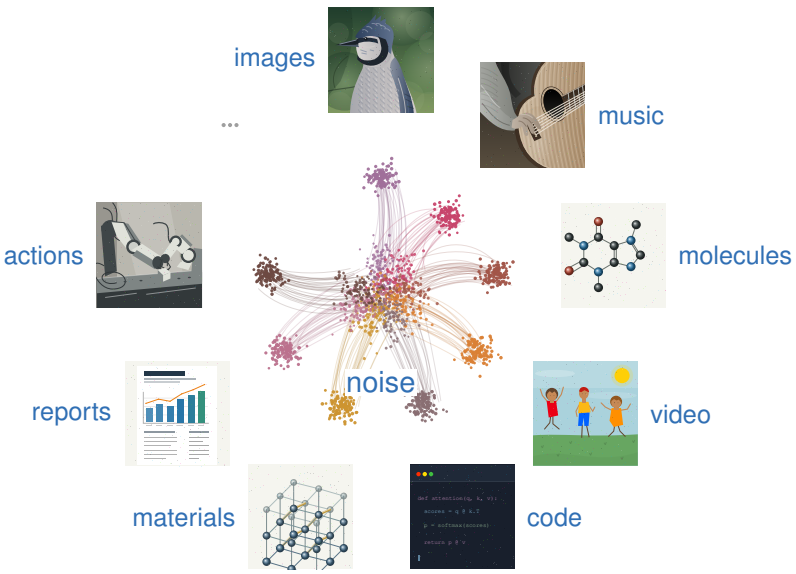

Figure 1: It is magical that any data can be generated starting from noise.

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.

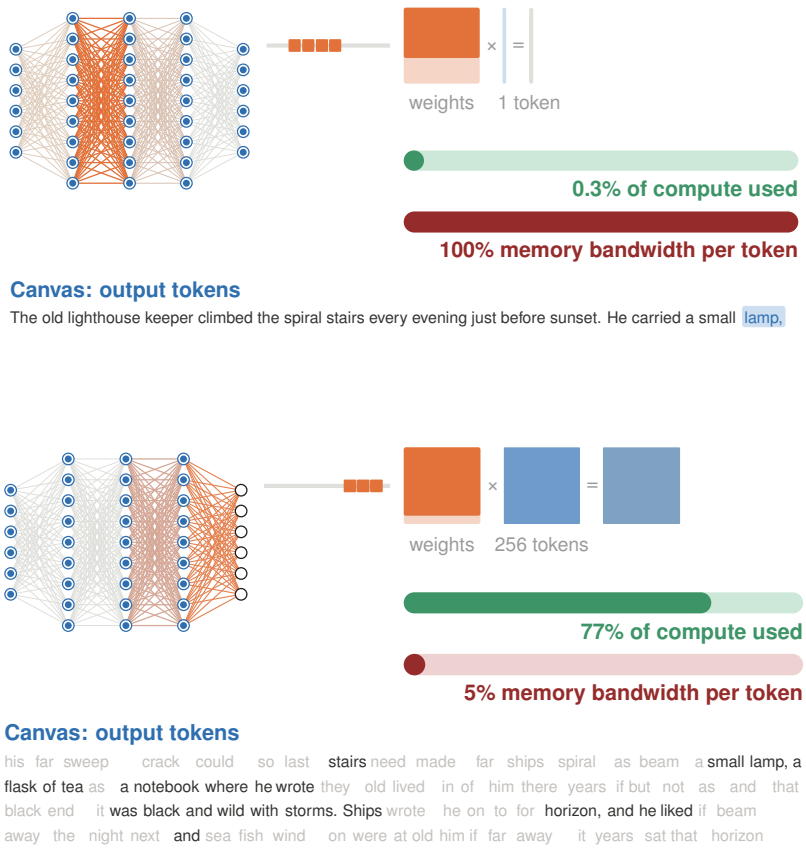

Figure 2: Autoregressive (top) vs diffusion (bottom) LLMs. Autoregressive methods generate one token at a time. For each new token, the GPUs need to move all the network weights (billions or trillions!) from one memory to registers where a small multiplication is performed. Text diffusion uses blocks of tokens and hence uses less memory bandwidth and better utilises the GPU available compute by performing big matrix multiplications. For this reason, many researchers keep trying to make text diffusion work in practice.

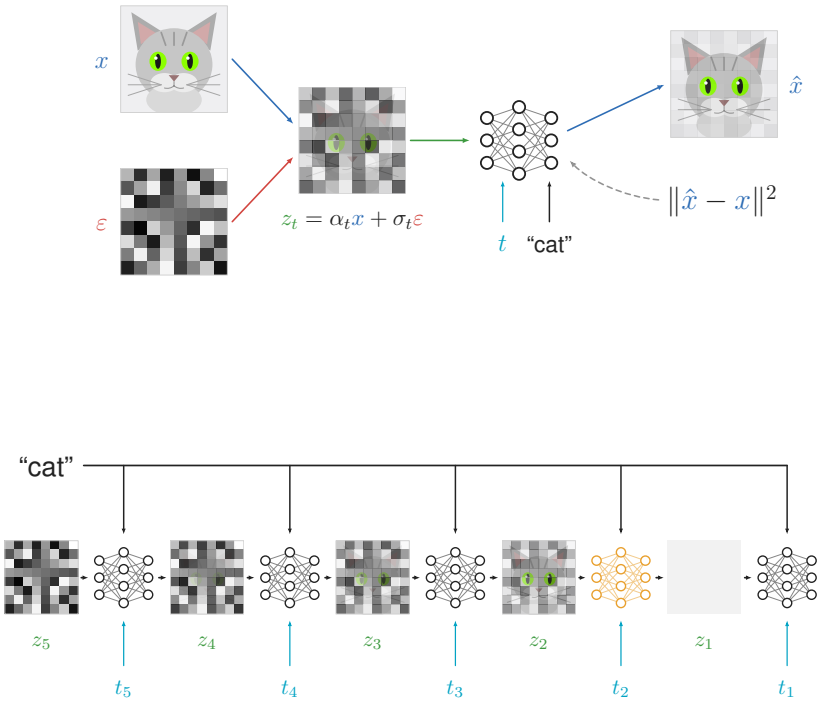

Figure 3: Training a diffusion model (top) and generating with the model (bottom). Training teaches the model to undo noise. Then at inference time, the model can start at noise and progressively move it toward clean data samples.

We then apply flow matching to text using two continuous representations: probability-simplex states in Section 8, and token embeddings in Section 9. In both cases we derive how token cross-entropy predictions can be converted into a velocity field for parallel generation. We move to discrete token states in Section 10, deriving probability paths and sampling updates from the same conditional-expectation principle. Section 12 then shows how to generate successive blocks with a reusable context cache and self-conditioning.

While diffusion for text is yet to match auto-regressive generation, it is very promising because it enables parallel inference as shown in Figure 2

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 3. 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.

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:

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

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 $\hat{\boldsymbol{\epsilon}}_{\boldsymbol{\theta}}$ at this noise level. This is exactly one call of the network as used inside `ancestral_sampling_step` which will be described later.

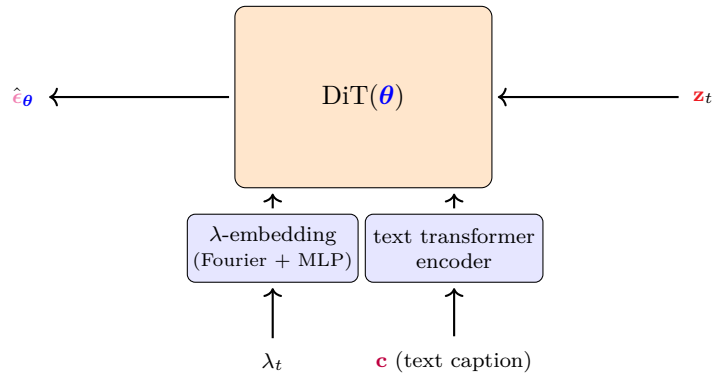

Figure 4: One call of the diffusion network, as used inside `ancestral_sampling_step`. The noisy latent $\mathbf{z}_t$ enters from the right; the scalar log-SNR λ_t and the text caption $\mathbf{c}$ enter from below, after each is lifted to a feature vector / sequence by its own encoder (λ-embedding 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:

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

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

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

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

In the high-noise regime, we have

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

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

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

The first equation for $\mathbf{x}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t)$ will be useful for generating new data, while the one for $\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\mathbf{z}_t, \lambda_t)$ 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:

```python
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 5 unrolls four iterations of the loop.

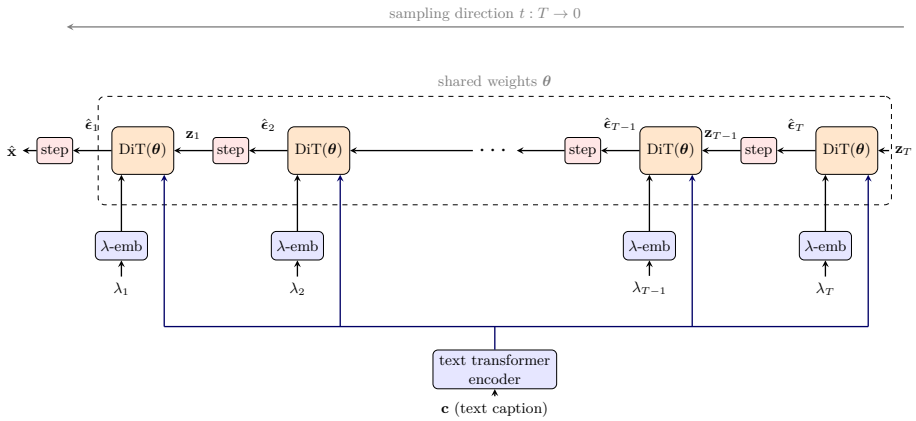

Figure 5: Inference (`generate_samples`) unrolled as a tied-parameter recurrent neural network (RNN). The *same* DiT (same weights $\boldsymbol{\theta}$) is applied at every step; what changes between steps is (i) the input latent $\mathbf{z}_t$ on the right of each block and (ii) the noise level λ_t 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.

3.1 Connection to looped and feedback language models.

Figure 5 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. This idea goes back to Hyperbolic Attention Networks (Gulcehre et al., 2018) and Universal Transformers (Dehghani et al., 2019).

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 5, 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

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

The quantity $\mathbf{z}_1 - (1-\sigma_{min})\mathbf{z}_0$ is known as the *target velocity*, and in many papers it is simply written as: $\mathbf{u}_t = \mathbf{x} - \mathbf{z}_0$.

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.
    """

