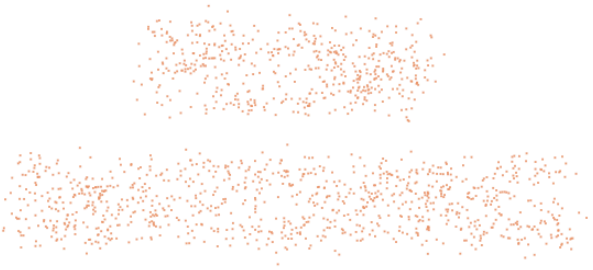
From Noise to Images: Flow Matching Explained



See how a model learns to turn random noise into an image, then train a small digit generator in your browser.

AUTHOR

Aritra Roy Gosthipaty

AFFILIATION

Hugging Face

PUBLISHED

Sep. 13, 2026

Table of Contents

Introduction

How do you turn random noise into an image?

Consider starting with an image filled with completely random pixels and somehow moving those pixels, little by little, until they form a handwritten digit, a face, or a photograph. Generative modeling is, in many ways, about learning how to perform such a transformation.

Given a dataset, our goal is to learn its underlying structure well enough to produce new objects that look as though they could have belonged to the dataset, without simply copying the examples we have already seen.

There are several approaches to generative modeling. In this post, we will focus on flow models and build up the ideas behind them from the ground up.

> 💡 Note
>
> To make the ideas easier to visualize, we will focus on image generation. The same concepts extend to many other modalities.

From noise to images

Imagine a dataset containing thousands of handwritten digits.

An image can be thought of as a point in a very high-dimensional space. For example, a grayscale 28×28 image can be represented by 784 numbers, and therefore corresponds to a point in $\mathbb{R}^{784}$.

Our dataset gives us many such points. But these points are not scattered uniformly throughout the space. Images that look like handwritten digits occupy some regions much more densely than others.

We can describe this using a probability distribution.

A probability distribution tells us how probability is spread over the space of possible images. Regions containing plausible handwritten digits should have high probability, while regions containing meaningless images should have very little probability.

We call the distribution underlying our dataset

$$p_{\text{data}}(x)$$

We do not actually know p_{data} explicitly. We only have samples from it (the images in our dataset). If we somehow learned this distribution, generation would become conceptually simple:

$$x \sim p_{\text{data}}$$

Draw a new sample, and we get a new image.

The problem is that sampling directly from this complicated, high-dimensional distribution is difficult. We choose a simple distribution that we already know how to sample from, usually a standard Gaussian,

$$z \sim \mathcal{N}(0, I)$$

The central idea is then to learn a transformation that takes samples from this simple noise distribution and transport them into the complicated data distribution:

$$p_{\text{noise}} \quad \longrightarrow \quad p_{\text{data}}$$

Once we know how to perform this transformation, generation becomes:

1. sample some noise
2. transform it
3. obtain an image

> 🤔 Food for thought
>
> The interesting question is therefore no longer how to generate an image but rather how do we continuously move an entire probability distribution from noise to data?

Optional: what does Gaussian noise mean?

What to train?

Choose a training image z and an independent noise array ε (pronounced “epsilon”) of the same shape. Now choose a number t between 0 and 1 and blend the two:

$$x_t = (1 - t)\varepsilon + tz$$

At $t = 0$, we have only noise.

$$x_0 = \varepsilon$$

At $t = 1$, we have the image.

$$x_1 = z$$

Halfway through, x_t is an equal mixture of both. We call t time, but it is simply a progress value along this path. As t moves from 0 to 1, x_t traces a straight-line trajectory from the noise sample ε to the image z. Because this trajectory is a straight line, its velocity is constant:

$$\text{target velocity} = \frac{d}{dt}x_t = \frac{d}{dt}\big[tz + (1 - t)\varepsilon\big] = z - \varepsilon$$

If we can ask the model to learn the velocity, we can go from noise to our data point easily!

