

Lecture 11: Diffusion

Peter Bloem
Deep Learning

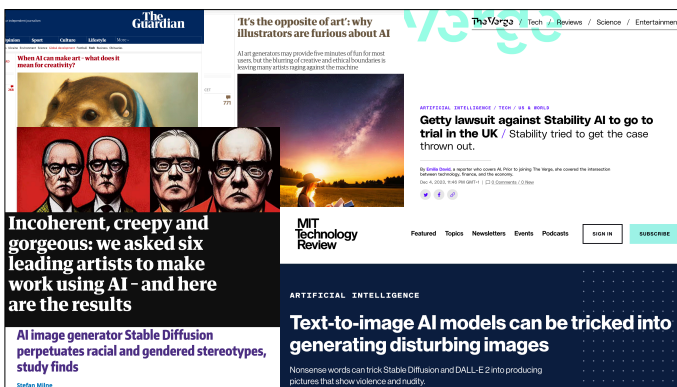
dlvu.github.io



This lecture is about *diffusion*, a technique for building generative models.



Diffusion models behind most of the “AI image generators”, whose output these days populates social media...



... and occasionally, the news.

THE PLAN

- part one: naive diffusion *<- no math!*
- part two: understanding Gaussians
- part three: Gaussian diffusion



We will first discuss a very simple version of the diffusion process, with all the math stripped away. This will help to illustrate how simple the basic idea is: it can be explained with (almost) no math at all.

Next, to help us build an efficient and robust diffusion system, we will need to use *Gaussian noise*. This is noise from a Gaussian or normal distribution. To help us build diffusion on top of Gaussian noise, we will need to understand Gaussians very well, so we will spend a little time developing our intuition, and setting up the properties that we need.

Then we will develop Gaussian diffusion, the basic principle behind modern diffusion systems like DALL-E, Midjourney and Stable Diffusion.

PART ONE: NAIVE DIFFUSION

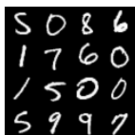


|section-nv|Naive diffusion|

|video||

GENERATIVE MODELING

given:



produce:



This is the basic task of generative modeling: We are given a set of instances. In this case images, but they could also be sentences, bits of music or anything else we can think of. Given this dataset, our job is to produce a model that functions like a black box with a single button: we push the button and out pops an instance that looks like it belongs in the dataset.

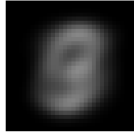
It shouldn't literally come from the data, that would be overfitting, but it should seem to follow the same distribution as the one we got the data from.

WHY IS GENERATIVE MODELING DIFFICULT

No input/output pairs.

How to get and use randomness?

How to avoid *mode collapse*?



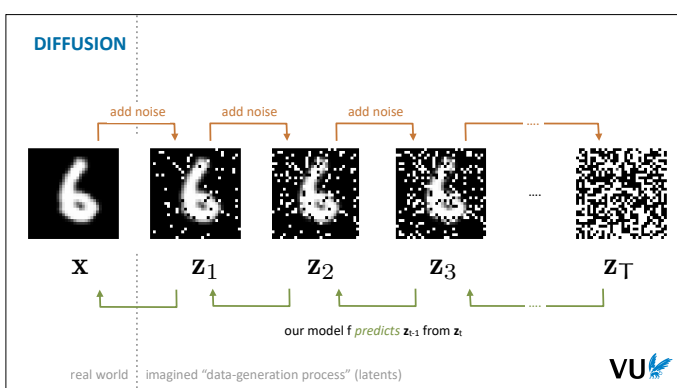
7

Why does generative modeling require all these special tricks to apply deep learning to them? What makes it so difficult?

The first reason is that it doesn't come with explicit input/output pairs. Neural networks are functions from an input to an output. We can only start training them when we have an example of what a given output should look like for a given input. In generative modeling we don't have that. We just want a sample from the data distribution, and we want a new sample every time. We don't even know what the sample should be, so long as it looks like the data.

To make the sample different each time, we need *randomness*. In some way, the model should consume random bits, and then it should translate these to an output sample.

If we do this naively, for instance, by randomly pairing up one noisy input with an instance in the data, and making the model predict one from the other, we get what is called *mode collapse*. Instead of producing different samples with high variety (for instance, a different digit every time), it produces the same sample every time: the point that minimizes the average distance to all data points. Instead of producing variety, the model collapses to a single output. The big challenge in generative modeling is avoiding this kind of behavior.

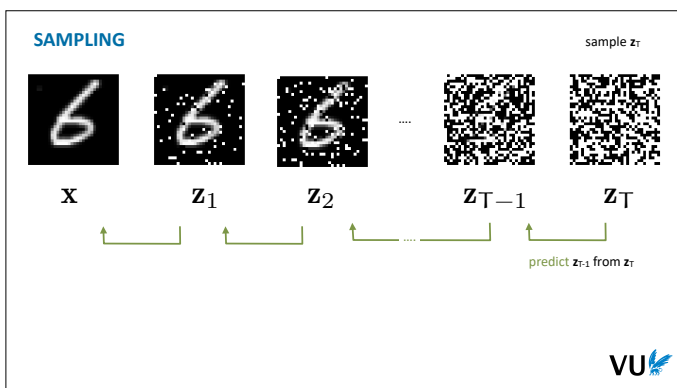


Here is the basic idea behind diffusion. We start with an image from the data, and we gradually, step by step, add a little noise to it. In this case we do that by picking a few random pixels and setting them to black or white at random. As we do this, the information in the original image gradually disappears, until we are left with an image that consists entirely of noise: each pixel is black or white with 50% probability, and there is no further structure.

We can then train a model to *reverse* this process. This is essentially a denoising model, something that neural networks have been able to do for decades. Note that we now have input/output pairs for our neural network. For each time step t we can just give it image $t+1$, and train it on how close the input is to the image at timestep t .

It's not possible to do this task perfectly, but for many of the noisy pixels, especially at the start of the diffusion process, you can see pretty well what it behind them, so we can at least make an educated guess at most stages of the diffusion process.

We call the image from our data x , and we call the sequence of noisy images z_t . We use T to refer to the number of noising steps we apply, so z_T is our final image. We set T to some high value where we're sure that at that point every pixel has been corrupted at least once and no information from the original image remains.



Then, when the time comes we essentially *fake* the process of denoising. For z_T we know exactly what the distribution is. We can sample a new z_T from this distribution very easily: we just create a new image, where we decide to set every pixel to black or white at random. This the same noisy image we would get if we took a sample from our data, and added noise to it bit by bit, except in this case, there is no actual image “behind” the noise.

However, we still ask our model to “denoise” it. As the model starts to remove parts of the noise, it will begin to hallucinate small details to take the place of this noise. In the next step, these details replace what it hallucinates for the next noise to remove and so on. If everything works, at the end of the denoising process, the model has hallucinated an entire image, that looks like it came from our dataset.

NAIVE DIFFUSION (1)

```

training
initialize model f
for x in Data:
  for t in steps:
    z_t = add_noise(z_{t-1})
    o_{t-1} = f(z_{t1}) # prediction
    loss = ||z_{t-1} - o_{t-1}||^2

  backprop & sgd on loss

sampling
given f
sample z_T # random noise
for t from T-1 to 0:
  z_{t-1} = f(z_t)
return z_0 # our sample of x

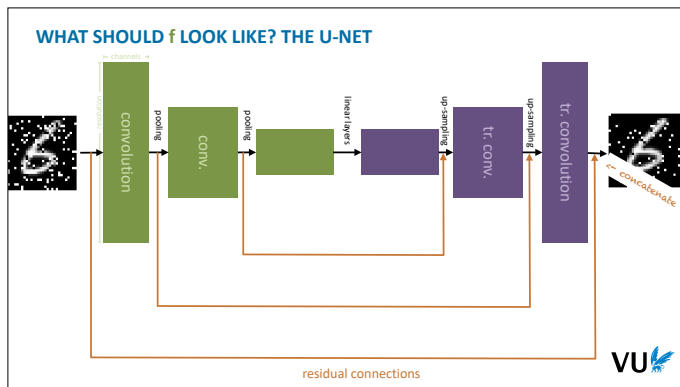
```

VU

Here is the training and sampling process.

During training, we follow the diffusion process, and add noise step by step. At each stage, we then immediately train the model for one step of gradient descent, to predict the previous image from the next one. We do this by running the model and comparing its output o_{t-1} to the true value for z_{t-1} . For now, we will simply use the squared Euclidean distance between the two.

We'll take a more principled approach to the loss later. For now we just work on the intuition that o_{t-1} should be as close to z_{t-1} as possible so the distance between them should be a reasonable thing to minimize (and the squared distance is easier to work with, and remove a square root from the computation, so should give better gradients.).



So, how do we set up the model? We need to map one image to another. This means that we cannot go with fully connected layers if we want to work on high-res images.

For that reason, we want the kind of hourglass shape that we also saw in the variational autoencoder. We use a series of convolutions and pooling operations to gradually work down to a low-resolution, high-dimensional representation, and then we reverse the process with transpose convolutions and upsampling.

A transpose convolution (sometimes called a deconvolution) is essentially a reverse convolution: where a convolution maps an input patch of $k \times k$ pixels to one output pixel, and transpose convolution maps one input pixel to a patch of output pixels. In both cases, we usually set them in such a way that the input and output resolutions are the same. A regular convolution would also work in the decoder stage (but the transpose provide a pleasing kind of structural symmetry).

This allows the model to learn powerful image-to-image functions. However, it also makes it difficult to learn certain very *simple* functions. For instance, in this case, the model needs to leave most of the image intact, and only remove a few noisy pixels. To output all the pixels that should stay the same, the model should encode them through all the convolutions and then decode them again. This is a complex function to learn, when all you want is to reproduce the input.

For this purpose, we add *residual connections*. Before we send the input through the convolutions, we remember it, and then at the end, concatenate it to the input along the channel dimension, and project down to 3 channels with a simple 1×1 convolution.

In other settings it's more common to add residual connections (like we saw in the transformer). As far as I can tell, you can do that here as well, without any loss in performance, but for some reason, the convention when building a UNet is to concatenate rather than add.

Note that UNets existed long before diffusion models. This is just the first time we've come across them in the course, so we introduce them here.

ADDITIONAL INPUTS

time: $f(x, t)$

It helps to know at which point in time we're denoising.

positional embeddings/encodings

Break the translational equivariance of convolutions.

Both are *embeddings*, added to each pixel at every convolution.

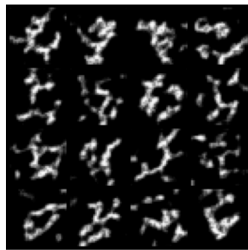


It help to tell the model at which point in time we are denoising, so we give it an extra argument t . We usually don't show this in the notation, but you can assume it's always there.

It can also help a lot to tell the model at which point in *space* image it's working. That is, the coordinates of the pixel to which a convolution is being applied. We've discussed before that convolutions are translation equivariant, which means they necessarily do the same thing regardless of where in the image they are applied. However, many things are very dependent on which part of the image you're in. For example, if you're generating portrait photos, your much more likely to generate a nose in the center of the image than in the top right.

For both t and the pixel coordinates, we create *embeddings*. These are then added to the pixels before every convolution and transpose convolution.

RESULT



Here is the result on a fairly quick experiment without too much tuning.

The final step doesn't exactly show recognizable digits, but it's enough to indicate that the method has promise.

WHAT ARE OUR LIMITATIONS?

Samples aren't iid.

Model needs to *identify* noisy pixels.

Keep the rest the same. But convolutions are *translation invariant*.

Model needs to *choose* arbitrary pixels to denoise.



What can we improve about this approach?

One issue is that we are feeding examples to our model in sequence. It will first see a bunch of very unnoisy images from the start of the diffusion process, and then a bunch of noisy images. This causes learning to become unstable. For instance, the network might forget what it's learned from the unnoisy images while it's learning about the noisy images.