```

```python
    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, transformer) 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 Text flow matching on the simplex

So far we have dealt with continuous data. However, there is great interest too in discrete data, like text. We can represent text using one-hot vectors and learn to predict the clean tokens from noisy inputs. For both this section and the next, **generation runs from noise at $t = 0$ to data at $t = 1$**, as in the flow-matching construction above.

8.1 Forward noising process

Let V be the vocabulary size and L the number of token positions. The clean token at position i is a one-hot vector $\mathbf{x}^{(i)} \in \mathbb{R}^V$: its entry for that token is one and the others are zero. For example, $(0, 1, 0)^\top$ represents the second of three vocabulary tokens. Stack these vectors as rows to form $\mathbf{x} \in \mathbb{R}^{L \times V}$.

The *probability simplex* contains vectors with nonnegative entries that sum to one. One-hot vectors are its vertices. To create noisy samples of discrete data, we can draw a random point on this simplex for each position using a Dirichlet distribution, and then mix it with the clean token:

$$\mathbf{z}_0^{(i)} \sim \text{Dirichlet}(a\mathbf{1}), \qquad a > 0, \tag{36}$$

$$\mathbf{z}_t = (1 - t)\mathbf{z}_0 + t\mathbf{x}. \tag{37}$$

This is the earlier path $\mathbf{z}_t = \alpha_t \mathbf{x} + \sigma_t \mathbf{z}_0$ with $\alpha_t = t$ and $\sigma_t = 1 - t$. Here $\mathbf{1}$ is the all-ones vector. Setting the hyper-parameter $a = 1$ gives uniform noise over the simplex. Each row of $\mathbf{z}_t$ remains on the simplex, moving from a random mixture towards the clean one-hot token as shown in Figure 6.

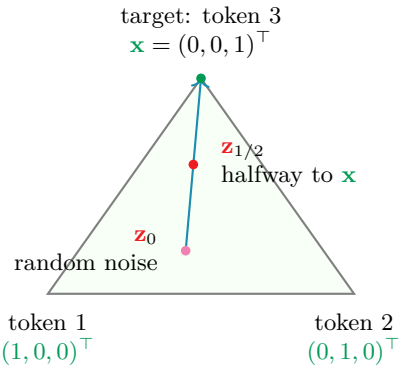

Figure 6: Three token choices give a triangular simplex. The clean token is $\mathbf{x} = (0, 0, 1)^\top$, $\mathbf{z}_0$ is random noise and $\mathbf{z}_{1/2}$ is halfway to $\mathbf{x}$.

8.2 Training and inference

Since the clean target $\mathbf{x}^{(i)}$ is one-hot, we train a classifier over the V token choices using cross-entropy. We then turn its clean-token probabilities into a sampling velocity.

8.2.1 Training: predict the clean token

Draw clean text $\mathbf{x} \sim p_d$, independent Dirichlet noise $\mathbf{z}_0$, and $t \sim \mathcal{U}(0, 1)$; form $\mathbf{z}_t$ using (37). A transformer reads the whole noisy sequence, the time, and optionally a clean prompt $\mathbf{c}$. Its forward pass returns the *logit matrix* $\ell_{\boldsymbol{\theta}}(\mathbf{z}_t, t, \mathbf{c}) \in \mathbb{R}^{L \times V}$. The entry $\ell_{\boldsymbol{\theta}}^{(i,k)}(\mathbf{z}_t, t, \mathbf{c})$ is the model's raw score for vocabulary token k at position i. Softmax exponentiates these scores and normalizes each row to give us probabilities:

$$\begin{aligned} \hat{\mathbf{x}}_{\boldsymbol{\theta}}(\mathbf{z}_t, t, \mathbf{c}) &= \operatorname{softmax}\big(\ell_{\boldsymbol{\theta}}(\mathbf{z}_t, t, \mathbf{c})\big), \\ [\hat{\mathbf{x}}_{\boldsymbol{\theta}}^{(i)}(\mathbf{z}_t, t, \mathbf{c})]_k &= \frac{\exp\big(\ell_{\boldsymbol{\theta}}^{(i,k)}(\mathbf{z}_t, t, \mathbf{c})\big)}{\sum_{j=1}^{V} \exp\big(\ell_{\boldsymbol{\theta}}^{(i,j)}(\mathbf{z}_t, t, \mathbf{c})\big)}. \end{aligned} \tag{38}$$

The resulting probabilities are positive and sum to one at each position. Write $[\mathbf{x}^{(i)}]_k$ for entry k of the one-hot target. The loss is

$$\boxed{\mathcal{L}_{\mathrm{CE}}(\boldsymbol{\theta}) = -\mathbb{E}\left[\frac{1}{L}\sum_{i=1}^{L}\sum_{k=1}^{V}[\mathbf{x}^{(i)}]_k \log[\hat{\mathbf{x}}_{\boldsymbol{\theta}}^{(i)}(\mathbf{z}_t, t, \mathbf{c})]_k\right].} \tag{39}$$

Only one entry of $\mathbf{x}^{(i)}$ is one, so the inner sum simply selects the log probability of the correct token. For example, if the target is $(0, 1, 0)^\top$ and the prediction is $(0.1, 0.7, 0.2)^\top$, its loss is

$$-\big(0 \log 0.1 + 1 \log 0.7 + 0 \log 0.2\big) = -\log 0.7.$$

Giving more probability to the correct token lowers this loss.

For a fixed noisy input, cross-entropy is minimized when predicted probabilities match the true probabilities of the clean tokens. An average of one-hot vectors is exactly this vector of probabilities. In other words, the expectation of an event is the same thing as the probability of the event. Thus the ideal prediction is

$$\hat{\mathbf{x}}_*(\mathbf{z}_t, t, \mathbf{c}) = \mathbb{E}[\mathbf{x} \mid \mathbf{z}_t, t, \mathbf{c}]. \tag{40}$$

8.2.2 Inference: turn the prediction into a velocity field

With both training endpoints fixed, define the *target velocity* $\mathbf{u}_t = \mathbf{x} - \mathbf{z}_0$ can be expanded as follows:

$$\mathbf{u}_t := \mathbf{x} - \mathbf{z}_0 = \mathbf{x} - \frac{\mathbf{z}_t - t\mathbf{x}}{1-t} = \frac{(1-t)\mathbf{x} - \mathbf{z}_t + t\mathbf{x}}{1-t} = \frac{\mathbf{x} - \mathbf{z}_t}{1-t}, \qquad t < 1. \tag{41}$$

During generation, $\mathbf{x}$ is unknown. As in the earlier flow-matching derivation, the network estimates it with $\hat{\mathbf{x}}_{\boldsymbol{\theta}}$. This enables us to fit the target velocity:

$$\boxed{\mathbf{v}_{\boldsymbol{\theta}}(\mathbf{z}_t, t, \mathbf{c}) = \frac{\hat{\mathbf{x}}_{\boldsymbol{\theta}}(\mathbf{z}_t, t, \mathbf{c}) - \mathbf{z}_t}{1 - t + \varepsilon_{\text{num}}}.} \tag{42}$$

For numerical sampling, we add a small constant $\varepsilon_{\text{num}} > 0$, such as 10^{-4}, to the denominator.

We can now write a simple sampling recipe. Start with Dirichlet noise. For increasing times $0 = t_0 < \cdots < t_N = 1$, use Euler updates

$$\mathbf{z}_{n+1} = \mathbf{z}_n + (t_{n+1} - t_n)\frac{\hat{\mathbf{x}}_{\boldsymbol{\theta}}(\mathbf{z}_n, t_n, \mathbf{c}) - \mathbf{z}_n}{1 - t_n + \varepsilon_{\text{num}}}. \tag{43}$$

Here, $\mathbf{z}_n$ denotes the numerical state at time t_n. Since $t_{n+1} - t_n \le 1 - t_n$, each update is a weighted average of two simplex points and stays on the simplex. At the end, predict the token probabilities once more and sample a token, or choose the most likely token, at each position.

9 Text flow matching in embedding space

The second approach runs the flow over token embeddings and still trains by classifying the clean tokens. The hope is that the embeddings will be lower-dimensional than the vocabulary size and provide a better representation. This is similar to using an existing VAE in flow matching for images or video. This is the approach discussed in LangFlow (Chen et al., 2026) and earlier work by Sander Dieleman. Again, noise is at $t = 0$ and clean data at $t = 1$.

9.1 Notation: from tokens to embeddings

Keep the one-hot sequence $\mathbf{x} \in \mathbb{R}^{L \times V}$ and let $\mathbf{E} \in \mathbb{R}^{V \times D}$ be a fixed embedding matrix: each row contains one token's D-dimensional embedding. (This matrix could have been learned by an AR model.) A one-hot vector selects the corresponding row:

$$\begin{aligned} \mathbf{z}_1^{(i)} &= \mathbf{E}^\top \mathbf{x}^{(i)} \in \mathbb{R}^D, \\ \mathbf{z}_1 &= \mathbf{x}\mathbf{E} \in \mathbb{R}^{L \times D}. \end{aligned} \tag{44}$$

The second line stacks all L token embeddings.

9.2 Training: forward pass and loss function

Draw a clean sequence and $t \sim \mathcal{U}(0, 1)$. Sample independent standard Gaussian noise for every embedding coordinate and interpolate:

$$\mathbf{z}_0^{(i)} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \qquad \mathbf{z}_t = (1-t)\mathbf{z}_0 + t\mathbf{z}_1. \tag{45}$$

This constructs the noisy training input directly, without simulating many noising steps.

The transformer's **forward pass** reads the whole $L \times D$ noisy embedding array, the time, and an optional clean prompt $\mathbf{c}$. It returns $L \times V$ logits $\ell_{\boldsymbol{\theta}}(\mathbf{z}_t, t, \mathbf{c})$, from which we obtain the token probability estimates:

$$\hat{\mathbf{x}}_{\boldsymbol{\theta}}(\mathbf{z}_t, t, \mathbf{c}) = \mathrm{softmax}\big(\ell_{\boldsymbol{\theta}}(\mathbf{z}_t, t, \mathbf{c})\big). \tag{46}$$

As in the previous section, we use the original one-hot tokens as targets, with the same cross-entropy loss:

$$\boxed{\mathcal{L}_{\mathrm{CE}}(\boldsymbol{\theta}) = -\mathbb{E}\left[\frac{1}{L}\sum_{i=1}^{L}\sum_{k=1}^{V}[\mathbf{x}^{(i)}]_k \log[\hat{\mathbf{x}}_{\boldsymbol{\theta}}^{(i)}(\mathbf{z}_t, t, \mathbf{c})]_k\right].} \tag{47}$$

9.3 Inference: derive the denoising update

We first convert the predicted token probabilities into a predicted clean embedding:

$$\boxed{\hat{\mathbf{z}}_{1,\boldsymbol{\theta}}(\mathbf{z}_t, t, \mathbf{c}) = \hat{\mathbf{x}}_{\boldsymbol{\theta}}(\mathbf{z}_t, t, \mathbf{c})\mathbf{E}.} \tag{48}$$

This is a probability-weighted average of the embedding rows. Since the matrix is fixed, the ideal classifier gives the conditional mean embedding:

$$\mathbb{E}[\mathbf{x} \mid \mathbf{z}_t, t, \mathbf{c}]\mathbf{E} = \mathbb{E}[\mathbf{x}\mathbf{E} \mid \mathbf{z}_t, t, \mathbf{c}] = \mathbb{E}[\mathbf{z}_1 \mid \mathbf{z}_t, t, \mathbf{c}].$$

As before, we expand the target velocity $\mathbf{u}_t$, eliminating the initial noise, as follows:

$$\mathbf{u}_t := \mathbf{z}_1 - \mathbf{z}_0 = \mathbf{z}_1 - \frac{\mathbf{z}_t - t\mathbf{z}_1}{1-t} = \frac{\mathbf{z}_1 - \mathbf{z}_t}{1-t}, \qquad t < 1. \tag{49}$$

During generation the clean endpoint is unknown, so we replace it by its predicted mean as before:

$$\boxed{\mathbf{v}_{\boldsymbol{\theta}}(\mathbf{z}_t, t, \mathbf{c}) = \frac{\hat{\mathbf{z}}_{1,\boldsymbol{\theta}}(\mathbf{z}_t, t, \mathbf{c}) - \mathbf{z}_t}{1 - t + \varepsilon_{\text{num}}}.} \tag{50}$$

This points towards the predicted clean embedding. We start from Gaussian noise this time (not Dirichlet as in the simplex method) and increase time from zero to one using

$$\mathbf{z}_{n+1} = \mathbf{z}_n + (t_{n+1} - t_n)\frac{\hat{\mathbf{z}}_{1,\boldsymbol{\theta}}(\mathbf{z}_n, t_n, \mathbf{c}) - \mathbf{z}_n}{1 - t_n + \varepsilon_{\text{num}}}. \tag{51}$$