A changing image, a constant training target

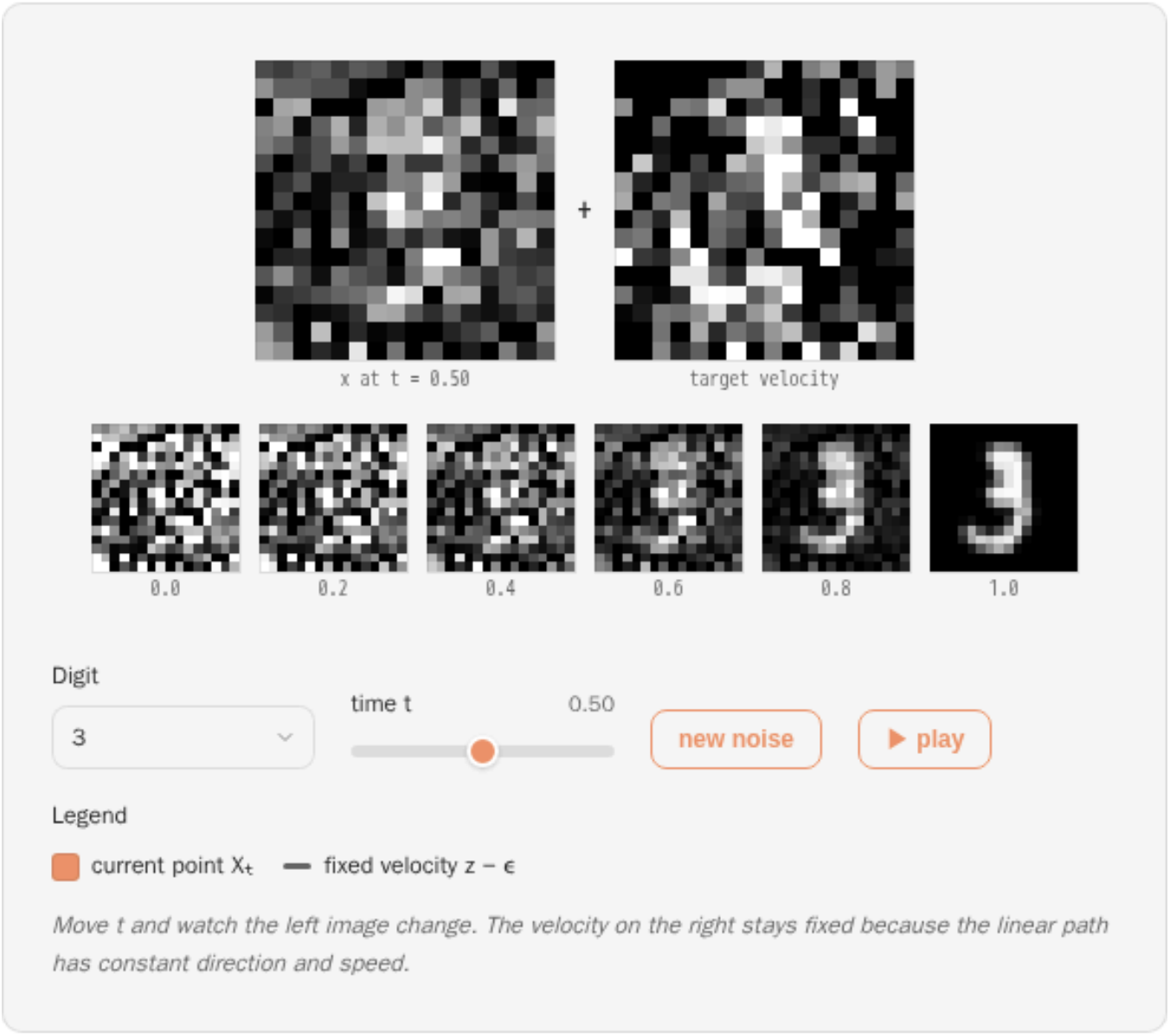


Figure 1 · Move time from 0 to 1. The noisy image changes, but the velocity on the right stays fixed. New noise creates a different path and target.

This gives us everything needed to construct a training example:

Give the model	Ask it to predict
The blended image x_t and time t	The velocity $z - \varepsilon$

The model sees neither the clean image nor the original noise separately. We use them to construct the blended image and its velocity target, then ask the model to predict that velocity from x_t and t alone.

The animation above uses a known image to construct a training path. During generation, however, we will have only noise and the velocity predicted by the trained model.

Optional: watch a whole noise cloud move toward one image

What to infer?

Suppose we already have a model that can predict velocity. How would we use it to generate an image?

Imagine a map of wind. At every location, an arrow tells you both the direction and speed in which something should move. A collection of such arrows is called a vector field.

Our model plays exactly this role. Given the current position and time, it predicts the velocity:

```
velocity = model(x, t)
```

Once we know the velocity, we can move a small amount in that direction. For a small time step h,

$$x_{t+h} \approx x_t + h, \mathrm{model}(x_t, t).$$

This procedure is called Euler's method.

After taking one step, we ask the model for another velocity. The answer may be different because both our position and the time have changed.

How many steps does it take to follow a curve?