We know that neural networks suffer from catastrophic forgetting under slight changes in the data distribution.

For this reason, it would be better to pick a random t and train the neural network on the images at that point in time. However, in our current set up that would be expensive, since we would have to run the whole diffusion process up to point t , just for one sample.

We could work out the distribution on how many pixels are corrupted at time t , sample that and then corrupt

that amount of pixels, but that would require a bit of math, and I promised no math in this section.

A second downside is that the model needs to identify which pixels are noise and then do something very different for those than for the other pixels. This is not what convolutions are good at. They are much better at doing the same thing to all pixels in parallel.

Even worse, only some of the corrupted pixels were added in this noising step. Ideally, the model removes those, and leaves the rest as they are. But the model has no way to tell which is which. The best it can do is denoise all pixels *a bit*.

ALTERNATIVE

Make f always predict x from z_t .

prediction target

VU

Before we move onto the alternative types of noise, here is one approach to diffusion that solves some of these problems.

Instead of training the network to predict the next step in the denoising process, we train it to *fully denoise* the image. That is, we always predict x from z_t , no matter how far along the noising process we are.

For high t , the image will be so noisy that there isn't much to predict, and the network will mode collapse to predicting mostly the data mean, possibly with very slight amount of variation. For low t , with little noise, it will be very easy for the model to make an almost perfect prediction.

NOISE: ONE PIXEL AT A TIME

Only corrupt one pixel per step.
Only corrupt as-yet uncorrupted pixels.
We keep track of which pixels we've already corrupted.

Thus:

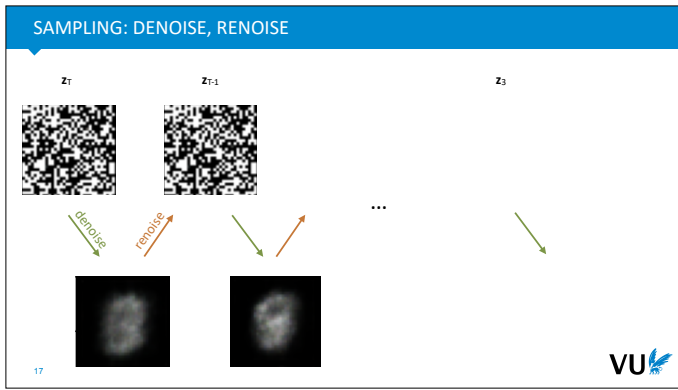
- T is exactly height \times width.
- We can easily sample z_t directly
Just corrupt t pixels.

VU

One thing we can now do is add noise one pixel at a time. We'll keep track of which pixels we've already corrupted, and only corrupt pixels we haven't corrupted before. This means we diffuse for exactly as many steps as we have pixels. This seems inefficient, but that turns out not to be the case.

First, because during training, we can always easily sample z_t *directly*. We just pick a random t between 0 and T , and corrupt t pixels of x chosen randomly (without replacement).

Second, because our sampling algorithm works by denoising and renoising, we don't need to sample with the same step size we did before. We can just denoise from level $t=1000$ and renoise at level $t=900$. This is a way to trade off sampling speed for quality.



The idea is that during sampling, instead of denoising a little bit—from level t to level $t-1$ —we fully denoise from level t , and then add some *noise back in* at level $t-1$. The idea is that the model doesn't need to worry about modeling the noise (which in the case of binomial noise is hard to do for a unet). It can just predict the fully cleaned image and we'll add some noise back using the noising operator we also used during training.

The result is that we start with a very fuzzy prediction: If we feed the model a fully noisy image with no signal, all it can do is predict the data mean. Then, we renoise, hiding most of the image, except some very small details of the fuzzy image. There are different every time, so they may steer the model slightly towards a different outcome. The more noise we remove, the more the predictions serve to add detail, and the more the prediction moves away from the data mean and towards a specific sample.

NAIVE DIFFUSION (2)

```

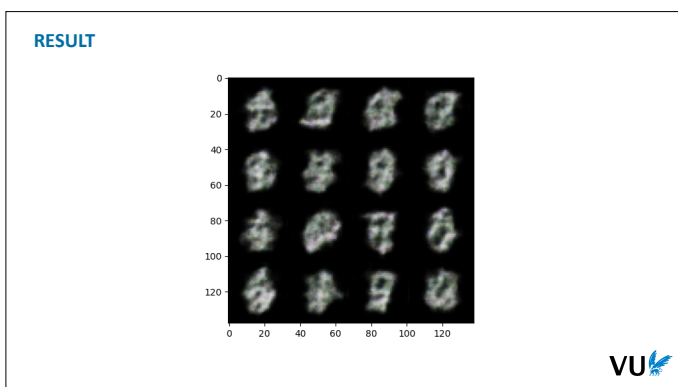
training
initialize model f
T = height * width
for x in Data:
    t = uniform(0, T) # random t
    z_t = add_noise(z_{t-1}, lvl=t)
    o = f(z_{t-1}) # prediction of x
    loss = ||x - o||^2
    brackprop & sgd on loss

sampling
given f
sample z # random noise
for t from T to 0 in steps of K:
    x = f(z)
    z = add_noise(x, lvl=t)
return x # last prediction

```

With that, here is our second diffusion algorithm in full.

The parameter K lets us trade off sampling speed for image quality.



COLD DIFFUSION

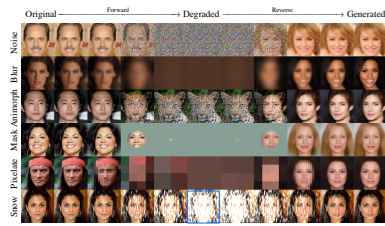


Figure 1: Demonstration of the forward and backward processes for both hot and cold diffusions. While standard diffusions are built on Gaussian noise (top row), we show that generative models can be built on arbitrary and even noiseless/cold image transforms, including the ImageNet-C snowification operator, and an anisotropization operator that adds a random animal image from AFHQ.

source: Bansal, A., Borgnia, E., Chu, H. M., Li, J. S., Kazemi, H., Huang, F., ... & Goldstein, T. (2022). Cold diffusion: Inverting arbitrary image transforms without noise. *arXiv preprint arXiv:2208.09392*.



This kind of no-frills approach to diffusion is not just a teaching tool. It can actually be used to generate high-quality images. In this paper, the authors show that a simple noise function and a squared error loss (like we used) is often all you need. In fact, you don't even need noise. You can gradually degrade the image by any means you can think of: even slowly overlaying a picture of a random animal allows the diffusion process to work.

The authors use our second algorithm, with a slight adaptation required for work with non-noisy degradation operators.

REQUIREMENTS AND LIMITATIONS

We want to

be able to sample from $p(z_t)$ easily.

We need to know what distribution our diffusion converges too.

be able to sample z_t easily.

Without performing t steps of diffusion explicitly.

have the noise affect the whole image.

So f doesn't need to select pixels.

have a principled approach to our loss.

The squared error was an ad-hoc choice.

solution: Gaussian noise (and a lot of math)



21

That's about as far as we can go without mathematics. In the remainder of the lecture, we will show how using Gaussian noise can solve many more of our problems.

PART TWO: UNDERSTANDING GAUSSIANS



|section-nv| Understanding Gaussians |

|video| |

Most diffusion models work on the basis of *Gaussian noise*. That is noise from a Gaussian or normal distribution. These distributions have many nice properties, that mean they are used a lot in machine learning. It can however, be a little daunting to work with Gaussians at first.

THE GAUSSIAN

$$N(\mathbf{x} | \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^k |\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\right)$$

don't worry about this



A big reason for this is that the function for the probability density is complex. This is usually how Gaussians are defined and how their basic properties are proved. We can make things a lot simpler by forgetting about this formula, and defining Gaussians a slightly different way.

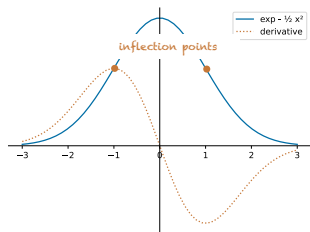
THE PLAN

- Define the *standard* Gaussian in 1D, and ND.
- Build the rest of the family on affine transformation.
Using only simple matrix and vector manipulations.
- Derive the canonical parametrization.
In terms of the mean and Covariance.
- Prove the properties we need for Gaussian diffusion.

24



1D NORMAL DISTRIBUTION (WITHOUT PARAMETERS)



definite *scale*: most probability mass is near 0 (decays more than exponentially)

~68% of probability mass in [-1, 1]

properties:
mean: 0
variance: 1

$$N_0^1(x) = p(x) \propto \exp\left(-\frac{1}{2}x^2\right)$$

$$\log p(x) = -\frac{1}{2}x^2 + c$$



We start with the 1D standard normal distribution. That is, a single distribution which has its mean at zero, and which has a very simple shape.

One of the main properties of the Gaussian is that it has a definite *scale*: it assigns some nonzero probability density to every point between negative infinity and positive infinity, but we can still say with strong certainty that a sample will be in the range [-1, 1], and with extreme certainty that a sample will be in the range [-10, 10]. This means, for instance, because people's heights are roughly normally distributed, you may see the occasional people being 2m tall, and one person in history might have come close to being 3m tall, we will never see a 4m tall person. In short, this normal distribution are why doors are feasible.

Compare this to the distribution on wealth, for instance, which does not have this property: if we look at 10 times as many people, the maximum wealth in the population will always increase by about 10 as well.

To get a distribution like this, we can use exponential decay: every step of 1 we take away from 0, we multiply the density by a fixed number smaller than 1. The function $e^{-|x|}$, or $\exp -|x|$, is an example. The normal distribution adds even more decay by *squaring* the x . $\exp -x^2$.

There are various reasons for this square: One is that with the square in there the curve has two *inflection points*. This is where the derivative flattens out and the curve goes from growing more quickly to growing more slowly (on the left) and from decaying more slowly to decaying more quickly (on the right). Because the peak is in between the two inflection points and that's where the rate of change slows down, we get a "bulk" of probability mass in that range. In short this is a natural range to consider the scale of the normal distribution.

We multiply by 1/2 before the square so that the inflection points land precisely on -1 and 1, which means that for the standard gaussian, the mean is at 0 and the bulk of the mass is in [-1, 1]. The variance also

MULTIVARIATE STANDARD GAUSSIAN

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} \quad x_i \sim N_s^1 \quad \text{The } x_i\text{'s are independent of each other}$$

$$p(\mathbf{x}) = N_s^d = N_s$$



Next, we define a Gaussian in **d dimensions**. That is, a distribution on vectors of d elements. Imagine that we sample d numbers x_i independently from N_s^1 and stick them in a vector \mathbf{x} . The resulting distribution on \mathbf{x} , we call N_s^d (or just N_s if the dimension is clear from context).

That is, the standard Gaussian in d dimensions just consists of d independent standard Gaussians, one along each axis.

WHAT DOES N_s LOOK LIKE?

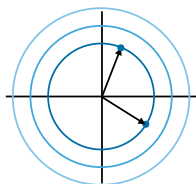
$$p(\mathbf{x}) = p(x_1) \cdot p(x_2) \cdot \dots \cdot p(x_d)$$

$$\log p(\mathbf{x}) = \log p(x_1) + \log p(x_2) + \dots + \log p(x_d)$$

$$= -\frac{1}{2}x_1^2 - \frac{1}{2}x_2^2 - \dots - \frac{1}{2}x_d^2 + c$$

$$= -\frac{1}{2} \sum_i x_i^2 + c$$

$$= -\frac{1}{2} \|\mathbf{x}\|^2 + c$$



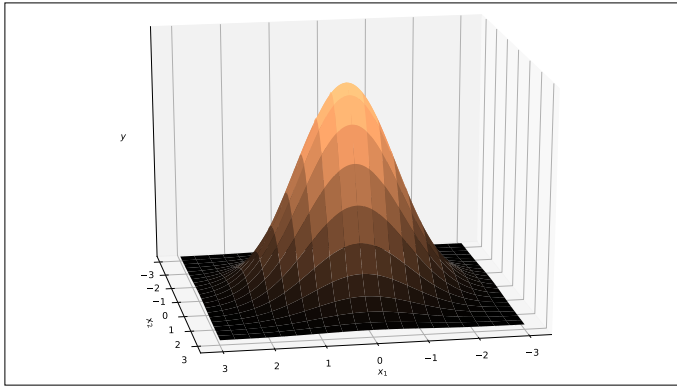
a spherical (or isotropic) distribution



What does the density of the standard Gaussian look like? When we sample \mathbf{x} , each of its elements is sampled independently, so $p(\mathbf{x}) = p(x_1) p(x_2) \dots p(x_d)$. We'll look at the log density of this to make things simpler. The log density of x_i under N_s^1 is $-1/2 x_i^2$ plus some constant, so we get a sum of these for all x_i . If we collect the squares into a sum, we see that this sum is equal to the squared length of \mathbf{x} .