All positions update together. At the end, we run the classifier once more and decode its token probabilities.

Author remark: The LangFlow paper notes that the loss is high only for $t < 0.2$, and uses this fact to propose a different noise scheduler. It also uses *self-conditioning* to improve the results. This trick seems to be used by everyone, which makes me wonder if this is what makes text diffusion work? I think Caglar Gulcehre has been exploring this question.

10 Discrete text diffusion

Up to now, we have considered continuous diffusion approaches for text. These are very promising, but have not yet been tested on a scale. The most impressive implementations at scale, thus far, use discrete diffusion, which we cover in this section.

In discrete diffusion, we generate text by replacing discrete tokens. The recipe is the same as for diffusion earlier: choose a forward noising rule, train a network to predict the clean data, and use that prediction to take backward sampling steps. We will derive the forward law (53) and the backward update (65) directly. These are the uniform token path and denoiser-based update of *Discrete Flow Matching*, also used in DiffusionGemma. Figure 7 shows the idea: the state is a row of actual tokens, which are kept or replaced as noise is removed, proceeding from the random tokens sentence at the top to the clear sentence at the bottom.

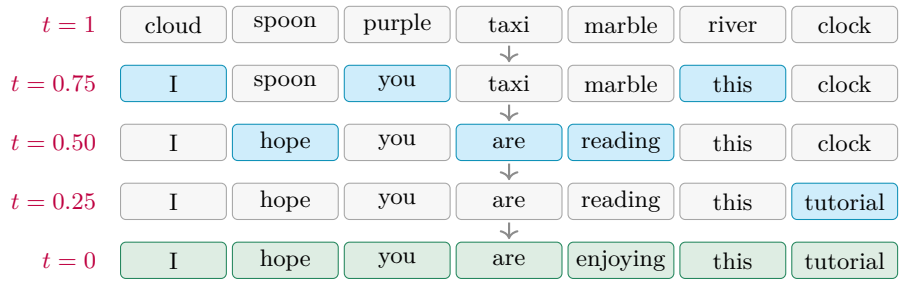

Figure 7: Discrete text diffusion in the uniform sampling setting. Each word stands for one token. The final green row is one possible coherent sentence.

10.1 Notation and the forward noising rule

We keep the earlier notation. There are L token positions and a vocabulary of size V. The green vector $\mathbf{x}^{(i)}$ is the clean token at position i, and the red vector $\mathbf{z}_t^{(i)}$ is its noisy version. Both are *one-hot*: each equals one of the basis vectors $\mathbf{e}_k$, whose kth entry is one and all other entries are zero. The notation $[\mathbf{x}^{(i)}]_k$ means entry k of that vector.

Here we return to the diffusion convention used earlier: $t = 0$ is clean text and $t = 1$ is noise. The schedule α_t now gives the probability of keeping a clean token. It decreases smoothly from $\alpha_0 = 1$ to $\alpha_1 = 0$, with $\alpha_t < 1$ for $t > 0$.

At position i, we keep $\mathbf{x}^{(i)}$ with probability α_t. Otherwise, we draw a token uniformly from the vocabulary. There are two ways to observe token k: keep it if it was the clean token, or draw it through the uniform branch. Adding the probabilities of these two branches gives

$$\Pr(\mathbf{z}_t^{(i)} = \mathbf{e}_k \mid \mathbf{x}) = \underbrace{\alpha_t [\mathbf{x}^{(i)}]_k}_{\text{keep the clean token}} + \underbrace{(1 - \alpha_t)\frac{1}{V}}_{\text{draw token } k \text{ uniformly}}. \tag{52}$$

We apply independent corruption draws at all positions, given the clean sequence. For the particular noisy sequence we observe, let k_i be its token index at position i, so $\mathbf{z}_t^{(i)} = \mathbf{e}_{k_i}$. Use (52) with $k = k_i$ at each position and multiply:

$$\boxed{p(\mathbf{z}_t \mid \mathbf{x}) = \prod_{i=1}^{L} \left[\alpha_t \, [\mathbf{x}^{(i)}]_{k_i} + \frac{1 - \alpha_t}{V} \right].} \tag{53}$$

Here α_t depends only on time and multiplies the selected entry of $\mathbf{x}^{(i)}$. This is the forward noising equation. At $t = 0$, it puts all probability on $\mathbf{z}_0 = \mathbf{x}$. At $t = 1$, every token is independently uniform. The probability vector changes smoothly with time, but a sampled $\mathbf{z}_t^{(i)}$ always remains a one-hot token. We are not interpolating the token vector as we did in the continuous simplex construction.

10.2 Training with cross-entropy

As before, a transformer reads the whole noisy sequence and produces $L \times V$ logits, and we apply a softmax across the vocabulary:

$$\hat{\mathbf{x}}_{\boldsymbol{\theta}}(\mathbf{z}_t, t) = \mathrm{softmax}\big(\ell_{\boldsymbol{\theta}}(\mathbf{z}_t, t)\big). \tag{54}$$

Row i predicts which clean token occupied position $i \in \{1 : L\}$. Since $\mathbf{x}^{(i)}$ is one-hot, we use the cross-entropy loss:

$$\boxed{\mathcal{L}_{\mathrm{CE}}(\boldsymbol{\theta}) = -\mathbb{E}\left[\frac{1}{L}\sum_{i=1}^{L}\sum_{k=1}^{V}[\mathbf{x}^{(i)}]_k \log[\hat{\mathbf{x}}_{\boldsymbol{\theta}}^{(i)}(\mathbf{z}_t, t)]_k\right].} \tag{55}$$

To obtain a training example, draw clean text $\mathbf{x} \sim p_d$, draw t uniformly in $(0, 1)$, and sample $\mathbf{z}_t$ using (53). One network call supplies the loss. That is, training does not require running a denoising trajectory.

As before, the ideal predictor satisfies

$$\begin{aligned}[\hat{\mathbf{x}}_*^{(i)}(\mathbf{z}_t, t)]_k &= \Pr(\mathbf{x}^{(i)} = \mathbf{e}_k \mid \mathbf{z}_t, t) \\ &= \mathbb{E}\big[[\mathbf{x}^{(i)}]_k \mid \mathbf{z}_t, t\big].\end{aligned} \tag{56}$$

10.3 The backward step when we know the clean token

Our goal is the transformation in Figure 7: replace random tokens over time until we obtain a coherent sentence, such as "I hope you are enjoying this tutorial". We must choose how often to keep or replace each token so that the distribution follows the desired probability path. The final sentence is unknown during generation. We first derive the update assuming its clean tokens are known in this section, then replace that knowledge by the model's prediction in the following section.

Consider a step $0 < \Delta t \le t$ from t to $t - \Delta t$. The clean weight must increase from α_t to $\alpha_{t-\Delta t}$. Fix a token position i and suppose, for this derivation, that we know its clean token $\mathbf{x}^{(i)}$. At time t, its noisy version $\mathbf{z}_t^{(i)}$ is generated in one of two ways: copy the clean token with probability α_t, or draw a uniformly random token with probability $1 - \alpha_t$.

We want to move to the earlier time $t - \Delta t$, where the desired uniform weight is $1 - \alpha_{t-\Delta t}$. Consider the following update: toss a fresh coin and either *keep the current token* or *replace it with the clean token*. The coin is independent of how the current token was generated.

To see how this changes the mixture, follow its uniform component. A draw from that component survives only when we choose to keep the current token. We therefore choose the keep probability so that the surviving uniform weight equals the desired uniform weight:

$$\underbrace{\Pr(\text{keep current token})}_{\text{fraction retained}} \underbrace{(1 - \alpha_t)}_{\text{current uniform weight}} = \underbrace{1 - \alpha_{t-\Delta t}}_{\text{desired uniform weight}}. \tag{57}$$

Dividing by the current uniform weight gives

$$\boxed{\Pr(\text{keep current token}) = \frac{1-\alpha_{t-\Delta t}}{1-\alpha_t}.} \tag{58}$$

The other branch has the remaining probability:

$$\Pr(\text{copy clean token}) = 1 - \frac{1-\alpha_{t-\Delta t}}{1-\alpha_t} = \frac{\alpha_{t-\Delta t}-\alpha_t}{1-\alpha_t}. \tag{59}$$

The current and clean tokens are one-hot. Their entries therefore give the probabilities within each branch. Combining the branches,

$$\Pr(\mathbf{z}_{t-\Delta t}^{(i)} = \mathbf{e}_k \mid \mathbf{z}_t, \mathbf{x}) \quad = \frac{1-\alpha_{t-\Delta t}}{1-\alpha_t}[\mathbf{z}_t^{(i)}]_k + \frac{\alpha_{t-\Delta t}-\alpha_t}{1-\alpha_t}[\mathbf{x}^{(i)}]_k. \tag{60}$$

Finally, write the keep probability as one minus the copy probability and collect terms in (60):

$$\Pr(\mathbf{z}_{t-\Delta t}^{(i)} = \mathbf{e}_k \mid \mathbf{z}_t, \mathbf{x}) \quad = [\mathbf{z}_t^{(i)}]_k + \frac{\alpha_{t-\Delta t}-\alpha_t}{1-\alpha_t}\big([\mathbf{x}^{(i)}]_k - [\mathbf{z}_t^{(i)}]_k\big). \tag{61}$$

10.4 Use the prediction to obtain the backward update

During generation, $\mathbf{z}_t$ is observed but $\mathbf{x}$ is unknown. Average (61) over the possible clean tokens. Everything except $\mathbf{x}$ is fixed when we condition on the current noisy sequence, so

$$\begin{aligned}\Pr(\mathbf{z}_{t-\Delta t}^{(i)} = \mathbf{e}_k \mid \mathbf{z}_t, t) &= \mathbb{E}\Big[\Pr(\mathbf{z}_{t-\Delta t}^{(i)} = \mathbf{e}_k \mid \mathbf{z}_t, \mathbf{x}, t) \mid \mathbf{z}_t, t\Big] \\ &= [\mathbf{z}_t^{(i)}]_k + \frac{\alpha_{t-\Delta t}-\alpha_t}{1-\alpha_t} \times \Big(\mathbb{E}\big[[\mathbf{x}^{(i)}]_k \mid \mathbf{z}_t, t\big] - [\mathbf{z}_t^{(i)}]_k\Big).\end{aligned} \tag{62}$$

This is the conditional-expectation trick again. We only need the mean clean one-hot vector, which is what cross-entropy learns.

Now, here is an important trick researchers in this topic use. Don't forget it! For a small step, use the first-order Taylor expansion of the schedule:

$$\begin{aligned}\alpha_{t-\Delta t} &\approx \alpha_t - \Delta t\, \dot\alpha_t, \\ \frac{\alpha_{t-\Delta t}-\alpha_t}{1-\alpha_t} &\approx -\frac{\Delta t\, \dot\alpha_t}{1-\alpha_t}.\end{aligned} \tag{63}$$

Substituting into (62) gives

$$\Pr(\mathbf{z}_{t-\Delta t}^{(i)} = \mathbf{e}_k \mid \mathbf{z}_t, t) \approx [\mathbf{z}_t^{(i)}]_k - \frac{\Delta t\, \dot\alpha_t}{1-\alpha_t}\Big(\mathbb{E}\big[[\mathbf{x}^{(i)}]_k \mid \mathbf{z}_t, t\big] - [\mathbf{z}_t^{(i)}]_k\Big). \tag{64}$$

Replace the unknown conditional mean by the trained prediction from (56). This gives us the expression for the backward sampler update:

$$\boxed{p_{\boldsymbol\theta}(\mathbf{z}_{t-\Delta t}^{(i)} = \mathbf{e}_k \mid \mathbf{z}_t, t) = [\mathbf{z}_t^{(i)}]_k - \frac{\Delta t\, \dot\alpha_t}{1-\alpha_t}\Big([\hat{\mathbf{x}}_{\boldsymbol\theta}^{(i)}(\mathbf{z}_t, t)]_k - [\mathbf{z}_t^{(i)}]_k\Big).} \tag{65}$$

Here $[\hat{\mathbf{x}}_{\boldsymbol{\theta}}^{(i)}]_k$ is the predicted probability of clean token k, while $[\mathbf{z}_t^{(i)}]_k$ is one if the current token is k and zero otherwise. Their difference is a scalar, not a vector. The schedule is decreasing, so $-\dot{\alpha}_t \geq 0$: probability moves towards the clean-token prediction. Choose the step so that $0 \leq -\Delta t \dot{\alpha}_t/(1-\alpha_t) \leq 1$. This makes the update a valid mixture of the current token and the prediction.