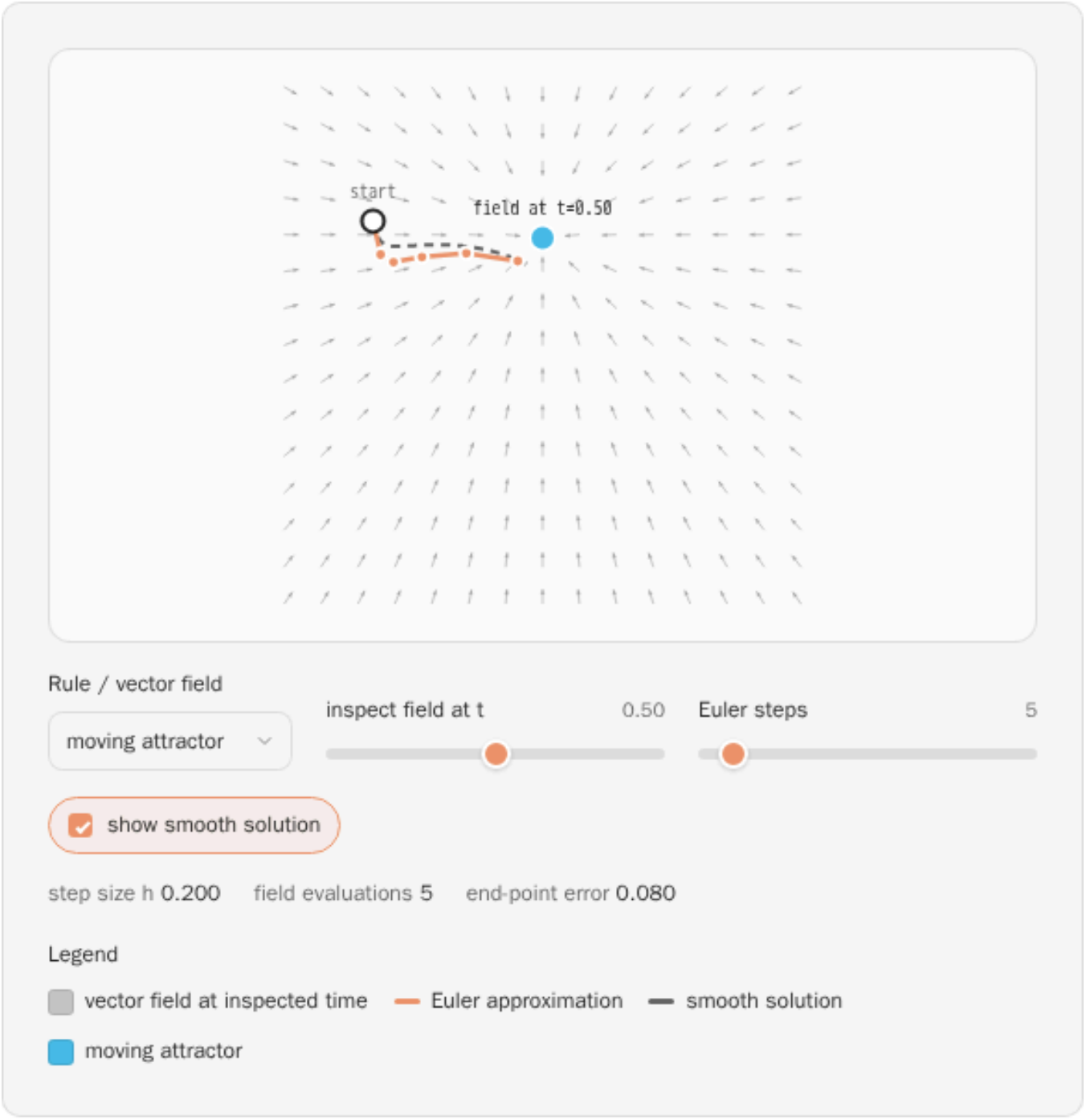


Figure 2 · Compare 4 Euler steps with 40. Smaller steps usually follow the smooth reference more closely, but require more evaluations of the field.

With a trained model, generation becomes a small loop:

```python
def generate(model, shape, steps=40):
    x = torch.randn(shape)          # start with Gaussian noise
    h = 1.0 / steps
    for i in range(steps):
        t = i * h
        x = x + h * model(x, t)      # predict a velocity, then move
    return x
```

We begin with fresh Gaussian noise and repeatedly follow the velocity predicted by the model. Each run starts from a different noise sample, so the same trained model can produce different images. The model itself does not change during generation.

We now know how to generate an image if we have the right vector field. The remaining question is how to train a model to produce those arrows.

Optional: ODEs, trajectories, and flows

Train the velocity field

We already know how to construct a training example.