What this tells us, is that all \mathbf{x} that have the same length, have the same (log) density. This means that the contour lines of N_s^2 form circles and in higher dimensions, the contour *surfaces* form *hyperspheres*.

For this reason we call this type of distribution *spherical* (the word *isotropic* is also used).



With a little more reasoning (which we'll skip), you can also work out that the smaller circles have higher density, and that the density decays with the diameter of the circle in the same way as it does in the 1D case.

This means that the distribution in 2D roughly looks like a rotated version of the 1D curve.

DEFINING THE REST OF THE GAUSSIAN FAMILY

We now *define*: any affine transformation of N_s is a "Gaussian".

Affine transform: multiplying by a matrix A and adding a vector t .

that is, if: $s \sim N_s$

and: $x = As + t$

then we *call* $p(x)$ a Gaussian.



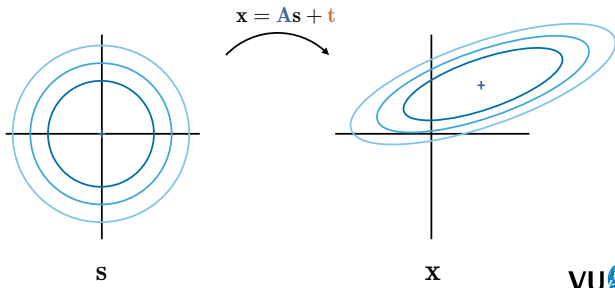
Next, we define the family of Gaussians. We will do this by assuming that we have a vector s which is drawn from a standard normal Gaussian in d dimensions. We then apply an affine transformation to s (that is, a **matrix multiplication** and a **translation**), resulting in the vector x .

Obviously, if we know s , then x is fully determined. But what if I told you I had done this, but didn't tell you what s I had drawn. What would the distribution $p(x)$ be? We call this distribution a *Gaussian*. The matrix A and the vector t function (for the time being) as the parameters of this Gaussian.

Note that we are using this as a definition of what a Gaussian is. In other contexts, you may see the Gaussian defined differently (in terms of that horrible formula) and then the fact that they are all affine transformations of N_s is a property of Gaussians. Here we use it as a definition, in order to simplify the construction of the Gaussians.

For our definition, it is not necessary that A is square. However, any Gaussian defined by a non-square A , can also be defined by a square A (possibly with s of a different dimension), so we will limit ourselves mostly to square A s in this lecture.

AFFINE TRANSFORMS OF CIRCLES ARE ELLIPSES



How should we visualize these Gaussians? A simple way is to think of all the points inside a circle under N_s . This circle will be mapped to an ellipse by A and t .

In higher dimensions, we call these ellipsoids.

If, say, 68% of the points s fall inside the inner circle, then they will all be mapped to the inner ellipse on the right. This means that 68% of the points x will fall inside the inner ellipse.

In short, the general Gaussian is not spherical.

question: What would A and t need to look like for the distribution on the right to be spherical as well?

THE MEAN OF A GAUSSIAN

NB: the mean is a *property* of Gaussian now, not part of its definition.

$$\begin{aligned}
 \mu &= E x \text{ with } x = A s + t \\
 &= E_{s \sim N_s} A s + t \quad \leftarrow E \text{ is a linear function} \\
 &= A (E s) + t \\
 &= \mathbf{0} + t \quad \quad E s = \begin{pmatrix} E s_1 \\ E s_2 \\ \vdots \\ E s_d \end{pmatrix} = \mathbf{0}
 \end{aligned}$$

Next, we can ask about some of the properties of these Gaussians. For example, the mean.

The mean is defined as the expected value of the distribution. That is, what does the average of a sample converge to for an increasing sample size?

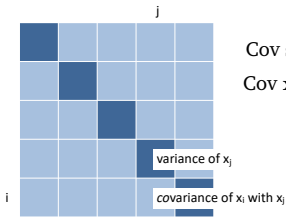
Again, the mean is normally part of the definition of a Gaussian, but we're flipping it around. We've already defined the Gaussian, so now we're studying the mean as a property.

We use two properties in this derivation. First, that the Expected value is a *linear function*. That means that any matrix multiplication and vector addition can be taken out of the mean in the same way we would take it out of a set of brackets.

The second is that the mean of the standard Gaussian is $\mathbf{0}$ (that is, the 0-vector). This follows from the fact that the elements of s are independently sampled, so the expected value of s is the vector made up of the expected values of its elements. And the expected value of N_s is 0.

The end result is that the mean of a Gaussian (A, t) is equal to t .

THE COVARIANCE OF A GAUSSIAN



$$\text{Cov } \mathbf{s} = \mathbf{I}$$

$$\begin{aligned} \text{Cov } \mathbf{x} &= E(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T \\ &= E(\mathbf{A}\mathbf{s} + \mathbf{t} - \boldsymbol{\mu})(\mathbf{A}\mathbf{s} + \mathbf{t} - \boldsymbol{\mu})^T \\ &= E\mathbf{A}\mathbf{s}(\mathbf{A}\mathbf{s})^T = \mathbf{A}E\mathbf{s}(\mathbf{A}\mathbf{s})^T = \mathbf{A}E\mathbf{s}\mathbf{s}^T\mathbf{A}^T \\ &= \mathbf{A}(E\mathbf{s}\mathbf{s}^T)\mathbf{A}^T = \mathbf{A}\mathbf{A}^T \end{aligned}$$



Next, the covariance. Again, think of this as a *property* of any multivariate distribution not (necessarily) a parameter.

The covariance matrix of $p(\mathbf{x})$ collects the variance of all the elements of \mathbf{x} along the diagonal and the covariance of any two elements on the off-diagonal elements. This tells us what the covariance matrix of \mathbf{s} is: the variances are all 1, because all elements are distributed according to N_{s^1} , and the covariances are all 0, because all elements are independently drawn. In short, the covariance matrix of \mathbf{s} is the identity matrix.

We can now work out what the covariance matrix of a Gaussian (\mathbf{A}, \mathbf{t}) is. We use the definition that the covariance matrix of $p(\mathbf{x})$ is the expected outer product of $(\mathbf{x} - \boldsymbol{\mu})$, where $\boldsymbol{\mu}$ is the mean of $p(\mathbf{x})$.

In the previous slide, we worked out that $\boldsymbol{\mu} = \mathbf{t}$, so if we fill in the definition of \mathbf{x} in terms of \mathbf{s} , the translation term \mathbf{t} and the $\boldsymbol{\mu}$ cancel out. Next, we are left with two matrix multiplications. Working these out of the expectation, we are left with $E\mathbf{s}\mathbf{s}^T$, which is the covariance of the standard Gaussian, which is \mathbf{I} , so it disappears.

The end result is that the covariance of the Gaussian (\mathbf{A}, \mathbf{t}) is $\mathbf{A}\mathbf{A}^T$.

TRADITIONAL PARAMETRIZATION

$$\mathbf{s} \sim N_{\mathbf{s}}$$

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{t}$$

$$\mathbf{y} \sim N(\boldsymbol{\mu} = \mathbf{t}, \boldsymbol{\Sigma} = \mathbf{A}\mathbf{A}^T)$$

NB: (\mathbf{t}, \mathbf{A}) is not a *unique* name for Gaussian, $(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is.



Because \mathbf{A} and \mathbf{t} transform so simply to the mean and the covariance (and back) this means we can now use the mean and covariance as a name for the Gaussian as well (we have a simple method for translating one type of name to another).

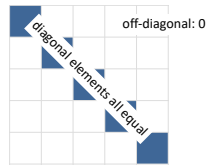
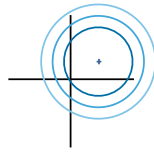
The mean and covariance is, of course, the traditional way of parametrizing the Gaussians. It has one benefit over the affine transformation: the mean and covariance are *unique names*. The affine transformation naming scheme has many names for the same Gaussian.

Consider for instance what we get if we use a rotation matrix and $\mathbf{t}=\mathbf{0}$.

SPHERICAL GAUSSIANS

$$\mathbf{x} = \sigma \mathbf{I} \mathbf{s} + \mathbf{t} \quad \mathbf{s} \sim N_{\mathbf{s}}$$

$$\mathbf{x} \sim N(\mathbf{t}, \sigma \mathbf{I} (\sigma \mathbf{I})^T) = N(\mathbf{t}, \sigma^2 \mathbf{I})$$



σ^2 : variance
 σ : standard deviation

One thing that is going to be important when we build diffusion models is the subset of Spherical Gaussians. These are Gaussians that retain the property of the standard Gaussian that all the contour lines are circles, spheres or hyper-spheres.

We achieve this if \mathbf{A} scales each dimension uniformly (that is by the same amount). In other words, if \mathbf{A} is the identity matrix \mathbf{I} times some scalar σ .

There are other matrices that also lead to spherical Gaussians (again, consider rotations), but if we limit ourselves to \mathbf{A} s that are a scalar times the identity matrix, we can define all spherical Gaussians.

Following what we derived earlier, this shows that the covariance matrix is \mathbf{I} times σ^2 .

By seeing what happens in the 1D case, we can see that these are natural analogues of the variance and the standard deviation, and we will name them as such. That is, even though a d -dimensional distribution doesn't have a single variance or standard deviation, in a spherical distribution the variance and standard deviation are the same in all directions, so in this specific case, we can talk about a single scalar variance and standard deviation.

PROPERTIES

- 1) Any affine transformation of a Gaussian is a Gaussian.
- 2) The sum of two Gaussian variables is a Gaussian.
- 3) Making one Gaussian sample the mean of another results in a Gaussian.
- 4) The KL divergence between two Gaussians has closed form solution.

To finish up our discussion, we will look at 4 properties of Gaussians that will help us to define the Gaussian diffusion process.

1) AFFINE TRANSFORMS OF GAUSSIANS

$$\mathbf{x} \sim \mathcal{N}(\cdot, \cdot)$$

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{t}$$

$$\mathbf{x} = \mathbf{B}\mathbf{s} + \mathbf{q} \quad \mathbf{s} \sim \mathcal{N}_s$$

$$\mathbf{y} = \mathbf{A}(\mathbf{B}\mathbf{s} + \mathbf{q}) + \mathbf{t} = \mathbf{A}\mathbf{B}\mathbf{s} + \mathbf{A}\mathbf{q} + \mathbf{t}$$

matrix mult translation vector



36

We start with the simplest one. We already know that, by definition, an affine transformation of the standard Gaussian is a Gaussian. If we combine this with the fact that the composition of two affine transformation is another affine transformation, we can show that the affine transformation of any Gaussian (standard or not) yields another Gaussian.

Say that we have a Gaussian G and an affine transformation (\mathbf{A}, \mathbf{t}) . What we want to show is that sampling from G and applying (\mathbf{A}, \mathbf{t}) results in a Gaussian.