The Taylor approximation has implications. The transformer sees the *whole current sequence*, so its predictions can express how one token depends on the others. However, independent draws for the next positions do not see each other's *newly sampled choices* until the next network call. If two plausible continuations are "I am" and "you are", drawing their words independently can also produce "I are".

Let us contrast this with continuous diffusion for text. The embedding flow in Section 9 moves the whole embedding array through a coupled velocity field: the velocity at one position can depend on every other position. It makes no categorical token draws during these ODE steps. This avoids the particular approximation of independently committing several token identities at each update and lets continuous representations evolve together before final decoding.

For $\alpha_t = 1 - t$, we have $\dot{\alpha}_t = -1$ and $1 - \alpha_t = t$. Therefore

$$\boxed{p_{\boldsymbol{\theta}}(\mathbf{z}_{t-\Delta t}^{(i)} = \mathbf{e}_k \mid \mathbf{z}_t, t) = \left(1 - \frac{\Delta t}{t}\right)[\mathbf{z}_t^{(i)}]_k + \frac{\Delta t}{t}[\hat{\mathbf{x}}_{\boldsymbol{\theta}}^{(i)}(\mathbf{z}_t, t)]_k.} \tag{66}$$

This tells us exactly how to sample: keep the current token with probability $1 - \Delta t/t$, or draw from the predicted clean-token distribution with probability $\Delta t/t$.

Start at $t = 1$ with independent uniform tokens. At each step, run the network, build these probabilities, and draw the next tokens. Use $0 < \Delta t \leq t$ and continue towards zero.

11 Masked discrete text diffusion

Section 10 corrupts a token by replacing it with a uniformly chosen vocabulary token. Here we make a different choice: replace it with a special `[MASK]` symbol. A visible word is then known to have survived the corruption, and a mask tells us exactly where information is missing. Generation starts with masks and fills them in. This is illustrated in Figure 8.

We will derive the absorbing-state construction introduced in Austin et al., *Structured Denoising Diffusion Models in Discrete State-Spaces* (D3PM, 2021), using the clean-token parameterisation of Shi et al., *Simplified and Generalized Masked Diffusion for Discrete Data* (MD4, 2024). With a linear masking schedule, its training loss is the one used in Prabhudesai et al., *Diffusion Beats Autoregressive in Data-Constrained Settings* (2025).

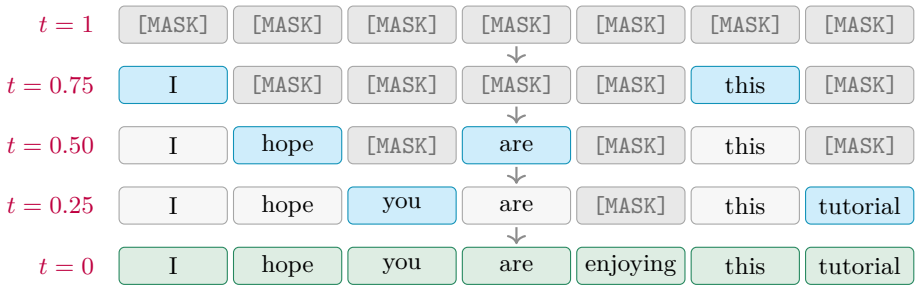

Figure 8: Masked diffusion. Cyan marks newly revealed tokens, grey boxes are still masked. Once revealed, a token stays fixed under the standard absorbing reverse sampler.

11.1 Forward process

We retain the convention of Section 10: $t = 0$ is clean text and $t = 1$ is complete noise. The clean vocabulary has V tokens. Add one extra state with index $m = V + 1$, represented by the one-hot vector $\mathbf{m}$ and displayed as `[MASK]`. Clean token vectors $\mathbf{x}^{(i)}$ now have $V + 1$ coordinates with a zero in coordinate m. The noisy token $\mathbf{z}_t^{(i)}$ can equal either a clean vocabulary token or $\mathbf{m}$. We write $\mathbf{e}_k$ for the one-hot vector of clean token $k \in \{1, \dots, V\}$ and use the scalar indicator

$$M_t^{(i)} = \mathbb{I}\{\mathbf{z}_t^{(i)} = \mathbf{m}\} = [\mathbf{z}_t^{(i)}]_m. \tag{67}$$

Thus $M_t^{(i)} = 1$ means that position i is masked. The vectors remain discrete one-hot states throughout sampling; soft probabilities describe distributions over these states.

Let α_t be the probability that a clean token survives. Assume that it decreases continuously from $\alpha_0 = 1$ to $\alpha_1 = 0$, with $0 < \alpha_t < 1$ for $0 < t < 1$. At a chosen time, keep the clean token with probability α_t and otherwise replace

it by $\mathbf{m}$. Therefore, for $k \in \{1, \dots, V, m\}$,

$$\boxed{q(\mathbf{z}_t^{(i)} = \mathbf{e}_k \mid \mathbf{x}) = \alpha_t [\mathbf{x}^{(i)}]_k + (1 - \alpha_t)[\mathbf{m}]_k.} \tag{68}$$

Here q denotes the prescribed corruption process; it plays the role of the forward $p(\mathbf{z}_t \mid \mathbf{x})$ in Section 10. Unlike uniform replacement, the noise branch cannot accidentally produce the correct clean token. It always produces the distinguished mask.

Corruption is independent across positions *conditional on the clean sequence*. If k_i is the observed token index at position i, the full forward distribution is

$$\boxed{q(\mathbf{z}_t \mid \mathbf{x}) = \prod_{i=1}^{L} \left[\alpha_t [\mathbf{x}^{(i)}]_{k_i} + (1 - \alpha_t)[\mathbf{m}]_{k_i} \right].} \tag{69}$$

At $t = 0$ this gives $\mathbf{z}_0 = \mathbf{x}$; at $t = 1$ every position is masked.

For an absorbing forward trajectory, a masked token must stay masked. Consider $0 \le s < t \le 1$. A token that survived to s must survive to t with probability α_t/α_s, because its total survival probability must be $\alpha_s(\alpha_t/\alpha_s) = \alpha_t$. Hence

$$\boxed{q(\mathbf{z}_t^{(i)} \mid \mathbf{z}_s^{(i)}) = \mathrm{Cat}\left(\frac{\alpha_t}{\alpha_s} \mathbf{z}_s^{(i)} + \left(1 - \frac{\alpha_t}{\alpha_s}\right) \mathbf{m} \right).} \tag{70}$$

$\mathrm{Cat}(\mathbf{p})$ means a categorical draw with probability vector $\mathbf{p}$. If $\mathbf{z}_s^{(i)} = \mathbf{m}$, both terms in (70) point to the mask, so its probability remains one. This is why the state is called *absorbing*.

For the rest of the worked examples choose

$$\alpha_t = 1 - t, \qquad \Pr(M_t^{(i)} = 1) = t. \tag{71}$$

At training time we can draw a fresh mask pattern directly using (69), and no intermediate states are needed. If we want to draw a *trajectory*, however, we use (70) so that masks are nested in time.

11.2 Transformer model predictions

The network reads the whole partially masked sequence, including the positions of the masks, and outputs $L \times V$ clean-token logits. At a masked position, apply softmax over the V ordinary tokens:

$$[\hat{\mathbf{x}}_{\boldsymbol{\theta}}^{(i)}(\mathbf{z}_t, t)]_k = \frac{\exp(\ell_{\boldsymbol{\theta},k}^{(i)}(\mathbf{z}_t, t))}{\sum_{a=1}^{V} \exp(\ell_{\boldsymbol{\theta},a}^{(i)}(\mathbf{z}_t, t))}, \qquad k \le V. \tag{72}$$

Append a zero mask coordinate. At a visible position, copy the input one-hot vector exactly. These two constraints are

$$\begin{aligned} [\hat{\mathbf{x}}_{\boldsymbol{\theta}}^{(i)}(\mathbf{z}_t, t)]_m &= 0, \\ \hat{\mathbf{x}}_{\boldsymbol{\theta}}^{(i)}(\mathbf{z}_t, t) &= \mathbf{z}_t^{(i)} \quad \text{if } M_t^{(i)} = 0. \end{aligned} \tag{73}$$

We are predicting clean data, so the mask must not be a possible clean output. The copy rule is enforced by construction because we do not need a network to relearn a visible token.

Note that we can use *bidirectional* attention. That is, a missing word can use visible words on either side. The training target at position i is the clean token at that same position, with no next-token shift. Attention to a visible future word is intentional, and access to a masked word's clean embedding would be leakage. Only corrupted inputs enter the network.

For an ideal denoiser the desired output is

$$\hat{\mathbf{x}}_*^{(i)}(\mathbf{z}_t, t) = \mathbb{E}[\mathbf{x}^{(i)} \mid \mathbf{z}_t, t]. \tag{74}$$

Each coordinate is the conditional probability of a clean token. The expectation conditions on the *entire* noisy sentence, not just on the mask at position i.

11.3 Backward step when the clean token is known

We now move backwards from t to $s < t$. Fix one position and temporarily assume that its clean token $\mathbf{x}^{(i)}$ is known. There are two cases.

Case 1: the current token is visible. If $\mathbf{z}_t^{(i)} \neq \mathbf{m}$, it must equal $\mathbf{x}^{(i)}$. It could not have been masked at the earlier time s, since a mask cannot become a word in the forward process. Therefore

$$q(\mathbf{z}_s^{(i)} = \mathbf{z}_t^{(i)} \mid \mathbf{z}_t^{(i)}, \mathbf{x}) = 1. \tag{75}$$

Case 2: the current token is masked. It was either already masked at s, or was still clean at s and became masked between s and t. The latter event has probability

$$\begin{aligned} &\Pr(\mathbf{z}_s^{(i)} = \mathbf{x}^{(i)}, \mathbf{z}_t^{(i)} = \mathbf{m} \mid \mathbf{x}) \\ &\quad = \underbrace{\alpha_s}_{\text{survive to } s} \underbrace{\left(1 - \frac{\alpha_t}{\alpha_s}\right)}_{\text{mask between } s \text{ and } t} = \alpha_s - \alpha_t. \end{aligned} \tag{76}$$

Conditioning on the observed mask means dividing by its probability $1 - \alpha_t$. Bayes' rule thus gives the reveal probability

$$\boxed{\rho_{s,t} := \Pr(\mathbf{z}_s^{(i)} = \mathbf{x}^{(i)} \mid \mathbf{z}_t^{(i)} = \mathbf{m}, \mathbf{x}) = \frac{\alpha_s - \alpha_t}{1 - \alpha_t}.} \tag{77}$$

Similarly, being masked at s implies being masked at t, so

$$\Pr(\mathbf{z}_s^{(i)} = \mathbf{m} \mid \mathbf{z}_t^{(i)} = \mathbf{m}, \mathbf{x}) = \frac{1 - \alpha_s}{1 - \alpha_t} = 1 - \rho_{s,t}. \tag{78}$$

Both probabilities are nonnegative and sum to one. Combining the two cases, the exact conditional backward distribution at this position is

$$\boxed{q(\mathbf{z}_s^{(i)} \mid \mathbf{z}_t, \mathbf{x}) = \mathrm{Cat}\Big(\mathbf{z}_t^{(i)} + M_t^{(i)} \rho_{s,t} (\mathbf{x}^{(i)} - \mathbf{m})\Big).} \tag{79}$$

To see this, for a visible token $M_t^{(i)} = 0$, and thus we copy it. On the other hand, for a mask, the probability vector is $(1 - \rho_{s,t})\mathbf{m} + \rho_{s,t}\mathbf{x}^{(i)}$.

11.4 Backward pass with network predictions

During generation we know $\mathbf{z}_t$ but not $\mathbf{x}$. Apply the conditional-expectation trick to (79). For any output coordinate k, including the mask coordinate,

$$\begin{aligned} q(\mathbf{z}_s^{(i)} = \mathbf{e}_k \mid \mathbf{z}_t, t) = [\mathbf{z}_t^{(i)}]_k + M_t^{(i)} \rho_{s,t} \\ \times \Big(\mathbb{E}[[\mathbf{x}^{(i)}]_k \mid \mathbf{z}_t, t] - [\mathbf{m}]_k\Big). \end{aligned} \tag{80}$$