Take an image z, pair it with random noise ε, choose a random time t, and interpolate between the two. This gives us a point somewhere along the path from noise to data x_t.

More importantly, because we created the path ourselves (the straight line), we also know the velocity that should take us along it.

So training is simply:

1. Create a random point somewhere between noise and data, x_t.
2. Ask the model which direction it would move from there.
3. Compare its prediction with the velocity we already know ($z - \varepsilon$).
4. Update the model and repeat.

```python
for z in data_loader:
    eps = torch.randn_like(z)
    t = torch.rand(z.shape[0], 1, device=z.device)

    x_t = (1 - t) * eps + t * z
    target = z - eps

    prediction = model(x_t, t)
    loss = ((prediction - target) ** 2).mean()

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Each training step shows the model only a small piece of the overall problem. By repeating this across many images, noise samples, and times, the model gradually learns what the velocity should look like throughout the space.

> 💡 Note
>
> training does not require following the entire trajectory. We can jump directly to any time t, construct the corresponding point, and know its training target immediately. The ODE solver only becomes necessary during generation, when we start from noise and must repeatedly follow the model's predictions without knowing the final image.

Watch a network learn where the arrows should point

WHAT IT HAS LEARNED

TRAINING LOSS

–

Target

two moons

field at time t 0.50

show the field

show the target

▶ train

reset

steps 0 loss –

Legend

the target distribution samples from the model the learned field

Give it a few thousand steps; the arrows organise themselves well before the samples look right.

Figure 3 · Choose a target shape and press train. Compare the generated points with the faint target points as the loss changes.

After enough training, the model gives us a vector field that we can follow from noise toward the data distribution.

Optional: the training objective

Try it on handwritten digits

We can now use the same recipe on images. To keep training fast in the browser, this demo represents each 16×16 digit with 24 numbers, using a compression method called PCA. The model moves those 24 coordinates and a decoder turns them back into pixels for display.

We also give the model a digit label c, so we can request a particular digit:

```
velocity = model(x_t, t, c)
```

During training, c is the label of the example image. During generation, we choose it. This is class conditioning. Unlike the hidden destination image z, the label remains available to the model.

Let it rip!

Press train and allow a few thousand updates. Pause before comparing sampler settings so the model stays fixed.

Turn fresh noise into handwritten digits

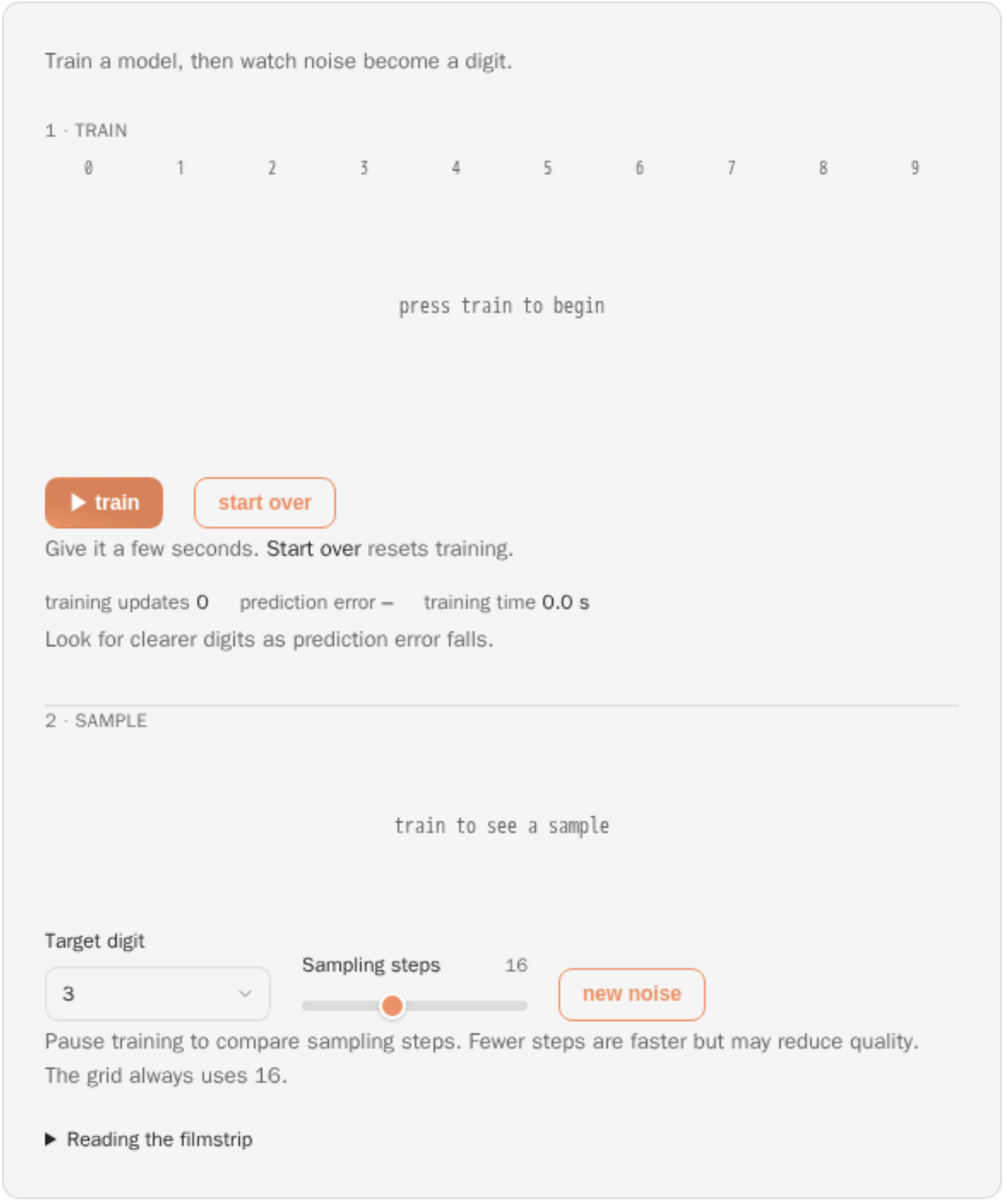


Figure 4 · The grid shows all ten digits; the filmstrip follows one sample from noise to an image.

Try these two experiments:

1. Choose a digit and press new noise: The label stays fixed while the starting point changes. Look for different handwriting styles.
2. Compare 16, 4, and 2 steps with the same noise. Fewer steps cost fewer model evaluations, but can change the result or reduce quality. Straight training paths do not guarantee accurate generation in two steps.

The learned model may produce plausible variations, but this small demo does not establish generalization. That requires evaluation on held-out data and checks for memorization.

What to take away

Flow matching turns a distribution-learning problem into a velocity-prediction problem. Mix data with noise to make an input, subtract them to make a target, and train with squared error. To generate, start from fresh noise and follow the learned field.

You can now read the core of a flow-matching implementation:

- identify its path
- its velocity target
- and its sampler

Different schedules, architectures, and conditions build on those same pieces.

Reference: the notation in one place

These lecture series helped shape this article:

- Stanford CME296: Diffusion & Large Vision Models, by Afshine and Shervine Amidi (Amidi & Amidi, 2026) — diffusion, flow matching, and modern image-generation architectures.
- MIT 6.S184: Flow Matching and Diffusion Models, by Peter Holderrieth (Holderrieth, 2026) — the mathematical foundations of flows and diffusion, with practical examples.
- Stanford CS109: Introduction to Probability for Computer Scientists, by Chris Piech (Piech, 2022) — probability foundations for the ideas used here.

🤖 Generated Content Disclaimer

The blog post was polished using an LLM. This in no way means that I have let an agent run in the background and let it generate the blog. I am a non-english speaker and think LLMs (which are mostly trained in the English Language) can rectify silly grammar mistakes or rephrase sentences that sound less intimidating and cleaner. Hope this helps with the idea of “why should I read, if this was LLM generated”. 🤗 The embeds used are completely LLM generated with major human in the loop feedback. If you find any issues, feel free to send a PR my way.

Citation

For attribution in academic contexts, please cite this work as

```
Aritra Roy Gosthipaty (2026). "From Noise to Images: Flow Matching Explained".
```

BibTeX citation

```
@misc{gosthipaty2026_from_noise_to_images_flow_matching_explained,
  title={From Noise to Images: Flow Matching Explained},
  author={Aritra Roy Gosthipaty},
  year={2026},

}
```

References

Amidi, A., & Amidi, S. (2026). *Stanford CME296: Diffusion & Large Vision Models*. Stanford University. https://www.youtube.com/playlist?list=PLoROMvodv4rNdy8rt2rZ4T2xM0OjADnfu

Holderrieth, P. (2026). *MIT 6.S184: Flow Matching and Diffusion Models*. Massachusetts Institute of Technology. https://www.youtube.com/playlist?list=PL57nT7tSGAAXwjhDYcxEycx5W7YoSrZyt

Piech, C. (2022). *Stanford CS109: Introduction to Probability for Computer Scientists*. Stanford University. https://www.youtube.com/playlist?list=PLoROMvodv4rOpr_A7B9SriE_iZmkanvUg