Sampling from G is, by definition, equivalent to sampling from \mathcal{N}_s and then transforming by some affine transformation (\mathbf{B}, \mathbf{t}) in the slides. All we are doing then, is applying another affine transformation (\mathbf{A}, \mathbf{t}) after that. The composition of two affine transformations is another affine transformation, so what we are doing is equivalent to sampling from \mathcal{N}_s and then applying a single affine transformation. By our definition, this is a Gaussian.

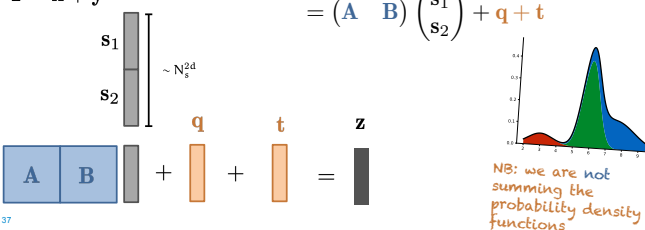
As the slides show, the combined affine transformation is $(\mathbf{A}\mathbf{B}, \mathbf{A}\mathbf{q} + \mathbf{t})$.

2) SUM OF GAUSSIAN VARIABLES

$$\mathbf{x} = \mathbf{A}\mathbf{s}_1 + \mathbf{t} \quad \mathbf{s}_1, \mathbf{s}_2 \sim \mathcal{N}_s \quad \mathbf{z} = \mathbf{x} + \mathbf{y} = \mathbf{A}\mathbf{s}_1 + \mathbf{t} + \mathbf{B}\mathbf{s}_2 + \mathbf{q}$$

$$\mathbf{y} = \mathbf{B}\mathbf{s}_2 + \mathbf{q} \quad = \mathbf{A}\mathbf{s}_1 + \mathbf{B}\mathbf{s}_2 + \mathbf{q} + \mathbf{t}$$

$$\mathbf{z} = \mathbf{x} + \mathbf{y} \quad = (\mathbf{A} \quad \mathbf{B}) \begin{pmatrix} \mathbf{s}_1 \\ \mathbf{s}_2 \end{pmatrix} + \mathbf{q} + \mathbf{t}$$



37

Here's a slightly more involved one to prove: if we take samples x and y from two separate Gaussian, their sum is also distributed as a Gaussian.

Note that this is distinct from taking the probability density functions of two Gaussians and summing them (the result of which is a mixture-of-gaussians).

The way to show this, and to work out what the parameters of the resulting Gaussian are, is to frame this process as an affine transformation. If we fill in the definitions of \mathbf{x} and \mathbf{y} , we get an affine transformation in two variables \mathbf{s}_1 and \mathbf{s}_2 . We can rephrase this as an affine transformation in a single variable, by simply concatenating \mathbf{A} and \mathbf{B} horizontally and \mathbf{s}_1 and \mathbf{s}_2 vertically.

Finally, we can see that the concatenation of \mathbf{s}_1 and \mathbf{s}_2 is a standard Gaussian vector. This is because we defined a standard Gaussian vector as one whose elements were sampled independently from \mathcal{N}_s^1 . The elements of \mathbf{s}_1 and \mathbf{s}_2 were each sampled independently, so \mathbf{s} must also be standard Gaussian.

The result is that z is an affine transformation of a standard Gaussian, so it must be a Gaussian.

Note, however, that this doesn't give us as nice a definition as we have found so far: the problem is that we are defining z as an affine transformation of Gaussian noise \mathbf{s} , but \mathbf{s} has twice as many dimensions as \mathbf{z} . In our previous definitions \mathbf{s} always had the same number of dimensions as the resulting variable.

The solution is to work out the covariance matrix. This is a unique parametrization of this Gaussian on \mathbf{z} , so it must be square.

BUT WHAT ARE THE PARAMETER OF THE GAUSSIAN ON z ?

$$\mathbf{x} = \mathbf{A}\mathbf{s}_1 + \mathbf{t} \sim \mathcal{N}(\boldsymbol{\mu}_x = \mathbf{t}, \boldsymbol{\Sigma}_x = \mathbf{A}\mathbf{A}^T)$$

$$\mathbf{y} = \mathbf{B}\mathbf{s}_2 + \mathbf{q} \sim \mathcal{N}(\boldsymbol{\mu}_y = \mathbf{q}, \boldsymbol{\Sigma}_y = \mathbf{B}\mathbf{B}^T)$$

$$\boldsymbol{\Sigma}_z = (\mathbf{A} \ \mathbf{B})(\mathbf{A} \ \mathbf{B})^T = \mathbf{A}\mathbf{A}^T + \mathbf{B}\mathbf{B}^T$$

$$\mathbf{x} + \mathbf{y} \sim \mathcal{N}(\boldsymbol{\mu}_x + \boldsymbol{\mu}_y, \boldsymbol{\Sigma}_x + \boldsymbol{\Sigma}_y)$$

$$\mathbf{z} = \mathbf{C}\mathbf{s}_d + \mathbf{t} + \mathbf{q}$$

with

$$\mathbf{C}\mathbf{C}^T = \mathbf{A}\mathbf{A}^T + \mathbf{B}\mathbf{B}^T$$

$$\mathbf{s}_d \sim \mathcal{N}_s^d$$

Cholesky or Eigendecomposition

38

To get the covariance matrix matrix, we need to work out what we get if we multiply $(\mathbf{A} \ \mathbf{B})$ by its transpose.

Note that $(\mathbf{A} \ \mathbf{B})^T$ flips the matrix along its diagonal. This means that the height and width become reversed.

The result is $\mathbf{A}\mathbf{A}^T + \mathbf{B}\mathbf{B}^T$, the sum of the two covariances of the original two distributions.

If we want to translate this back to an affine transformation, we will need to find a square matrix \mathbf{C} , such that $\mathbf{C}\mathbf{C}^T$ is equal to this expression. Happily this is always possible (usually in a few ways), using operations like a Eigen- or Cholesky decomposition (note that is $\mathbf{A}\mathbf{A}^T + \mathbf{B}\mathbf{B}^T$ symmetric)

SUMMING SPHERICAL GAUSSIANS

$$\mathbf{x} = \boldsymbol{\mu}_x + \sigma_x \mathbf{s}_1 \quad \mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}_x, \sigma_x^2 \mathbf{I})$$

$$\mathbf{y} = \boldsymbol{\mu}_y + \sigma_y \mathbf{s}_2 \quad \mathbf{y} \sim \mathcal{N}(\boldsymbol{\mu}_y, \sigma_y^2 \mathbf{I})$$

$$\mathbf{x} + \mathbf{y} \sim \mathcal{N}(\boldsymbol{\mu}_x + \boldsymbol{\mu}_y, \sigma_x^2 \mathbf{I} + \sigma_y^2 \mathbf{I})$$

$$= \mathcal{N}(\boldsymbol{\mu}_x + \boldsymbol{\mu}_y, (\sigma_x^2 + \sigma_y^2) \mathbf{I})$$

$$\mathbf{x} + \mathbf{y} = \boldsymbol{\mu}_x + \boldsymbol{\mu}_y + \sqrt{\sigma_x^2 + \sigma_y^2} \mathbf{s}$$

square the variances, sum and then take square root

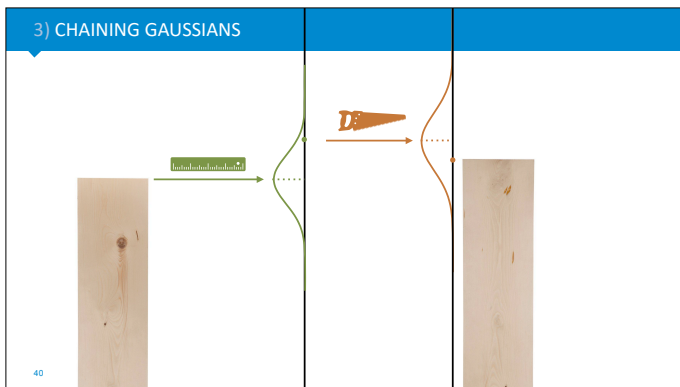
VU

39

This operation, summing two Gaussian variables, will become especially important in the context of spherical Gaussians.

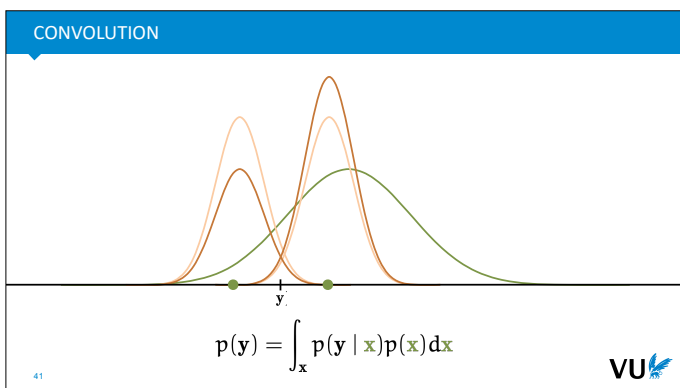
Applying the rule we derived in the previous slide, we can see how to work out the sum of two Gaussians in the affine transformations notation: we take the two variances (the sigma's without a square), we square them, then sum them then take the square root again.

We recommend that you memorize this rule as well as understanding it.



The third property, we'll call *chaining* Gaussians. This is when we take a sample from **one Gaussian**, and make it the mean of **another Gaussian**.

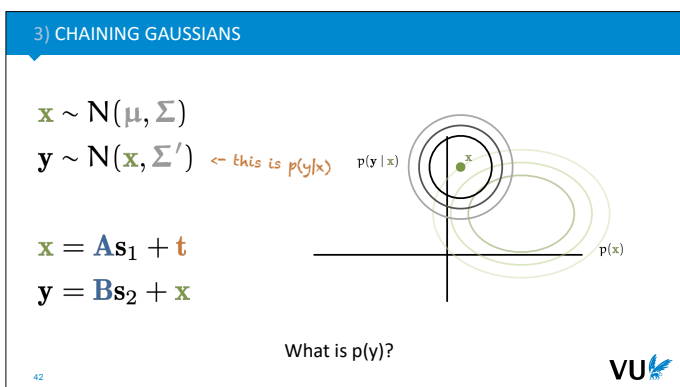
This happens, for instance, if you take one noisy measurement, and use it in another noisy process. For example if I measure the length of one plank of wood, and then try cut another plank of wood at that length, the length of the resulting plank of wood will be a combination of how poorly I measured the first plank and how poorly I made my cut. If both processes are Gaussian, what we'd like to show, is that the resulting distribution on the length of the second plank has an acceptable variance, and that the mean of the whole process is equal to the length of the first plank, so that at least on average the planks are the same length.



We can think of this as a *convolution* of two Gaussians. Each possible measurement x contributed some probability density to the final point y , proportional to how likely x is to be measured. Adding up (or rather integrating) all these bits of probability density, will give us the answer.

We call this a *convolution*, because we're sliding one Gaussian along the horizontal axis, and taking a (weighted) "sum" of all the copies we encounter.

Of course, this is not an easy integral to work out, especially if we want to move to higher dimensions. Instead, we will take a geometric perspective again.



Here's what the picture looks like in N dimensions. Writing the two Gaussians as transformations of standard Gaussian noise, shows how we can solve this puzzle: we just plug the definition of \mathbf{x} into the definition of \mathbf{y} .