Replace the conditional mean by the transformer prediction. We obtain

$$\boxed{p_{\boldsymbol{\theta}}(\mathbf{z}_s^{(i)} \mid \mathbf{z}_t, t) = \mathrm{Cat}\Big(\mathbf{z}_t^{(i)} + M_t^{(i)} \rho_{s,t} (\hat{\mathbf{x}}_{\boldsymbol{\theta}}^{(i)}(\mathbf{z}_t, t) - \mathbf{m})\Big).} \tag{81}$$

Equivalently, the complete sampling rule is

$$\boxed{\begin{aligned} \mathbf{z}_t^{(i)} \neq \mathbf{m}: &\quad \mathbf{z}_s^{(i)} = \mathbf{z}_t^{(i)}, \\ \mathbf{z}_t^{(i)} = \mathbf{m}: &\quad \mathbf{z}_s^{(i)} \sim \mathrm{Cat}\Big((1 - \rho_{s,t})\mathbf{m} + \rho_{s,t} \hat{\mathbf{x}}_{\boldsymbol{\theta}}^{(i)}(\mathbf{z}_t, t)\Big). \end{aligned}} \tag{82}$$

There are two random decisions at a masked position: whether to reveal a token, and which token to reveal. The first is set by the schedule; the second is learned from data. A high predicted token probability does not, in this ancestral rule, increase the probability of selecting that position for revelation.

With $\alpha_t = 1 - t$ and $s = t - \Delta t$, the coefficient is exactly

$$\boxed{\rho_{s,t} = \frac{t - s}{t} = \frac{\Delta t}{t}.} \tag{83}$$

For a general differentiable schedule and a small step, $\rho_{t-\Delta t,t} = -\Delta t\, \dot{\alpha}_t/(1-\alpha_t) + o(\Delta t)$. Thus the reverse reveal rate is $-\dot{\alpha}_t/(1-\alpha_t)$. The *forward* masking rate is instead $-\dot{\alpha}_t/\alpha_t$, obtained by expanding (70). For the linear schedule these rates are $1/t$ and $1/(1-t)$, respectively.

11.5 Training loss

The clean-token cross-entropy at position i is given by:

$$C_i(\boldsymbol{\theta}; \mathbf{x}, \mathbf{z}_t, t) = -\sum_{k=1}^{V} [\mathbf{x}^{(i)}]_k \log[\hat{\mathbf{x}}_{\boldsymbol{\theta}}^{(i)}(\mathbf{z}_t, t)]_k. \tag{84}$$

Crucially, only the entry for the correct clean token contributes to this sum.

Suppose a position is masked and its clean token has index a. The true clean-conditioned backward distribution puts mass $1-\rho_{s,t}$ on the mask and $\rho_{s,t}$ on a. The model puts the same mass on the mask and mass $\rho_{s,t}[\hat{\mathbf{x}}_{\boldsymbol{\theta}}^{(i)}]_a$ on a. Expanding $\mathrm{KL}(q\|p_{\boldsymbol{\theta}}) = \sum_k q_k \log(q_k/[p_{\boldsymbol{\theta}}]_k)$ gives

$$\begin{aligned}\mathrm{KL}(q\|p_{\boldsymbol{\theta}}) &= (1-\rho_{s,t})\log\frac{1-\rho_{s,t}}{1-\rho_{s,t}} + \rho_{s,t}\log\frac{\rho_{s,t}}{\rho_{s,t}[\hat{\mathbf{x}}_{\boldsymbol{\theta}}^{(i)}]_a} \\ &= -\rho_{s,t}\log[\hat{\mathbf{x}}_{\boldsymbol{\theta}}^{(i)}]_a = \rho_{s,t}C_i(\boldsymbol{\theta};\mathbf{x},\mathbf{z}_t,t).\end{aligned} \tag{85}$$

Only one term of this loss depended on the neural network parameters $\boldsymbol{\theta}$ that we need to optimize.

When the position is visible, both distributions copy the same token and the KL is zero. The two cases combine as

$$\mathrm{KL}\Big(q(\mathbf{z}_s^{(i)} \mid \mathbf{z}_t,\mathbf{x}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{z}_s^{(i)} \mid \mathbf{z}_t,t)\Big) = M_t^{(i)}\rho_{s,t}C_i. \tag{86}$$

Now, take a grid $0 = t_0 < \cdots < t_K = 1$. Using the derived ρ, the loss can be expanded as follows:

$$\boxed{\mathcal{L}_K(\boldsymbol{\theta}) = \mathbb{E}\left[\frac{1}{L}\sum_{j=1}^{K}\frac{\alpha_{t_{j-1}}-\alpha_{t_j}}{1-\alpha_{t_j}}\sum_{i=1}^{L}M_{t_j}^{(i)}C_i(\boldsymbol{\theta};\mathbf{x},\mathbf{z}_{t_j},t_j)\right].} \tag{87}$$

For an interval of width Δt, the schedule difference is $\alpha_{t-\Delta t}-\alpha_t = -\dot{\alpha}_t\Delta t + o(\Delta t)$. Consequently (87) becomes the integral

$$\boxed{\mathcal{L}_{\text{mask}}(\boldsymbol{\theta}) = \int_0^1 \frac{-\dot{\alpha}_t}{1-\alpha_t}\mathbb{E}_{\mathbf{x}\sim p_d,\,\mathbf{z}_t\sim q(\cdot|\mathbf{x})}\left[\frac{1}{L}\sum_{i=1}^{L}M_t^{(i)}C_i\right]\mathrm{d}t.} \tag{88}$$

The reconstruction term tends to zero as the initial grid interval shrinks. This gives the continuous-time masked-diffusion objective. For the linear schedule, $-\dot{\alpha}_t/(1-\alpha_t) = 1/t$. Sampling t uniformly turns the integral into an expectation:

$$\boxed{\mathcal{L}_{\text{mask}}(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},t,\mathbf{z}_t}\left[\frac{1}{Lt}\sum_{i=1}^{L}M_t^{(i)}\sum_{k=1}^{V}[\mathbf{x}^{(i)}]_k\log[\hat{\mathbf{x}}_{\boldsymbol{\theta}}^{(i)}(\mathbf{z}_t,t)]_k\right].} \tag{89}$$

Up to the constant normalisation $1/L$, this is the loss in Prabhudesai et al., Section 2.2, where the masking ratio is called r instead of t. The factor $1/t$ follows from the reverse-process KL. Dropping it gives a different training objective, even if a sufficiently flexible predictor can have the same pointwise optimum.

11.6 The complete training recipe

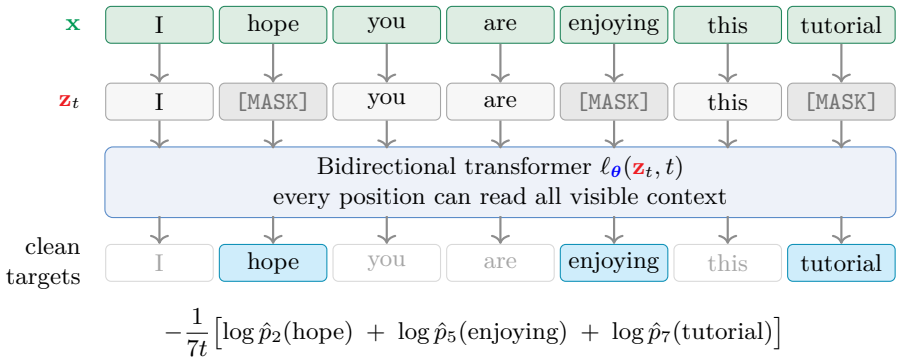

Figure 9: One training example. A fresh random mask hides positions $2, 5, 7$. The transformer predicts distributions at the original positions, and only the three highlighted clean targets contribute cross-entropy. Here $\hat{p}_i(k) = [\hat{\mathbf{x}}_{\boldsymbol{\theta}}^{(i)}(\mathbf{z}_t, t)]_k$. The target row shows labels used to score predictions, not sampled network outputs. The loss is divided by Lt, with $L = 7$.

For a minibatch, repeat the following operations for each sequence:

1. Draw clean text $\mathbf{x} \sim p_d$ and a time $t \sim U(0, 1)$.
2. Independently draw $M_t^{(i)} \sim \text{Bernoulli}(t)$ for each position. Set $\mathbf{z}_t^{(i)} = \mathbf{m}$ if $M_t^{(i)} = 1$, and $\mathbf{z}_t^{(i)} = \mathbf{x}^{(i)}$ otherwise.
3. Run the bidirectional transformer once on $\mathbf{z}_t$. Compute clean-token log-softmax values for the masked positions.
4. Sum their cross-entropies, multiply by $1/t$, divide by the sequence length L, and average over the minibatch.
5. Differentiate this loss with respect to $\boldsymbol{\theta}$ and update the parameters. Resample the masking ratio and pattern on the next visit.

Appendix C has a comparison of masked diffusion against auto-regressive models at small scale.

12 Block text diffusion

Section 10 generates a fixed-length sequence. To keep writing, generate one block, add it to the completed text, and generate the next block conditioned on everything written so far. This is the block organisation of Arriola et al. (2025). Here, we use exactly the uniform-token noising rule and sampler of Section 10. Uniform diffusion within blocks is also studied in BlockGen.

We keep the same notation: L is now the number of tokens in the *current block*, $\mathbf{x}$ is its clean target, and $\mathbf{z}_t$ is its noisy state. The context $\mathbf{c}$ contains the prompt and all completed blocks. It stays clean and fixed while the current block is denoised. Here, $t = 1$ is random tokens and $t = 0$ is clean text.

12.1 Architecture

We use an architecture inspired by DiffusionGemma: one transformer with shared weights $\boldsymbol{\theta}$, used in two ways.

- **Encode the clean context.** Run the transformer on $\mathbf{c}$ with *causal attention*. Each position sees only itself and earlier positions. Store the key and value vectors from every attention layer. This is the *KV cache*, which later queries can read.
- **Denoise the current block.** Each of its L positions can attend to the cached context and to all positions in the current block. This *bidirectional attention* lets neighbouring tokens help one another. The output is one vocabulary distribution per position:

$$\hat{\mathbf{x}}_{\boldsymbol{\theta}} = \mathrm{softmax}\big(\ell_{\boldsymbol{\theta}}(\mathbf{z}_t, t, \mathbf{c})\big) \in \mathbb{R}^{L \times V}. \tag{90}$$

Here $\ell_{\boldsymbol{\theta}}$ denotes the logits, as before. The implementation supplies the stored context keys and values instead of recomputing $\mathbf{c}$.

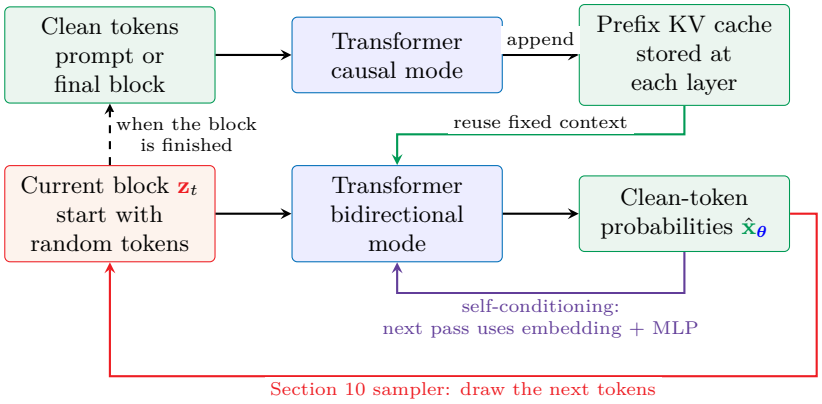

Figure 10: Generate one block, then add it to the context. The blue boxes use the same transformer weights with different attention masks. Denoising reads the fixed prefix cache. A completed block is encoded again in causal mode to append its cache entries. The inner feedback loop supplies the previous prediction as self-conditioning. The outer loop updates the discrete tokens.

12.2 Self-conditioning

A sampled token records one choice. The predicted probability vector also describes alternatives and confidence. *Self-conditioning* feeds a summary of that prediction into the next denoising pass. Write $\hat{\mathbf{x}}_{\text{previous}}$ for the previous pass's $L \times V$ probability table.

Use the purple embedding matrix $\mathbf{E} \in \mathbb{R}^{V \times D}$ from Section 9. For each position i, the vector $\mathbf{E}^\top \hat{\mathbf{x}}^{(i)}_{\text{previous}}$ is a probability-weighted average of token embeddings.

Before the next transformer pass, add a learned transformation of this soft guess to the current token's embedding:

$$\underbrace{\mathbf{E}^\top \mathbf{z}_t^{(i)}}_{\text{current token}} + \underbrace{\text{MLP}_{\boldsymbol{\theta}}\left(\mathbf{E}^\top \hat{\mathbf{x}}^{(i)}_{\text{previous}}\right)}_{\text{previous soft guess}}. \tag{91}$$

The MLP is a small trainable feedforward network mapping D numbers to D numbers, shared across positions. This is the feedback mechanism used in the DiffusionGemma implementation. Thus the prediction in (90) now also receives $\hat{\mathbf{x}}_{\text{previous}}$ as an input. For the first pass of each block, and for every causal context pass, omit the feedback term.

Why does it help? Even if the current sampled word is "spoon", the soft feedback can retain the earlier "cat or dog" guess. The model can refine that guess using the whole block instead of recovering it from the sampled tokens alone. This memory can make successive predictions more consistent. It does not make the parallel token draws jointly exact. The small-step warning in Section 10 still applies.

12.3 Block-wise training