GEOMETRIC SOLUTION

$$\mathbf{x} = \mathbf{A}\mathbf{s}_1 + \mathbf{t}$$

$$\mathbf{y} = \mathbf{B}\mathbf{s}_2 + \mathbf{x}$$

$$\mathbf{y} = \underbrace{\mathbf{B}\mathbf{s}_2 + \mathbf{A}\mathbf{s}_1 + \mathbf{t} + \mathbf{0}}_{\text{sum of two Gaussians } (\mathbf{A}, \mathbf{t}) \text{ and } (\mathbf{B}, \mathbf{0}) \text{ so:}} \mathbf{N}(\mathbf{t} + \mathbf{0}, \mathbf{B}\mathbf{B}^T + \mathbf{A}\mathbf{A}^T)$$

$$\text{or: } \mathbf{y} = \mathbf{C}\mathbf{s} + \mathbf{t} \quad \text{with } \mathbf{C}\mathbf{C}^T = \mathbf{B}\mathbf{B}^T + \mathbf{A}\mathbf{A}^T$$

43



Doing this gives us \mathbf{y} as a function of two vectors of standard Gaussian noise \mathbf{s}_1 and \mathbf{s}_2 . The trick here, is that we can interpret this as the sum of two Gaussians (one with mean \mathbf{t} , and one with mean $\mathbf{0}$).

We've seen already how to combine these into a a single Gaussian. We sum the means, and the covariances.

The result is that our distribution on \mathbf{y} has \mathbf{t} as a mean, and the sum of the two covariances for its covariance matrix.

In the example of cutting a plank, we see that we do in deed get two planks of the same length on average, and the sum of the variances of our measuring and our cutting needs to make an acceptable error margin for the final product.

THE SPHERICAL CASE

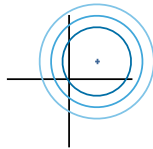
$$\mathbf{x} = \sigma\mathbf{s}_1 + \mathbf{t}$$

$$\mathbf{y} = \tau\mathbf{s}_2 + \mathbf{x}$$

$$\mathbf{y} = \sigma\mathbf{s}_2 + \tau\mathbf{s}_1 + \mathbf{t} + \mathbf{0}$$

$$\mathbf{N}(\mathbf{t} + \mathbf{0}, (\sigma^2 + \tau^2)\mathbf{I})$$

$$\mathbf{y} = \sqrt{\sigma^2 + \tau^2}\mathbf{s} + \mathbf{t}$$



44



As before, the story becomes much simple in the spherical case. Here, we just square and sum the variances and then take the square root.

4) KL DIV. HAS A CLOSED FORM SOLUTION

$$K(p, q) = -\mathbb{E}_{\mathbf{x} \sim q} \log \frac{p(\mathbf{x})}{q(\mathbf{x})}$$

45



The KL divergence, written here as an expectation, expresses how "similar" two probability distributions are. If the are exactly the same, the KL divergence is 0, and the more they differ, the bigger it gets.

It's relevant to us, because the quantity we're actually interested in optimizing (the data log-likelihood under our model) can often be rewritten into one or more KL divergences, as we saw earlier with the VAE.

For this to lead to a loss function we can use, we need a closed-form expression for the KL divergence. The formula here is no good to us, because the expectation isn't necessarily computable. We need an expression that we cannot just compute quickly, but that we can backpropagate through (remember, this will be our loss function).

This is our final reason for liking Gaussians: the KL divergence between two Gaussians has a simple closed-form expression.

USEFUL PROPERTY

$$\begin{aligned}
 E_{\mathbf{x} \sim N(\boldsymbol{\mu}, \sigma^2)} \|\mathbf{x} - \mathbf{y}\|^2 &= E_{\mathbf{s} \sim N_s} \|\sigma \mathbf{s} + \boldsymbol{\mu} - \mathbf{y}\|^2 \\
 &= E_{\mathbf{s}} (\sigma \mathbf{s} + \mathbf{q})^T (\sigma \mathbf{s} + \mathbf{q}) \text{ with } \mathbf{q} = \boldsymbol{\mu} - \mathbf{y} \\
 &= E_{\mathbf{s}} \sigma^2 \mathbf{s}^T \mathbf{s} + 2\mathbf{s}^T \mathbf{q} + \mathbf{q}^T \mathbf{q} \\
 &= \sigma^2 E_{\mathbf{s}} \mathbf{s}^T \mathbf{s} + 2\mathbf{q}^T E_{\mathbf{s}} \mathbf{s} + E \|\mathbf{q}\|^2 \\
 &= \sigma^2 d + 2\mathbf{q}^T \mathbf{0} + E_{\mathbf{s}} \|\boldsymbol{\mu} - \mathbf{y}\|^2 \\
 &= \sigma^2 d + \|\boldsymbol{\mu} - \mathbf{y}\|^2
 \end{aligned}$$

46



We will make use of the following useful property: the expected value between random vector \mathbf{x} and constant vector \mathbf{y} , if \mathbf{x} is distributed according to a spherical Gaussian, is the squared distance between \mathbf{y} and the mean of the Gaussian, plus d times the variance.

If you've following the derivation, note that on line 5 we make use of the fact that $E \mathbf{s}^T \mathbf{s} = d$ (where d is the dimension of \mathbf{s}). You can work this out by writing the dot product as a sum, and working the expectation inside the sum. Then note that $E s^2$ is the variance of N_{s^1} , which we know to be 1.

$$\begin{aligned}
 K [N(\boldsymbol{\mu}, \sigma^2), N(\boldsymbol{\nu}, \tau^2)] &= -E_{\mathbf{x} \sim N(\boldsymbol{\nu}, \tau^2)} \log \frac{N(\mathbf{x} | \boldsymbol{\mu}, \sigma^2)}{N(\mathbf{x} | \boldsymbol{\nu}, \tau^2)} \\
 &= -\frac{1}{2\sigma^2} E_{\mathbf{x}} \|\mathbf{x} - \boldsymbol{\mu}\|^2 - \frac{1}{2\tau^2} E_{\mathbf{x}} \|\mathbf{x} - \boldsymbol{\nu}\|^2 + c \\
 &= \frac{1}{2\sigma^2} (\tau^2 d + \|\boldsymbol{\mu} - \boldsymbol{\nu}\|^2) - \frac{1}{2\tau^2} (\tau^2 d + \|\boldsymbol{\nu} - \boldsymbol{\nu}\|^2) + c \\
 &= \frac{\tau^2 d}{2\sigma^2} + \frac{1}{2\sigma^2} \|\boldsymbol{\mu} - \boldsymbol{\nu}\|^2 - \frac{1}{2} d + c \\
 &= \frac{1}{2\sigma^2} \|\boldsymbol{\mu} - \boldsymbol{\nu}\|^2 + c
 \end{aligned}$$

47



Now, to the KL divergence.

We won't work it out for general Gaussians (although it exists), we will just show a simple case: the KL loss for two spherical Gaussians, in terms of the means. We will assume that our model only controls the mean of one of the Gaussians, so that we can ignore any terms that are constant with respect to that mean.

In the first line, we expand the logarithm over the two Gaussians, and remove any terms that don't contain $\boldsymbol{\mu}$, which is the term we assume we're optimizing. Then we work the expectations inside as shown.

To line 2, we can apply the property from the last slide, resulting in line 3 as shown. The second term reduces to a constant, and the first to a constant (in $\boldsymbol{\mu}$) plus a constant, times the distance between $\boldsymbol{\mu}$ and $\boldsymbol{\nu}$, which is the fundamental property that we should be minimizing.

This should make some intuitive sense: to make two spherical Gaussians as similar as possible to one another, without changing their variances: align their means.

RECAP

We can do a lot of thinking about Gaussians by:

- Using a geometric definition.
- Relying on linear properties.

We have shown that:

- affine transformations between Gaussians
- summing of Gaussian variables
- and chaining Gaussians

all result in *more Gaussians*.

48



|section-nv| Gaussian diffusion |

|video| |

At last, we are ready to start building Gaussian diffusion models.

PART THREE: GAUSSIAN DIFFUSION



Denoising Diffusion Probabilistic Models

Jonathan Ho UC Berkeley jonathanh@berkeley.edu
Ajay Jain UC Berkeley ajayj@berkeley.edu
Pieter Abbeel UC Berkeley pabbeel@cs.berkeley.edu

Abstract

We present high quality image synthesis results using diffusion probabilistic models, a class of latent variable models inspired by considerations from nonequilibrium thermodynamics. Our best results are obtained by training on a weighted variational bound designed according to a novel connection between diffusion probabilistic models and denoising score matching with Langevin dynamics, and our models naturally admit a progressive noisy decomposition scheme that can be interpreted as a generalization of autoregressive decoding. On the unconditional CIFAR10 dataset, we obtain an inception score of 9.46 and a state-of-the-art FID score of 1.77. On 256x256 LSUN, we obtain sample quality similar to ProgressiveGAN. Our implementation is available at <https://github.com/yang-song/denoising-diffusion-probabilistic-models>.

1 Introduction

Deep generative models of all kinds have recently exhibited high quality samples in a wide variety of data modalities. Generative adversarial networks (GANs), autoregressive models, flows, and variational autoencoders (VAEs) have synthesized striking image and audio samples [12] [25] [3] [5] [19] [23] [14] [16], and there have been remarkable advances in energy-based modeling and score matching that have produced images comparable to those of GANs [11] [20].

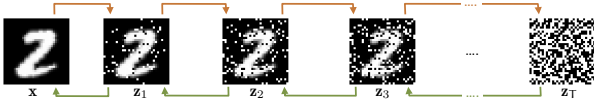


There are many different ways to do this, and many state-of-the-art models add their own little tweak and touches to the derivation.

We will focus on this specific approach. The first to show that diffusion models could be trained to do the things we'd seen models like GANs do, like generate realistic human faces. Besides being very powerful, this model is also remarkable for its simplicity. More powerful models have been published since, but this is probably the best in terms of its tradeoff of simplicity and performances, so it's a good model to use as a foundation of your understanding of diffusion models.

If you are continuing in diffusion models after this lecture, perhaps for your thesis, then we suggest carefully following all the steps in the derivation of this model, and then seeing how other models deviate.

RECAP: DIFFUSION



Add noise to the image step by step. Train a model to reverse the process.

Make sure that:

the process converges to a *known distribution* on z_T .

Should be easy to sample from.

we can sample the noisy image z_t directly for any t .

Without going through t steps of diffusion.

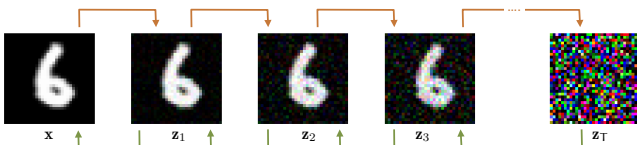


Here is the picture of diffusion models that we've built up so far. We add noise to an image, slowly erasing the information in it step by step, and then we train a model to reverse this process.

The key feature that made this work is that the noise we converge to in the diffusion is a known distribution that we can easily sample from.

We also saw that a nice property to have was if we can sample the noisy image z_t at time step t directly rather than explicitly having to go to t steps of diffusion. We will require both these properties from our Gaussian noise process.

GAUSSIAN NOISE

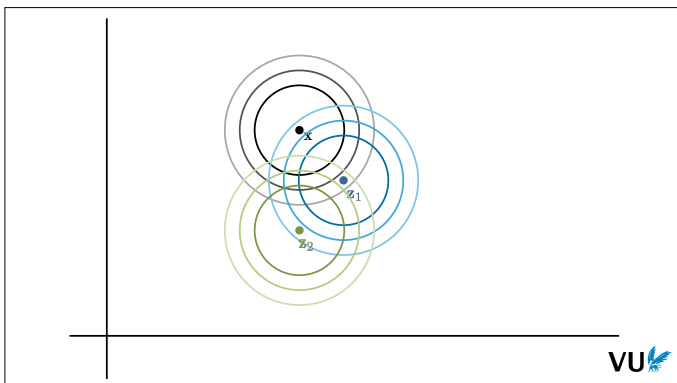


52



Here is what Gaussian noise looks like.

We will model our data as color images, even though this particular dataset is black and white. We don't have to do this, but it isn't difficult for the model to learn that the images should always be black and white, and it's visually a little more interesting.



Here is what that looks like in 2 dimensions. We start with a Gaussian that is centered on \mathbf{x} (that is, \mathbf{x} is its mean). We then sample z_1 from that distribution. This is an image that will look like a noisy version of \mathbf{x} . We then repeat the process, we create a Gaussian that is centered on z_1 , sample from that and so on.

This is, of course, the Gaussian chaining that we talked about in the last part. So we know that our distribution on z_t is a Gaussian.

Currently, the variance of this Gaussian grows bigger with every step we take, and the mean stays at \mathbf{x} , so we're not really removing information from \mathbf{x} very effectively. We'll fix that by slightly tweaking the process.

$$\mathbf{z}_{t+1} = \mathbf{z}_t + \sigma^2 \mathbf{s}$$

$$\sim \mathcal{N}(\mathbf{z}_t, \sigma^2 \mathbf{I})$$

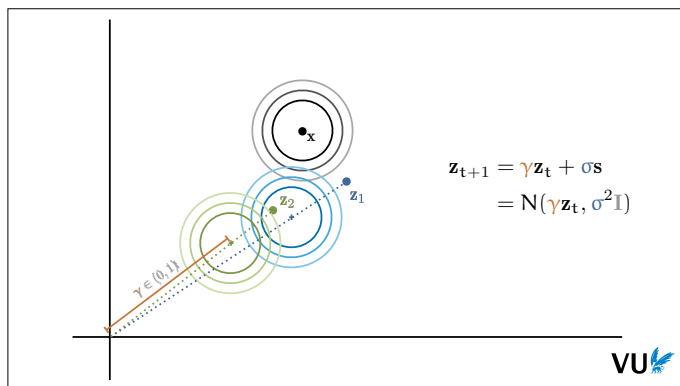
$$\mathbf{z}_T \sim ?$$

Wouldn't it be nice if this was $\mathcal{N}(\mathbf{0}, \mathbf{I})$?



Here's how we can write down this diffusion process.

What we need it for our process to converge to a single, known Gaussian that we can sample from. This can be any Gaussian, but it would be neatest if this was the *standard* Gaussian. We'll see what we need to do to achieve that.



The first thing we need is for the mean to go to $\mathbf{0}$. We can achieve this with a simple trick. Instead of putting the next Gaussian right on top of the previous sample, we first shrink the sample a bit, by multiplying it by some value, γ , between 0 and 1.

Note that the Gaussian chaining property still applies: this shrinking is an affine operation, so we are still chaining Gaussians.

This shrinking creates a kind of force that steadily pulls the mean to $\mathbf{0}$. It's reasonable to assume that this will cause the process to converge to a zero-mean distribution, but we should really prove that. Additionally, we should work out what to set the variance to at each step, so that the distribution converges to variance 1.

WHAT γ AND σ ENSURE CONVERGENCE TO $\mathcal{N}(\mathbf{0}, \mathbf{I})$?

$$\mathbf{z}_1 = \gamma \mathbf{x} + \sigma \mathbf{s}_1$$

$$\mathbf{z}_2 = \gamma \mathbf{z}_1 + \sigma \mathbf{s}_2$$

$$\mathbf{z}_2 = \gamma(\gamma \mathbf{x} + \sigma \mathbf{s}_1) + \sigma \mathbf{s}_2 = \gamma^2 \mathbf{x} + \gamma \sigma \mathbf{s}_1 + \sigma \mathbf{s}_2$$

$$= \gamma^2 \mathbf{x} + \sqrt{\gamma^2 \sigma^2 + \sigma^2} \mathbf{s}$$

$$\mathbf{z}_3 = \gamma \mathbf{z}_2 + \sigma \mathbf{s}_3$$

$$= \gamma(\gamma^2 \mathbf{x} + \sqrt{\gamma^2 \sigma^2 + \sigma^2} \mathbf{s}) + \sigma \mathbf{s}_3$$

$$= \gamma^3 \mathbf{x} + \gamma \sqrt{\gamma^2 \sigma^2 + \sigma^2} \mathbf{s} + \sigma \mathbf{s}_3$$

$$= \gamma^3 \mathbf{x} + \sqrt{\gamma^4 \sigma^2 + \gamma^2 \sigma^2 + \sigma^2} \mathbf{s}'$$

$$\mathbf{z}_t = \gamma^t \mathbf{x} + \sigma \mathbf{s} \quad \text{with} \quad \sigma^2 = \sigma \sum_{i=0}^{t-1} \gamma^{2i}$$



To work this, we will write down the first couple of \mathbf{z}_t , and rewrite them in terms of \mathbf{x} . To do this, we can apply the same principle that we used to show the chaining property was true. We write each \mathbf{z} as an affine transformation of the previous \mathbf{z} and some Gaussian noise \mathbf{s} . We then plug in the definition of the previous \mathbf{z} , and work this into a function of \mathbf{x} and a single standard noise vector \mathbf{s} .

Remember, to combine two standard noise vectors \mathbf{s}_1 and \mathbf{s}_2 , we square their variances, sum, and square-root to get the variance for a single standard noise vector \mathbf{s} .

By \mathbf{z}_3 , the pattern (hopefully) becomes clear: the scalar in front of \mathbf{x} is just γ to the power of t . The variance is a sum with t terms, where each term is our chosen standard deviation σ , times γ to the power of i , where i increases in steps of 2.

WHAT γ AND σ ENSURE CONVERGENCE TO $N(0, I)$?

$$z_t = \gamma^t + \bar{\sigma} s \text{ with } \bar{\sigma}^2 = \sigma^2 \sum_{i=0}^{t-1} \gamma^{2i}$$

$$1 = \sigma^2 \sum_{i=0}^{t-1} \gamma^{2i}$$

$$= \sigma^2 \frac{1}{1 - \gamma^2}$$

$$\sigma^2 = 1 - \gamma^2$$

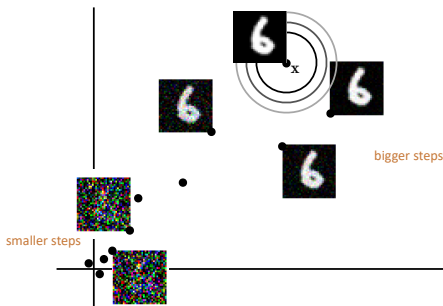
$$\sum_{i=0}^t a^i = \frac{1}{1-a} \text{ if } a \in (0, 1)$$



With this, we can work out what to set σ to so that the variance of z_t converges to 1.

They key is to recognize that the sum on the right-hand-side is a standard geometric sum in for γ^2 which a closed form expression is known. With that, we can rewrite and figure out what σ^2 should be.

SCHEDULE γ



In practice, it turns out that it's better to take big steps at the start of the diffusion process, and smaller steps at the end. This is because denoising is easier if you can see more of the image. That mean that the start of the process, we can ask the mode to remove more noise, which we achieve by taking bigger steps there. This means we can take smaller steps at the end, when it's more difficult for the model to figure out what is "behind" the noise.

$$\gamma_1 > \gamma_2 > \dots > \gamma_T$$

$$z_t = \gamma_t z_t + \sigma_t s$$

$$\sigma_t^2 \stackrel{?}{=} 1 - \gamma_t^2$$



We achieve this by setting a noise *schedule*. We use a different value for each timestep, and ensure that these decrease with t .

If all of these are smaller than 1, we can be sure that the process still converges to mean 0, but what should we set the variance to at each step to ensure that we get a standard Gaussian at the end? A lucky guess might be the same value we got earlier: one minus the square of γ , except now to do this at each timestep for the specific γ_t we used.

Let's see if that guess pays off.

$$\begin{aligned}
z_1 &= \gamma_1 x + \sigma_1 s_1 \\
z_2 &= \gamma_2 z_1 + \sigma_2 s_2 \\
z_2 &= \gamma_2(\gamma_1 x + \sigma_1 s_1) + \sigma_2 s_2 = \gamma_1 \gamma_2 x + \gamma_1 \sigma_1 s_1 + \sigma_2 s_2 \\
&= \gamma_1 \gamma_2 x + \sqrt{\gamma_1^2 \sigma_1^2 + \sigma_2^2} s \\
z_3 &= \gamma_1 \gamma_2 \gamma_3 x + \sqrt{\sigma_1^2 \gamma_2^2 \gamma_3^2 + \sigma_2^2 \gamma_3^2 + \sigma_3^2} s \\
z_t &= \bar{\gamma}_t x + \bar{\sigma} s \\
&\text{with } \bar{\gamma}_t = \gamma_1 \gamma_2 \cdots \gamma_t \\
&\text{and } \bar{\sigma}^2 = \sum_{i=0}^{t-1} \sigma_i^2 \gamma_{i+1}^2 \cdots \gamma_t^2
\end{aligned}$$



We proceed as before, by writing out the definitions of the first few z_t , and plugging the previous z_t into the definition of the next.

The result is more complex, but not by much.

To get the mean of z_t , x is scaled by the product of all γ_t so far. The scalar in front of s is once again a sum of terms. Each term i consists of the variance use at time i times all the square of the γ_t 's used after time i .

$$\begin{aligned}
z_t &= \bar{\gamma}_t x + \bar{\sigma} s \\
&\text{with } \bar{\gamma}_t = \gamma_1 \gamma_2 \cdots \gamma_t \\
&\text{and } \bar{\sigma}^2 = \sum_{i=0}^{t-1} \sigma_i^2 \gamma_{i+1}^2 \cdots \gamma_t^2 \\
(1 - \gamma_1^2)^2 \gamma_2^2 \gamma_3^2 \gamma_4^2 &+ (1 - \gamma_2^2) \gamma_3^2 \gamma_4^2 + (1 - \gamma_3^2)^2 \gamma_4^2 + (1 - \gamma_4^2) \\
\gamma_2^2 \gamma_3^2 \gamma_4^2 &- \gamma_1^2 \gamma_2^2 \gamma_3^2 \gamma_4^2 \\
\gamma_3^2 \gamma_4^2 &- \gamma_2^2 \gamma_3^2 \gamma_4^2 \\
\gamma_4^2 &- \gamma_3^2 \gamma_4^2 \\
1 &- \gamma_4^2 \\
= 1 - \gamma_1^2 \gamma_2^2 \cdots \gamma_t^2 &= 1 - \bar{\gamma}_t^2
\end{aligned}$$



Finally, we need to show that if we set each to the value of one minus the square of the gamma used at t , we get the desired result (a variance that goes to 1).

The key is to realize that if we multiply out the brackets, we get a *telescoping sequence*. We show it here for $t=4$, but it works for all t . Each term before we multiply our the brackets yields two terms afterwards one with the squared γ_t 's from i to t and one with the squared γ_t 's from $i+1$ to t . Since i increments by 1 each term, the first term we create by multiplying our one set of brackets will be equal to the second term we generate by multiplying out the next set of brackets. This is true for every set of brackets, so that we end up only with the final 1 and the full sequences of squared γ_t 's from the first term.

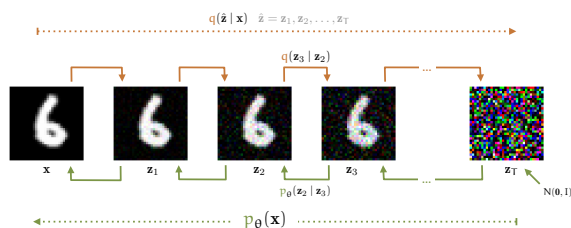
This tells us that the variance of z_t can be expressed as one minus the square of the product of γ_t all so far, which goes to 0, so the variance goes to 1.

$$\begin{aligned}
z_t &= \gamma_t z_{t-1} + \sqrt{1 - \gamma_t^2} s && \leftarrow \text{affects all pixels in the image} \\
z_t &= \bar{\gamma}_t x + \sqrt{1 - \bar{\gamma}_t^2} s && \leftarrow \text{easily sampled at arbitrary } t \\
&\text{with } \bar{\gamma}_t = \gamma_1 \gamma_2 \cdots \gamma_t \\
z_T &\rightarrow N(0, I) \text{ as } T \rightarrow \infty && \leftarrow \text{converges to a known distribution}
\end{aligned}$$



So there we have it: a noise step for our diffusion process that generally affects all pixels to some extent, that can be easily sampled at time t and that converges to standard noise as t increases.

$$\arg \min_{\theta} -\log p_{\theta}(\mathbf{x})$$



The next thing we want is a *principled loss*. Instead of just appealing to intuition, and minimizing the squared distance between two fairly arbitrarily chosen images, we are going to start with a well-chosen principle, and derive from this a loss that we can optimize.