Take a context $\mathbf{c}$ and the next L clean tokens $\mathbf{x}$ from the dataset. Draw $t \sim \mathcal{U}(0, 1)$ and corrupt *only* this target block using (53). With $\alpha_t = 1 - t$, each token is replaced by an independent uniform vocabulary draw with probability t.

Train with and without feedback. On half the examples, first predict the clean tokens with no feedback and *without gradients*. Detach those probabilities, use them as $\hat{\mathbf{x}}_{\text{previous}}$, and run the prediction again with (91). On the other examples, use zero feedback. Detaching stops differentiation through the preliminary prediction. The feedback MLP still learns in the final pass. This is the two-pass recipe described in NVIDIA's DiffusionGemma training guide.

Apply the same cross-entropy as Section 10.2 to the final prediction $\hat{\mathbf{x}}_{\boldsymbol{\theta}}$:

$$\mathcal{L}_{\text{CE}}(\boldsymbol{\theta}) = -\mathbb{E}\left[\frac{1}{L}\sum_{i=1}^{L}\sum_{k=1}^{V}[\mathbf{x}^{(i)}]_k \log[\hat{\mathbf{x}}^{(i)}_{\boldsymbol{\theta}}]_k\right]. \tag{92}$$

The expectation includes the data, noise, time, and feedback choice. We supervise every position in the target block, including tokens that were not replaced.

The context must exclude the target block's clean answers and all later text. During training the context comes from the dataset. During generation it contains our completed output.

12.4 Block-wise inference

Use the following loop, illustrated in Figure 10:

1. **Prepare the context.** Causally encode the prompt once and store its KV cache.
2. **Start a block.** Draw L independent uniform tokens, set $t = 1$, and clear the previous-prediction feedback.
3. **Refine the block.** Predict $\hat{\mathbf{x}}_{\boldsymbol{\theta}}$ from the current tokens, cached context, and feedback. Apply (66): at each position, keep the current token with probability $1 - \Delta t/t$, or draw from the predicted clean-token distribution with probability $\Delta t/t$. Save $\hat{\mathbf{x}}_{\boldsymbol{\theta}}$ as the feedback for the next pass and decrease t by Δt. Repeat until $t = 0$, using $0 < \Delta t \leq t$ and evaluating the final update at positive t.
4. **Commit and continue.** The resulting block is now fixed. If it contains the end-of-sequence token, stop there and discard its remaining positions. Otherwise run a *causal* pass on the completed block, with feedback off and the old cache available. Append its new keys and values to the cache, extend $\mathbf{c}$, and return to step 2.

The final causal pass is necessary. The denoising pass used a different attention mask, so its keys and values are not the required clean context cache. Self-conditioning needs no extra transformer pass during generation, because the preceding iteration already computed the prediction we feed back. Here, we kept Section 10's sampler and explicit time input. DiffusionGemma's published system adds its own entropy-based sampling and stopping rules.

13 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.

13.1 Approach 1: Noise into a separate diffusion head

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

$$p(\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N) = \prod_{\tau=1}^{N} p(\mathbf{x}^\tau | \mathbf{x}^1, \ldots, \mathbf{x}^{\tau-1}) \tag{93}$$

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

$$\mathbf{h}^\tau = f_{\boldsymbol{\theta}}(\mathbf{x}^0, \mathbf{x}^1, \ldots, \mathbf{x}^{\tau-1}) \tag{94}$$

and then generate the next patch via:

$$\mathbf{x}^\tau \sim p(\mathbf{x}^\tau | \mathbf{h}^\tau) \tag{95}$$

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:

$$\mathbf{z}_t^\tau = \alpha_t \mathbf{x}^\tau + \sigma_t \boldsymbol{\epsilon}_t \tag{96}$$

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:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}\left[\|\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\mathbf{z}_t^\tau, \lambda_t, \mathbf{h}^\tau) - \boldsymbol{\epsilon}_t\|_2^2\right] \tag{97}$$

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.

13.2 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

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

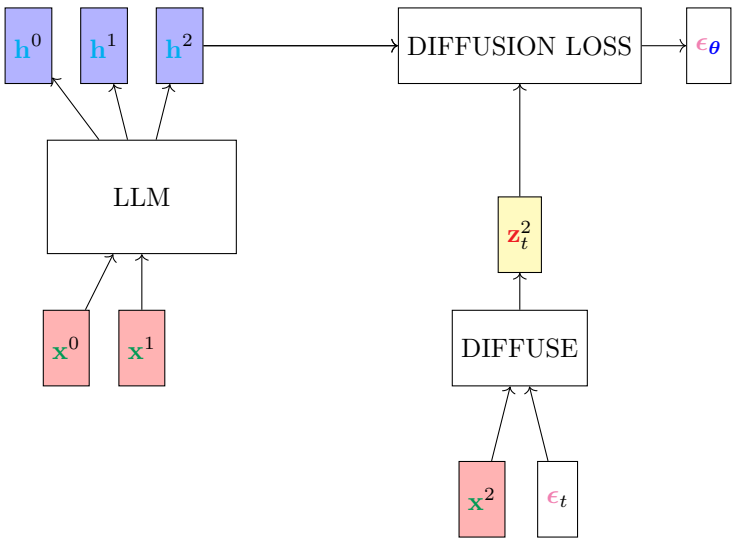

Figure 11: MAR-style architecture. The LLM consumes clean past patches $\mathbf{x}^0, \mathbf{x}^1$ and emits a hidden state $\mathbf{h}^2$. The forward diffusion (DIFFUSE) corrupts the target patch $\mathbf{x}^2$ into $\mathbf{z}_t^2$ using a fresh noise sample ϵ_t. A separate diffusion head $\epsilon_{\boldsymbol{\theta}}$ 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.

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:

$$\mathbf{z}_t^\tau = \alpha_t \mathbf{x}^\tau + \sigma_t \epsilon_t^\tau, \qquad \epsilon_t^\tau \sim \mathcal{N}(0, \mathbf{I}), \qquad \tau = 1, \ldots, N. \tag{98}$$

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:

$$\mathcal{L}_{\text{diff}}(\boldsymbol{\theta}) = \mathbb{E}\left[\sum_{\tau=1}^{N} \| \epsilon_{\boldsymbol{\theta}}^{\tau}(\mathbf{z}_t^1, \ldots, \mathbf{z}_t^N, t, \text{prefix}) - \epsilon_t^{\tau} \|_2^2\right], \tag{99}$$

$$\mathcal{L}_{\text{Transfusion}}(\boldsymbol{\theta}) = \mathcal{L}_{\text{LM}}(\boldsymbol{\theta}) + \lambda_{\text{img}} \mathcal{L}_{\text{diff}}(\boldsymbol{\theta}). \tag{100}$$

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.

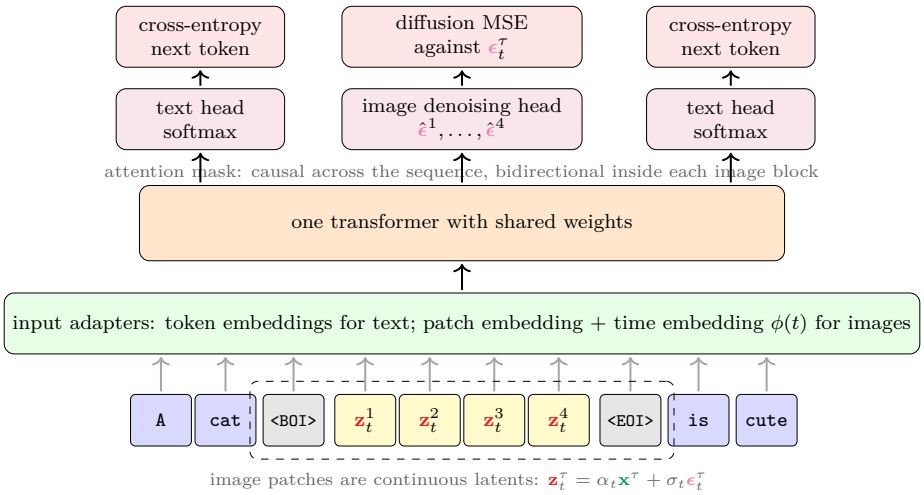

Figure 12: Transfusion-style training picture. Text tokens (blue), marker tokens (gray), and noisy continuous image patches (yellow) are placed into one sequence. A small set of modality-specific adapters maps them into the shared transformer space. The same transformer then feeds two kinds of output 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.

13.3 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 $1, \ldots, \tau - 1$. 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 derived earlier, 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:

$$\begin{aligned} q(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{x}) &= \frac{q(\mathbf{z}_{t+1}|\mathbf{z}_t, \mathbf{x})\, q(\mathbf{z}_t|\mathbf{x})}{q(\mathbf{z}_{t+1}|\mathbf{x})} \\ &\propto q(\mathbf{z}_{t+1}|\mathbf{z}_t)\, q(\mathbf{z}_t|\mathbf{x}). \end{aligned} \tag{101}$$

The prior follows from the way we diffuse:

$$q(\mathbf{z}_t|\mathbf{x}) = \mathcal{N}(\mathbf{z}_t; \alpha_t \mathbf{x}, \sigma_t^2 \mathbf{I}). \tag{102}$$

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

$$q(\mathbf{z}_{t+1}|\mathbf{z}_t) = \mathcal{N}\left(\mathbf{z}_{t+1}; \frac{\alpha_{t+1}}{\alpha_t}\mathbf{z}_t,\ \left(\sigma_{t+1}^2 - \frac{\alpha_{t+1}^2}{\alpha_t^2}\sigma_t^2\right)\mathbf{I}\right). \tag{103}$$

Multiplying two Gaussians gives another Gaussian. Define

$$A = \frac{\alpha_{t+1}}{\alpha_t}, \qquad B = \sigma_{t+1}^2 - A^2\sigma_t^2. \tag{104}$$

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

$$q(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{x}) \propto \exp\left\{-\frac{1}{2}\left[\frac{1}{B}\|\mathbf{z}_{t+1} - A\mathbf{z}_t\|_2^2 + \frac{1}{\sigma_t^2}\|\mathbf{z}_t - \alpha_t\mathbf{x}\|_2^2\right]\right\}. \tag{105}$$

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

$$\begin{aligned} \Sigma^{-1} &= \left(\frac{A^2}{B} + \frac{1}{\sigma_t^2}\right)\mathbf{I} \\ &= \frac{A^2\sigma_t^2 + B}{B\sigma_t^2}\mathbf{I} = \frac{\sigma_{t+1}^2}{B\sigma_t^2}\mathbf{I}, \end{aligned} \tag{106}$$

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

$$\begin{aligned} \Sigma &= \frac{B\sigma_t^2}{\sigma_{t+1}^2}\mathbf{I} \\ &= \left(\sigma_{t+1}^2 - \frac{\alpha_{t+1}^2}{\alpha_t^2}\sigma_t^2\right)\frac{\sigma_t^2}{\sigma_{t+1}^2}\mathbf{I}. \end{aligned} \tag{107}$$

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

$$\begin{aligned}\boldsymbol{\mu} &= \Sigma\left(\frac{A}{B}\mathbf{z}_{t+1} + \frac{\alpha_t}{\sigma_t^2}\mathbf{x}\right) \\ &= \frac{A\sigma_t^2}{\sigma_{t+1}^2}\mathbf{z}_{t+1} + \frac{B\alpha_t}{\sigma_{t+1}^2}\mathbf{x}. \end{aligned} \tag{108}$$

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

$$q(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{x}) = \mathcal{N}\left(\mathbf{z}_t; \boldsymbol{\mu}_{t|t+1}, \Sigma_{t|t+1}\right), \tag{109}$$

$$\boldsymbol{\mu}_{t|t+1} = \frac{\alpha_{t+1}}{\alpha_t}\frac{\sigma_t^2}{\sigma_{t+1}^2}\mathbf{z}_{t+1} + \left(\sigma_{t+1}^2 - \frac{\alpha_{t+1}^2}{\alpha_t^2}\sigma_t^2\right)\frac{\alpha_t}{\sigma_{t+1}^2}\mathbf{x}, \tag{110}$$

$$\Sigma_{t|t+1} = \left(\sigma_{t+1}^2 - \frac{\alpha_{t+1}^2}{\alpha_t^2}\sigma_t^2\right)\frac{\sigma_t^2}{\sigma_{t+1}^2}\mathbf{I}. \tag{111}$$

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

$$\boldsymbol{\mu}_{t|t+1} = \frac{\alpha_t}{\alpha_{t+1}}e^{\lambda_{t+1}-\lambda_t}\mathbf{z}_{t+1} + \left(1 - e^{\lambda_{t+1}-\lambda_t}\right)\alpha_t\mathbf{x}, \tag{112}$$

$$\Sigma_{t|t+1} = \left(1 - e^{\lambda_{t+1}-\lambda_t}\right)\sigma_t^2\mathbf{I}. \tag{113}$$

Finally, setting

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

gives the compact sampler form:

$$q(\mathbf{z}_t|\mathbf{z}_{t+1}, \mathbf{x}) = \mathcal{N}\left(\mathbf{z}_t;\, \alpha_t\left[\frac{1-c}{\alpha_{t+1}}\mathbf{z}_{t+1} + c\mathbf{x}\right], c\,\sigma_t^2\mathbf{I}\right). \tag{114}$$

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:

$$\dot{\mathbf{z}}_t = \mathbf{v}(\mathbf{z}, t) \tag{115}$$

where

$$\mathbf{v}(\mathbf{z}, t) = \mathbb{E}[\dot{\mathbf{z}}_t | \mathbf{z}_t = \mathbf{z}] = \dot{\alpha}_t \mathbb{E}[\mathbf{x} | \mathbf{z}_t = \mathbf{z}] + \dot{\sigma}_t \mathbb{E}[\boldsymbol{\epsilon} | \mathbf{z}_t = \mathbf{z}]. \tag{116}$$

The proof consists of two steps.

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

$$\partial_t p_t(\mathbf{z}) + \nabla_{\mathbf{z}} \cdot (\mathbf{v}(\mathbf{z}, t) p_t(\mathbf{z})) = 0. \tag{117}$$

- 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:

$$\dot{\mathbf{z}}_t = \mathbf{v}(\mathbf{z}, t). \tag{118}$$

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

$$\hat{p}_t(\mathbf{k}) = \mathbb{E}[e^{i\mathbf{k} \cdot \mathbf{z}_t}] \tag{119}$$

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

$$\partial_t \hat{p}_t(\mathbf{k}) = i\mathbf{k} \cdot \mathbb{E}[\dot{\mathbf{z}}_t e^{i\mathbf{k}\cdot\mathbf{z}_t}] \tag{120}$$

$$= i\mathbf{k} \cdot \mathbb{E}_{\mathbf{z}\sim p_t}[\mathbb{E}[\dot{\mathbf{z}}_t e^{i\mathbf{k}\cdot\mathbf{z}_t}|\mathbf{z}_t = \mathbf{z}]] \tag{121}$$

$$= i\mathbf{k} \cdot \mathbb{E}_{\mathbf{z}\sim p_t}[\mathbb{E}[(\dot{\alpha}_t \mathbf{z}_* + \dot{\sigma}_t \epsilon) e^{i\mathbf{k}\cdot\mathbf{z}_t}|\mathbf{z}_t = \mathbf{z}]] \tag{122}$$

$$= i\mathbf{k} \cdot \mathbb{E}_{\mathbf{z}\sim p_t}[\mathbb{E}[(\dot{\alpha}_t \mathbf{z}_* + \dot{\sigma}_t \epsilon)|\mathbf{z}_t = \mathbf{z}] e^{i\mathbf{k}\cdot\mathbf{z}}] \tag{123}$$

$$= i\mathbf{k} \cdot \mathbb{E}_{\mathbf{z}\sim p_t}[\mathbf{v}(\mathbf{z}, t) e^{i\mathbf{k}\cdot\mathbf{z}}] \tag{124}$$

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. (124) reads

$$\partial_t \int_{\mathbb{R}^d} e^{i\mathbf{k}\cdot\mathbf{z}} p_t(\mathbf{z}) d\mathbf{z} = i\mathbf{k} \cdot \int_{\mathbb{R}^d} \mathbf{v}(\mathbf{z}, t) e^{i\mathbf{k}\cdot\mathbf{z}} p_t(\mathbf{z}) d\mathbf{z} \tag{125}$$

from which we deduce

$$\int_{\mathbb{R}^d} e^{i\mathbf{k}\cdot\mathbf{z}} \partial_t p_t(\mathbf{z}) d\mathbf{z} = \int_{\mathbb{R}^d} \mathbf{v}(\mathbf{z}, t) \cdot \nabla_{\mathbf{z}} [e^{i\mathbf{k}\cdot\mathbf{z}}] p_t(\mathbf{z}) d\mathbf{z} \tag{126}$$

$$= -\int_{\mathbb{R}^d} \nabla_{\mathbf{z}} \cdot [\mathbf{v}(\mathbf{z}, t) p_t(\mathbf{z})] e^{i\mathbf{k}\cdot\mathbf{z}} d\mathbf{z} \tag{127}$$

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 $\mathbf{z}$):

$$\partial_t p_t(\mathbf{z}) + \nabla_{\mathbf{z}} \cdot (\mathbf{v}(\mathbf{z}, t) p_t(\mathbf{z})) = 0. \tag{128}$$

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

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

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.

Appendix C: A small experiment of masked diffusion versus AR

Section 11 derived the masked-diffusion loss (89) and noted that it is the loss used by Prabhudesai et al. (2025), who claim that masked diffusion beats autoregressive (AR) language models when data, rather than compute, is the bottleneck. This appendix reports a small reproduction of that study, run on a MacBook M3 with the PyTorch MPS backend. The short summary: the *mechanism* reproduces very clearly (AR overfits after a handful of passes over the data, masked diffusion keeps improving for roughly ten times longer), but at this scale the headline does not (diffusion catches up with AR, it does not overtake it).

C.1 The experiment

The paper trains both model families on a fixed budget of U unique tokens, repeated for many epochs, and asks which one reaches the lower validation loss. I have an issue with this because the two losses are different, one is likelihood and the other is a variational upper-bound. But let's entertain this for the time being, and we'll look at other things like samples and other simple downstream benchmarks.

	Paper	This appendix
Data	C4 (English), GPT-2 BPE	same, vocabulary capped at 8,192
Unique tokens U	25M, 50M, 100M	250k, 500k, 1M
Sequence length L	2048	128
Parameters	7M to 2.5B	1.6M ("xs") and 5.1M ("s")
Epochs	up to 800	1, 2, 4, 8, 16, 64, 256
Batch size	256	64

Model. One transformer is used for both families, as in the paper: RoPE, SwiGLU, RMSNorm, no biases, tied input and output embeddings. The "xs" model has width 128, 3 layers and 4 heads; "s" has width 256, 4 layers and 4 heads. The only differences between the two families are the attention mask and the target.

- **AR**: causal attention and next-token cross-entropy.
- **Masked diffusion**: bidirectional attention and the loss (89), following the recipe of Section 11.6 exactly: draw $t \sim \mathcal{U}(0,1)$ per sequence, mask each token independently with probability t, and weight the cross-entropy on the masked positions by $1/t$.

Both are trained with AdamW ($\beta = (0.9, 0.95)$, weight decay 0.1, gradient clipping at 1), 1% warm-up and cosine decay. Each (objective, U, epochs, size) cell is a separate run trained from scratch, 84 runs in all, plus those of the regularisation study below: about 7 hours on the laptop. Tokens outside

the 8,192 most frequent are mapped to a single unknown token (about 14% of validation tokens), because with so little data most of the GPT-2 vocabulary never occurs and the output softmax would otherwise dominate the cost of these tiny models.

What "validation loss" means. Every number below is the average negative log-likelihood per token, in nats, on 128k held-out tokens from the C4 validation split. For AR this is exact: $-\frac{1}{L}\sum_i \log p_{\boldsymbol{\theta}}(x^{(i)} \mid x^{(<i)})$. A masked-diffusion model has no closed-form likelihood, so we report (89) evaluated on the held-out text with four fixed random maskings per sequence. By the KL argument of Section 11.5 this is an *upper bound* on the negative log-likelihood, so the comparison is tilted slightly against diffusion; it is the same comparison the paper makes. A loss of 4.8 is a perplexity of about 120. "Final" is the loss at the end of a run; "early-stopped" is the lowest loss seen at any evaluation point during the run.

C.2 Results

AR overfits, diffusion keeps going. Figure 13 shows the final validation loss against the number of passes over the same data. AR reaches its best loss at 8 to 16 epochs in every panel and then overfits catastrophically: by 64 to 256 epochs its validation loss is between 7 and 18 nats, far worse than after a single epoch. Masked diffusion starts well behind, sits on a plateau near 6.3 nats for the first few epochs, and is still improving at 64 epochs. The two curves cross at around 16 epochs.

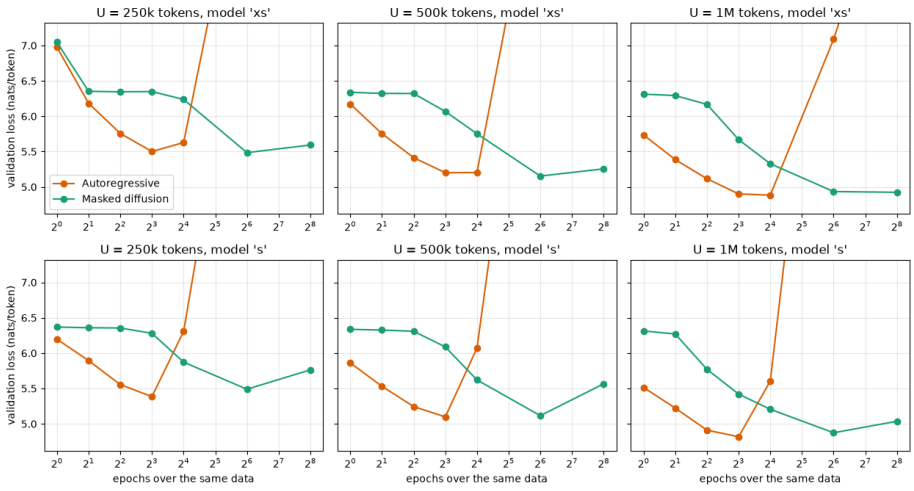

Figure 13: Final validation loss against epochs over the same data, for three data budgets (columns) and two model sizes (rows). The vertical axis is clipped; the AR curves continue up to 7–18 nats.

Figure 14 shows diffusion needs more epochs, and Figure 15 shows the validation loss *during* training. In the AR panel every run that goes past roughly 5×10^6 training tokens (five passes) turns upwards. In the diffusion panel the runs of different lengths lie almost on top of one another and keep falling for another order of magnitude in training tokens. This is the picture in the paper's Figure 4.

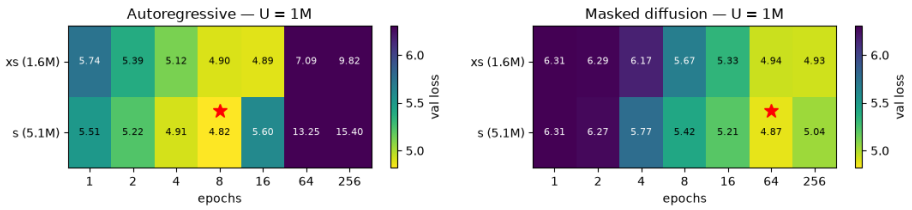

Figure 14: Final validation loss over model size and epochs at $U = 1$M. The star marks the best cell: 8 epochs for AR, 64 for diffusion.

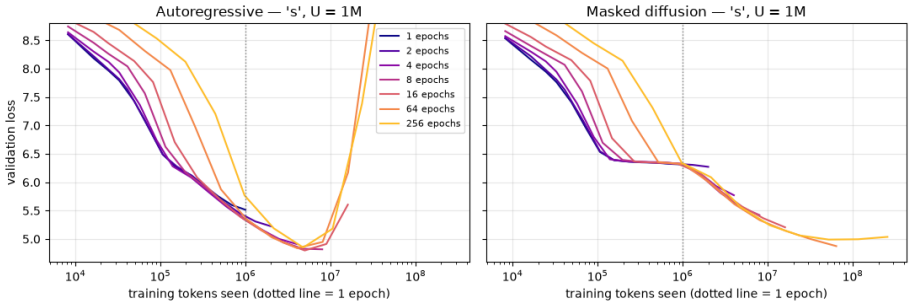

Figure 15: Validation loss during training for each epoch budget ("s" model, $U = 1$M). The dotted line marks one pass over the data.

In this small experiment, diffusion does not win, but it is close. The best result of each family is in the table below. With the larger model AR is ahead at every data budget, by 0.02 to 0.10 nats. Diffusion is ahead only with the smallest model at 250k and 500k tokens, by 0.02 to 0.05 nats, and not if AR is allowed early stopping. Each cell is a single seed, so differences of a few hundredths of a nat are within noise.

U	Model	Best AR (epochs)	AR, early-stopped	Best diffusion (epochs)
250k	xs	5.501 (8)	5.473	5.485 (64)
250k	s	5.385 (8)	5.383	5.490 (64)
500k	xs	5.202 (8)	5.135	5.155 (64)
500k	s	5.098 (8)	5.053	5.116 (64)
1M	xs	4.885 (16)	4.829	4.925 (256)
1M	s	4.818 (8)	4.797	4.874 (64)

Two further differences from the paper are visible in Figure 13. First, diffusion *also* overfits here: in five of the six panels its loss turns back up between 64 and 256 epochs (for example from 5.116 to 5.566 at 500k tokens with the "s" model), whereas the paper reports no such overfitting. With only a few thousand training sequences even the masked objective eventually memorises.

Compute frontier and break-even compute. The paper's headline figure plots, for each data budget, the best validation loss reachable within a given training compute, counted as $6 \times$ parameters $\times$ training tokens. Figure 16 is the same plot for our grid: each faint point is one run, and each line is the best loss achieved by any model size and epoch count at or below that compute. The orange points far above the rest are the overfit AR runs. AR's frontier is below diffusion's at low and medium compute by up to 0.6 nats, and the two meet only at the right-hand edge.

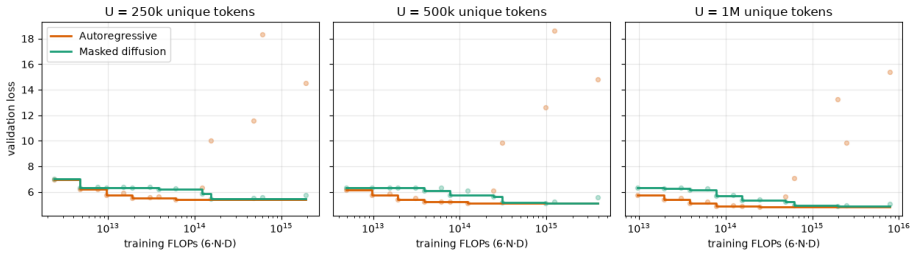

Figure 16: Best validation loss against training compute for each data budget. Lines are the frontier over all model sizes and epoch counts; faint points are individual runs.

The half-life of data reuse. The paper summarises the effect with the data-constrained scaling law of Muennighoff et al., in which E epochs over U unique tokens are worth an effective amount of fresh data

$$D' = U + U\, R_D^* \left(1 - e^{-(E-1)/R_D^*}\right), \qquad \mathcal{L} = \frac{B}{D'^{\,\beta}} + E_0. \tag{129}$$

The constant R_D^* is the number of epochs over which repetition keeps paying off. Fitting this law to the early-stopped losses of each model size (Figure 17) gives

	R_D^*, AR	R_D^*, diffusion	ratio
"xs" model	4.8	92.8	19
"s" model	3.7	57.5	16
Paper	31	494	16

The absolute values are an order of magnitude smaller than the paper's, as one would expect with a hundred times less data, but the *ratio* between the two families is the same. The AR fits are good ($R^2 \approx 0.99$). The diffusion fits are

weaker ($R^2 \approx 0.90$) and pushed E_0 to its lower bound, so the diffusion half-lives should be read as rough.

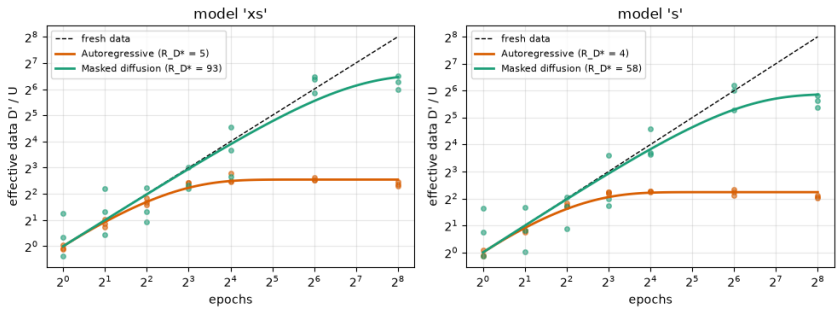