The principle we choose, just as we did with the VAE, is *maximum likelihood*: we should try to set the parameters of our UNet so that the probability of our data is as high as possible.

And that high probability density should transfer to our validation and test data (so that we are not overfitting).

To keep things simple we will

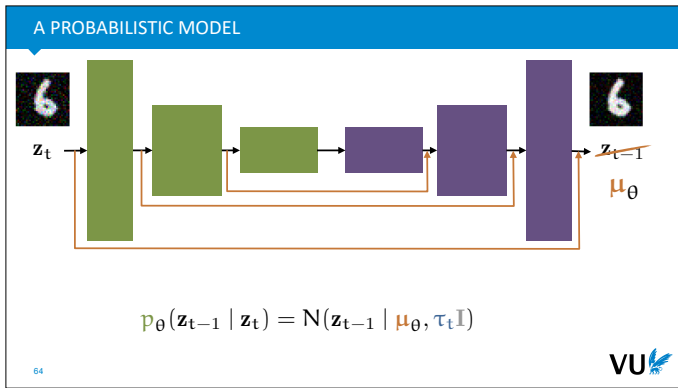
- Only state the maximum likelihood objective for one instance x . It would be more correct to maximize the likelihood for the whole dataset, or even better to maximize the expected likelihood of one instance sampled from the data distribution, but there we will just do what we always do in deep learning: loop over the individual instances (or batches of instances) and follow the gradient of the loss for these with stochastic gradient descent. In short, this part of the approach is exactly the same for generative models as it is for classification or regression, so we will start with the maximum likelihood objective for a single instance.
- Take the negative log-likelihood as a starting point. As we noted before, we put a logarithm in front of a probability (density) to help with the math and the numerical stability, and we put a minus in front because we are looking for a quantity to *minimize*.

The next question is, what is the probability of x under our model? In this view the entire denoising chain, from sampling z_T all the way to the final x is a process that we call p . p is controlled by the parameters of our UNet, which we collectively call θ . Our job is to set θ so that this whole sampling chain is as likely as possible to generate our training, validation and test data.

Note the subtle, but fundamental change in perspective. Before we started by assuming that there was a diffusion process. Now, we never refer to it in our objective. If we could optimize this objective function directly, we would never need to worry about the existence of a diffusion process at all. All we would need, would be the denoising process.

As it is, we cannot easily optimize this function directly, so we bring in the diffusion process, and call it q .

This is similar to the way we set up a VAE: we start with just a generator network (also known as the decoder) and we bring in a second network to help us optimize it.



To fit this picture we need to slightly adapt our UNet, or at least our perspective on it. The UNet should now compute the function $p(z_{t-1} | z_t)$. That is, instead of giving us a single guess for what picture preceded z_t , it gives us a probability distribution on the whole space of possible images.

Practically, this isn't as involved as it sounds. The simplest way to achieve this is to treat the output of the UNet as a mean of a probability distribution. We will combine this with a variance to get a spherical Gaussian on z_{t-1} . τ_t is a hyperparameter that we schedule manually.

It's not too difficult to also have the network output a variance, as we do in the VAE. This is often done as well, but in Ho et al, which we follow here, the UNet only outputs a mean.

AIMS

what we want to optimize:	ingredients:
$\arg \min_{\theta} -\log p_{\theta}(\mathbf{x})$	(continuous) expectation
functions we can compute:	Jensen's inequality:
$p_{\theta}(z_t z_{t+1})$ (our model)	$\log E_{\mathbf{x}} f(\mathbf{x}) \geq E_{\mathbf{x}} \log f(\mathbf{x})$
$p_{\theta}(z_T) = N(\mathbf{0}, \mathbf{I})$	
$q(z_t z_{t-1}) = N(z_t \gamma_t z_{t-1}, 1 - \gamma_t^2)$	
$q(z_t \mathbf{x}) = N(z_t \gamma_t z_{t-1}, 1 - \gamma_t^2)$	

VU

So, here's the situation: we want to rewrite the negative log probability density of the data into something we can compute and that we can backpropagate through. The functions that we can easily compute are

- under p (our model):
 - the density of z_t conditioned on z_{t+1}
 - the density of z_T (the last step in our diffusion process) which we assume to be standard Gaussian noise.
- under q (the diffusion process which we will bring in)
 - the density of z_t conditioned on z_{t-1}
 - the density of z_t conditioned on \mathbf{x}

In our derivation, we will make use of the properties of continuous expectation, and of Jensen's inequality. The latter is a general rule about what happens when you take the sum of points on a concave function, versus summing first and then applying the function. In our case, we need a specific consequence of Jensen's inequality, which is that if we are working with the logarithm, a concave function, of an expectation, a sum or integral, then moving the logarithm inside the expectation changes the value, but it makes it strictly larger.

We will use this in the same way we did in the VAE. We can't work this loss into something we can compute, but we can compute a lower bound using Jensen's inequality. Maximizing the lower bound (or minimizing its negative) will, if the lower bound is good enough, also maximize the thing we're lower-bounding.

$$\begin{aligned}
(1) \quad -\log p(\mathbf{x}) &= -\log \int_{\hat{\mathbf{z}}} p(\mathbf{x}, \hat{\mathbf{z}}) d\hat{\mathbf{z}} \\
(2) \quad &= -\log \int_{\hat{\mathbf{z}}} p(\mathbf{x} | \mathbf{z}_1) p(\mathbf{z}_1 | \mathbf{z}_2) \cdots p(\mathbf{z}_{T-1} | \mathbf{z}_T) p(\mathbf{z}_T) d\hat{\mathbf{z}} = \int_{\hat{\mathbf{z}}} p(\mathbf{z}_T) \prod_t p(\mathbf{z}_{t-1} | \mathbf{z}_t) d\hat{\mathbf{z}} \\
(3) \quad &= -\log \int_{\hat{\mathbf{z}}} \frac{q(\hat{\mathbf{z}} | \mathbf{x})}{q(\hat{\mathbf{z}} | \mathbf{x})} p(\mathbf{z}_T) \prod_t p(\mathbf{z}_{t-1} | \mathbf{z}_t) d\hat{\mathbf{z}} \\
(4) \quad &= -\log \int_{\hat{\mathbf{z}}} q(\hat{\mathbf{z}} | \mathbf{x}) p(\mathbf{z}_T) \prod_t \frac{p(\mathbf{z}_{t-1} | \mathbf{z}_t)}{q(\mathbf{z}_{t-1} | \mathbf{z}_t)} d\hat{\mathbf{z}} \\
(5) \quad &= -\log E_{\hat{\mathbf{z}} \sim q(\hat{\mathbf{z}} | \mathbf{x})} p(\mathbf{z}_T) \prod_t \frac{p(\mathbf{z}_{t-1} | \mathbf{z}_t)}{q(\mathbf{z}_{t-1} | \mathbf{z}_t)} \\
(6) \quad &\leq E_{\hat{\mathbf{z}} \sim q(\hat{\mathbf{z}} | \mathbf{x})} -\log p(\mathbf{z}_T) \prod_t \frac{p(\mathbf{z}_{t-1} | \mathbf{z}_t)}{q(\mathbf{z}_{t-1} | \mathbf{z}_t)} \\
(7) \quad &= E_{\hat{\mathbf{z}} \sim q(\hat{\mathbf{z}} | \mathbf{x})} -\log p(\mathbf{z}_T) - \sum_t \log p(\mathbf{z}_{t-1} | \mathbf{z}_t) - \log q(\mathbf{z}_t | \mathbf{z}_{t-1}) \\
(8) \quad &= E_{\hat{\mathbf{z}} \sim q(\hat{\mathbf{z}} | \mathbf{x})} - \sum_t \log p(\mathbf{z}_{t-1} | \mathbf{z}_t) + c \\
(9) \quad &= E_{\hat{\mathbf{z}} \sim q(\hat{\mathbf{z}} | \mathbf{x})} \sum_t \frac{1}{2\tau_t} \|\mathbf{z}_{t-1} - \mu_{\theta}\|^2 + c
\end{aligned}$$

Here is our full derivation. Step by step:

1: We can get to a particular \mathbf{x} by many different sequences of denoising steps. If we call this sequence of intermediate pictures $\hat{\mathbf{z}}$, then the total probability is the probability of \mathbf{x} given a particular $\hat{\mathbf{z}}$, integrated over all possible $\hat{\mathbf{z}}$. In probability theory jargon: we start with the joint probability of \mathbf{x} and the $\hat{\mathbf{z}}$ that generated it, and then we marginalize out $\hat{\mathbf{z}}$ to get our loss.

2: We use the chain rule of probability to decompose the joint probability into the product of all probabilities of \mathbf{z}_t given its successor. This is a very common approach in sequence modeling. If you've never seen it before, have a look at [these slides](#).

3: We feed in the diffusion process. There is not much intuition here, except that we're allowed to do this because the factor we add in is equal to 1. Note that this probability is slightly different than the one under p , because here \mathbf{x} only appears in the conditional. The reason is that we want q to correspond to our diffusion process in which \mathbf{x} is given and we sample $\hat{\mathbf{z}}$.

4: We move the numerator out in front, and the denominator we decompose in the same way as we did with p . This gives us a factor $q(\mathbf{z}_t | \mathbf{z}_{t-1})$ for every factor $p(\mathbf{z}_{t-1} | \mathbf{z}_t)$. I recommend trying this with a short sequence on pen and paper, to see that it works out this way.

5: The probability next to the integral shows that we have created an *expectation*. Specifically, the expected value of some probabilities multiplied together, under our diffusion process. This is relevant, because we can estimate expectations by sampling, and we can easily sample from our diffusion process. In short, this is starting to look like something we can estimate.

6: Just as with the VAE, we use Jensen's inequality to move the logarithm inside the expectation. This turns the quantity we're working out into an upper bound of the quantity we actually want to minimize. *This is called the evidence lower bound or ELBO (it's a lower bound if you take the perspective that you're maximizing the probability of the data).*

7, 8: Next, we work out all the log factors separately. In our case, only the terms containing $p(\mathbf{z}_{t-1} | \mathbf{z}_t)$ are affected by the parameters. The rest are constants, so we can ignore them without affecting the gradients we ultimately get.

9: We open up the formula for the Gaussian that our model outputs. Because of the logarithm and our assumption that the variance τ_t is constant, the only term that is influenced by the parameters is the squared euclidean distance. The multiplicative constant should be included if we're strict (since it's different for different t), but it's often found that the algorithm performs better if we ignore it and tune the learning rate carefully.

OUR LOSS (V1)

$$\sum_t E_{z_{t-1}, z_t \sim q(\cdot|x)} \|z_{t-1} - \mu_\theta(z_t)\|^2$$

pick a random t sample z_{t-1}, z_t from q minimize sq. distance between model output and z_{t-1}

And the end of all our exploring
Will be to arrive where we started
And know the place for the first time.
—T.S. Eliot



If we move the sum outside the expectation, a very simple algorithm presents itself. We will take this sum of expectations and optimize one term at a time. That is, we pick a random t and optimize only the term corresponding to t .

In that case, the expectation over the whole sequence \mathbf{z} reduces to the marginal distribution on z_{t-1} and z_t . These we can sample easily to estimate the expectation.

Finally, we just compute the squared Euclidean distance between z_{t-1} and the model output and backpropagate that.