Figure 17: Effective data as a multiple of the unique data, against epochs. Lines are the fitted law, points are individual runs mapped back through the fit, and the dashed line is "every epoch is as good as fresh data". AR saturates at about four times its unique data; diffusion follows the fresh-data line for far longer.

Regularising AR does not close the gap. The paper argues that diffusion's advantage is not ordinary regularisation. Figure 18 repeats their test: an AR model ("s", 500k tokens, 64 epochs) with attention dropout, or with a random fraction of input tokens hidden from attention in every layer. Both delay the overfitting, and neither prevents it. The early-stopped loss does not improve (5.05 to 5.16, against 5.05 without regularisation), and after 64 epochs every variant is between 6.2 and 9.5 nats, while diffusion is at 5.12 and still falling.

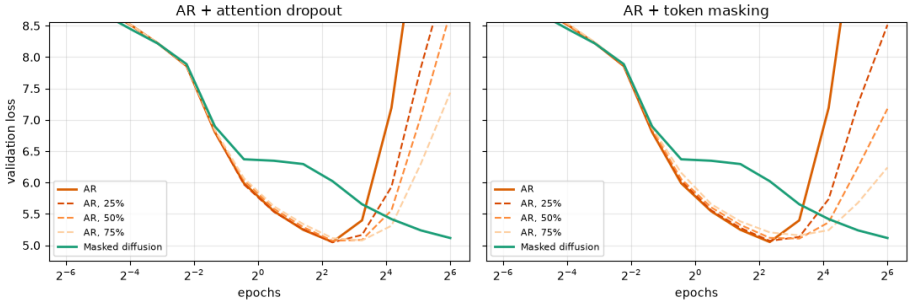

Figure 18: Validation loss during training for AR with attention dropout (left) and input-token masking (right) at rates of 25%, 50% and 75%, against plain AR and masked diffusion.

Other tasks. The best model of each family at $U = 1$M (AR after 8 epochs, diffusion after 64) was evaluated on text from other distributions, on two multiple-choice benchmarks from the paper, and on BLiMP grammatical minimal pairs.

Answers are ranked by likelihood: exact for AR, and the bound (89) restricted to the answer tokens, averaged over 16 maskings, for diffusion.

Task	AR	Diffusion	Chance
C4 validation loss ↓	4.818	4.875	–
WikiText-2 loss ↓	5.843	5.817	–
TinyStories loss ↓	4.497	4.586	–
SciQ accuracy ↑	34.4%	33.8%	25%
ARC-Easy accuracy ↑	22.4%	22.2%	25%
BLiMP accuracy ↑	57.6%	59.5%	50%

There is no consistent winner. AR is ahead on two of the losses; diffusion is marginally ahead on WikiText-2 and BLiMP. The two question-answering tasks are ties, with ARC-Easy at chance for both. The one suggestive pattern is that diffusion, which is 0.06 nats behind on the training distribution, is 0.03 ahead on WikiText-2, the corpus least like the training data. That is the direction the paper reports, but TinyStories goes the other way and a single seed cannot settle it.

Samples. Both families were sampled from the best checkpoint at $U = 1$M (AR after 8 epochs, diffusion after 64), at temperature one with no truncation. AR samples left to right. Diffusion starts from $L = 128$ masks and reveals them in a random order, re-running the network between reveals; this is the sampler (82) with the number of tokens revealed per step fixed rather than random. The samples below are the first one drawn in each setting, unedited. Line breaks are removed, ‖ marks a document boundary, and *[unk]* is a token outside the capped vocabulary (it is never sampled, but can appear in text we supply).

AR, unconditional.
brought off as the year in Week passed someone to tell our eyes over the community feeling as I wanted to mention that I took intended. To start alone, Jng untilbras by the right rest of the world, thank you communication to the garden for about my office name and will have the best lady. The "a spa summit" Can you know about "As a relaxing workforce for 4 more than pro), I love their potential for the car, I just made the Dinecization ofog kitchen accessories and showerists changed the valley from yours. You may go any way sereph started making some of the other

Diffusion, unconditional, one token per step (128 network calls).
now the culture were afraid to learn areas that are of weed. In a particular event included the proficiency of injury go in Annclaysils of differently and instance/ facility once the face gets wet tables with for more variation for its interaction. The knowledge of art and fruit of Lesniasis held a backma hard to protect which creatures, de la pan that had to take 3 hours to be being and scared as words (for words rather). sleeping however, they could suit a number of death. No be way with girl shifts." After Order in the country or the pop of essays, transportation may such as God asked to

Diffusion, unconditional, 16 steps (8 tokens per step).
that felt like I will take mind in in your life but just read a common, I mean that the women stay everything out of the best valuable but in my own all of this for click back, you canm not well, just, opinions and get down Thanks is particular. This seems for long as I essential to make things what now I find

sure to get myself and do for those who can live within.it is their true role and how to called inland, home industry sites and those looking ways Will myself worry more heavy land with the rest of America If New York we have elected questions in New York if the initial ground

AR, continuing a prompt (prompt in bold, 48 tokens in total).
The best way to learn a new language is *attended. The theme one area has an supporting reference pur box for tough sense as it does. Learn more to allow Members and Services, agencies and ESG product practitioners are available about the*

Diffusion, continuing the same prompt.
The best way to learn a new language is *corporate informed. What your needs, work place, play in each other ways? Who have the Right to be some of the negative amount of risks in order. And it can be taken. Very% of time. Very important, they are an extremely slow rate and! We are going to learn about the way. Finally, the South African America and all others in the world. It is incredibly dangerous, in mind. I well mean things to check out how many guides to a source theory. Even absolutelyProvals of their reputation is more and supportive, for all questions about these products*

Diffusion, infilling between a given start and end (both in bold).
We went to the beach because *we ran away the time they always have time to get its welcome, or get their respect. You and interested in any of these points. We thought we facilitate you in internet, now there are plenty of big but my mis Post was getting to write my the club had a great, cool/ter program for a client HERE! We look for us for a proud enjoyed with most experienced banking. By Join that the corporate site he aims to become a quick creamy to add 0/2 full-to-share north right, we will admit that. As we go,* ***and then we[unk] home.***

At a perplexity near 120 both models produce locally fluent web-style English with no coherence beyond a phrase or two, and the one-token-per-step diffusion samples are hard to tell apart from the AR ones by eye. Neither model follows the meaning of the prompt. The 16-step sample is visibly worse, with more broken grammar and stranded function words. This is the "I are" problem of Section 10 in practice: eight tokens are drawn independently at each step, and each draw cannot see the others.

Latency. I timed the generation of one 128-token sample on the laptop (average of three, after a warm-up call):

Sampler	Calls	s / sample	ms / call	tokens / s
AR (no key–value cache)	128	1.68	13	76
Diffusion, one token per step	128	1.03	8	124
Diffusion, 16 steps	16	0.13	8	985

Diffusion is faster in both settings: by a factor of 1.6 with the same number of network calls, and by a factor of 13 with 16 steps. For a model this small the arithmetic is negligible and the time is dominated by per-call overhead, so latency is set by the number of sequential network calls, which diffusion lets us choose. The price of fewer calls is the drop in sample quality seen above. AR is also slower *per call* here, most likely because its input grows by one token at every step while the diffusion input keeps a fixed shape; this was not profiled.

This is a small-scale illustration of Figure 2, with two caveats. The AR sampler recomputes the whole prefix at every step; with a key–value cache it would do far less arithmetic per step and would probably close most of the gap to diffusion at one token per step, although it would still need 128 sequential calls. And on large models, where arithmetic rather than call overhead dominates, the comparison has to be redone.

Training throughput, from a speed check (batch of 64 sequences of 128 tokens), tells the other half of the story:

	Steps per second		Time to best model at $U = 1$M	
Model	AR	Diffusion	AR	Diffusion
"xs" (1.6M)	24.7	14.5	–	–
"s" (5.1M)	11.7	10.5	1.5 min (8 epochs)	10.4 min (64 epochs)

A training step costs about the same for the two objectives with the larger model, but diffusion needs eight times as many epochs to reach its best loss, so about seven times the training time. Scoring text is also more expensive for diffusion: AR gives an exact likelihood in one pass, while the diffusion bound needs one pass per random masking (16 in the task evaluations above). In short, at this scale diffusion buys cheaper generation at the cost of more expensive training and evaluation.

C.3 What to take from this

Claim in the paper	At laptop scale
AR overfits after a few epochs	Yes, strongly
Diffusion keeps improving with repetition	Yes, up to about 64 epochs
Diffusion's data-reuse half-life is far longer	Yes, by the same factor of 16–19
Dropout and token masking do not rescue AR	Yes
AR is better when compute is the constraint	Yes: 5.51 vs 6.31 after one epoch
Diffusion's best loss beats AR's best	No: AR ahead by 0.02–0.10 nats
Diffusion is better downstream	No: a draw

The masked objective of Section 11 really does behave like a strong form of data augmentation: every visit to a sequence poses a different prediction problem, so the model can revisit the same text many more times before it memorises it. What the experiment does not show is that this is enough to come out ahead. That is not a contradiction of the paper. Its own margin at 100M tokens is small (3.55 against 3.71) and is reached only with much larger models trained for about 500 epochs; the crossover moves to more compute as the data grows, and a 5M parameter model on 1M tokens is plausibly on the wrong side of it.

Several caveats apply. The models are about a thousand times smaller than the paper's and see 128-token contexts. The diffusion loss is a bound and the AR loss is exact.