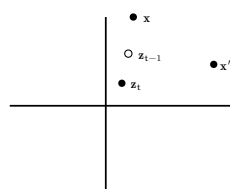
This is, of course, what we were doing already. You may ask what the point of all this is, if we just end up where we started, when the original idea was already obvious. One benefit of this derivation, is that it tells us how to adapt the model in a rigorous way. We can, for instance, decide not to ignore the variance of the distribution that the model outputs. Or, we can make that variance an output of the model instead of a hyperparameter. In such cases, all we have to do is follow the derivation and see how the loss function changes as a result.

V1 work, but suffers from high variance.

What if we could compare $p(z_{t-1} | z_t)$ to $q(z_{t-1} | z_t)$?

NB q is in the opposite direction of the diffusion process.

$q(z_{t-1} | z_t)$ is intractable ...



but $q(z_{t-1} | z_t, x)$ isn't.

In fact, it's a Gaussian.

68



The version of our algorithm in the last slide was used in early diffusion models. It worked, but it suffered from high variance: that is, the loss differed a lot from one sample to the next. This isn't surprising: we do a lot of sampling, and everything we sample to estimate some expectation that we want to compute, increases the variance. There is one place where we can replace a sample with a more direct computation.

In the algorithm we have now, we sample z_{t-1} and then we maximize the probability density of z_{t-1} under p (given z_t). What if, instead, we could work out the probability distribution that q puts on z_{t-1} given z_t and just tell our model to get as close to that distribution as possible? Comparing two probability distributions to each other should lead to less variance than comparing one probability distribution to a sample from another distribution.

The problem is that $q(z_{t-1} | z_t)$ is the opposite direction of how we can easily compute q . We can try to flip the conditional around with Bayes' rule, but it turns out this doesn't lead to a tractable distribution. To see

why, imagine that the data can come from two points \mathbf{x} and \mathbf{x}' as shown in the image. What we want to know is the distribution on \mathbf{z}_{t-1} given \mathbf{z}_t . If we started with \mathbf{x} , then this \mathbf{z}_{t-1} is very likely, since the diffusion process mostly follows a straight line towards the origin. However, if we start with \mathbf{x}' , then all the probability mass is on the line between \mathbf{x}' and the origin, and this \mathbf{z}_{t-1} is very unlikely.

The takeaway is that the distribution $q(\mathbf{z}_{t-1}|\mathbf{z}_t)$ is highly dependent on the shape of the distribution on \mathbf{x} . This distribution is first of all highly complex, and second of all unknown (it's what we're trying to learn).

However, what we also see is that if we condition on which \mathbf{x} we started with, the resulting distribution $q(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{x})$ is actually quite simple. In fact, with a little analysis, it turns out that this distribution is a Gaussian.

IT TURNS OUT... (TAKE MY WORD FOR IT)

$$q(\mathbf{z}_{t-1} | \mathbf{z}_t, \mathbf{x}) = N(\tilde{\boldsymbol{\mu}}_{t-1}, \tilde{\boldsymbol{\sigma}}_{t-1}^2) \text{ with}$$

$$\tilde{\boldsymbol{\mu}}_{t-1} = \frac{1}{\gamma_t} \left(\mathbf{z}_t - \frac{1 - \gamma_t}{\sqrt{1 - \gamma_t}} \mathbf{s}_t \right)$$

$$\tilde{\boldsymbol{\sigma}}_{t-1}^2 = \frac{1 - \gamma_{t-1}}{1 - \gamma_t} (1 - \gamma_t^2)$$

\mathbf{s}_t : standard normal vector used to generate \mathbf{z}_t from \mathbf{x}



The derivation itself is a little complex so we'll skip it here, but these are the parameters of the resulting Gaussian.

Note that the mean looks like a random value itself, but that isn't what's happening here. What we have done is to remove \mathbf{x} from the formula by remembering the specific gaussian noise \mathbf{s}_t that we sampled when we generated \mathbf{z}_t . So, what we are looking at is just a function of know values. In other words, this function doesn't work for general \mathbf{z}_t and \mathbf{x} , but it we generated \mathbf{z}_t from \mathbf{x} (as we do in sampling from our diffusion process), then we can use this formula if we remember the specific Gaussian noise we used.

So now we have a distribution to compare our model's output distribution to, not just a single point. But how do we work this into the loss? We wanted to work from first principles, so we can't just start comparing these two distributions arbitrarily. We will have to go back to our derivation, and take a slightly different route.

$$\begin{aligned} (1) \quad -\log p(\mathbf{x}) &= -\log E_{\mathbf{z} \sim q(\cdot|\mathbf{x})} p(\mathbf{z}_T) \prod_t \frac{p(\mathbf{z}_{t-1} | \mathbf{z}_t)}{q(\mathbf{z}_{t-1} | \mathbf{z}_t, \mathbf{x})} \\ (2) \quad &= -\log E_{\mathbf{z}} p(\mathbf{z}_T) \prod_{t=1}^T \frac{p(\mathbf{z}_{t-1} | \mathbf{z}_t)}{q(\mathbf{z}_{t-1} | \mathbf{z}_t, \mathbf{x})} \\ (3) \quad &= -\log E_{\mathbf{z}} p(\mathbf{z}_T) \frac{p(\mathbf{x} | \mathbf{z}_1)}{q(\mathbf{z}_1 | \mathbf{x})} \prod_{t=2}^T \frac{p(\mathbf{z}_{t-1} | \mathbf{z}_t)}{q(\mathbf{z}_{t-1} | \mathbf{z}_t, \mathbf{x})} \frac{q(\mathbf{z}_t | \mathbf{x})}{q(\mathbf{z}_t | \mathbf{x})} \\ (4) \quad &= -\log E_{\mathbf{z}} p(\mathbf{z}_T) \frac{p(\mathbf{x} | \mathbf{z}_1)}{q(\mathbf{z}_1 | \mathbf{x})} \prod_{t=2}^T \frac{p(\mathbf{z}_{t-1} | \mathbf{z}_t)}{q(\mathbf{z}_{t-1} | \mathbf{z}_t, \mathbf{x})} \frac{q(\mathbf{z}_t | \mathbf{x})}{q(\mathbf{z}_t | \mathbf{x})} \\ (5) \quad &= -\log E_{\mathbf{z}} p(\mathbf{z}_T) \frac{p(\mathbf{x} | \mathbf{z}_1)}{q(\mathbf{z}_1 | \mathbf{x})} \prod_{t=2}^T \frac{p(\mathbf{z}_{t-1} | \mathbf{z}_t)}{q(\mathbf{z}_{t-1} | \mathbf{z}_t, \mathbf{x})} \frac{q(\mathbf{z}_1 | \mathbf{x})}{q(\mathbf{z}_2 | \mathbf{x})} \frac{q(\mathbf{z}_2 | \mathbf{x})}{q(\mathbf{z}_3 | \mathbf{x})} \frac{q(\mathbf{z}_3 | \mathbf{x})}{q(\mathbf{z}_4 | \mathbf{x})} \frac{q(\mathbf{z}_4 | \mathbf{x})}{q(\mathbf{z}_1 | \mathbf{x})} \\ (6) \quad &= -\log E_{\mathbf{z}} p(\mathbf{x} | \mathbf{z}_1) \frac{p(\mathbf{z}_T)}{q(\mathbf{z}_T | \mathbf{x})} \prod_{t=2}^T \frac{p(\mathbf{z}_{t-1} | \mathbf{z}_t)}{q(\mathbf{z}_{t-1} | \mathbf{z}_t, \mathbf{x})} \text{KL}(a, b) = -E_{\mathbf{x} \sim a} \log \frac{b(\mathbf{x})}{a(\mathbf{x})} \\ (7) \quad &\leq -E_{\mathbf{x}} \log p(\mathbf{x} | \mathbf{z}_1) - E_{\mathbf{x}, \mathbf{z}} \log \frac{p(\mathbf{z}_T)}{q(\mathbf{z}_T | \mathbf{x})} - \sum_{t=2}^T E_{\mathbf{x}_{1:t}, \mathbf{z}_t} \log \frac{p(\mathbf{z}_{t-1} | \mathbf{z}_t)}{q(\mathbf{z}_{t-1} | \mathbf{z}_t, \mathbf{x})} \\ (8) \quad &= -E_{\mathbf{x}} \log p(\mathbf{x} | \mathbf{z}_1) + \text{KL}[q(\mathbf{z}_T | \mathbf{x}), p(\mathbf{z}_T)] + \sum_{t=2}^T \text{KL}[q(\mathbf{z}_{t-1} | \mathbf{z}_t, \mathbf{x}), p(\mathbf{z}_{t-1} | \mathbf{z}_t)] \end{aligned}$$



Here is the derivation.

1: In the first line, we have already followed some steps from the original derivation, including marginalizing in \mathbf{z} , working in q , and turning the integral into an expectation. Note that we have not applied Jensen's inequality yet (the logarithm is still outside the expectation).

2: We note that $q(\mathbf{z}_t | \mathbf{z}_{t-1})$ is equal to $q(\mathbf{z}_t | \mathbf{z}_{t-1}, \mathbf{x})$. This is because \mathbf{z}_t is independent of \mathbf{x} when conditioned on \mathbf{z}_{t-1} . Put simply, if we already know \mathbf{z}_{t-1} then also knowing \mathbf{x} doesn't give us extra information. So adding \mathbf{x} to the conditional doesn't change anything. It will however change things when we use Bayes' rule to flip the conditional around.

3: We separate the first factor from our product. This will help us later on.


4: Next, we flip the conditional in the denominator of our product using Bayes' rule. We flip around \mathbf{z}_t and \mathbf{z}_{t-1} , and keep \mathbf{x} in the conditional for all factors.

5: Note that this extra factor that we got from Bayes' rule "telescopes" over the factors. That is, every z that appears in the numerator as z_{t-1} will appear in the denominator as z_t in the next factor. All of these cancel out, against each other, except the first and the last, which we take out of the sum.

6: We cancel the the two $q(z_1|x)$'s

7: We apply Jensen's and separate out three terms for our loss function as shown.

8: We now recognize that almost all terms are KL divergences. That is, they naturally express the similarity between two Gaussian distributions. Only the first term is a log-probability of x given the final image in our denoising process z_1 .

$$\begin{aligned}
 & \underbrace{-\mathbb{E}_{z_T} \log p(x | z_1)}_{\text{separate decoder model (in Ho et al.)}} + \text{KL}[q(z_T | x), p(z_T)] + \sum_{t=2}^T \text{KL}[q(z_{t-1} | z_t, x), p(z_{t-1} | z_t)] \\
 \\
 & \text{KL}[q(z_{t-1} | z_t, x), p(z_{t-1} | z_t)] = \frac{1}{2\tau_t} \|\tilde{\mu}_{t-1} - \mu_\theta\|^2 + c \\
 \\
 & \text{with: } \tilde{\mu}_{t-1} = \frac{1}{\gamma_t} \left(z_t - \frac{1 - \gamma_t^2}{\sqrt{1 - \gamma_t}} s_t \right) \\
 \\
 & \text{previously: } \|z_{t-1} - \mu_\theta(z_t)\|^2
 \end{aligned}$$


Looking at this loss function, let's see what we have. The middle term doesn't have any parameters, so we can ignore it. The first term is an odd duck. What Ho et al do is to train a separate model for this part, which tells us how to generate x given z_1 . Note that z_1 is almost completely denoised, so this model has an easy job.

The main reason for this separate decoder is that Ho et al. want to evaluate their model in the number of bits it uses to represent an image. This is a great evaluation metric (see [here](#) for an explanation), but it requires a discrete distribution on your space of images, while the Gaussian is continuous. Ho et al. use this extra term as an opportunity to translate to a discrete representation of images.

For the final term (or rather T-2 terms), we can fill in what we worked out earlier about the KL divergence for two spherical Gaussians: as a function of the means the KL divergence is just the (scaled) squared euclidean distance plus some constants that don't affect our gradients.

Ho et al find that they get better results by ignoring the scaling factor, but other authors prefer to include it. We'll ignore it for the sake of simplicity and just minimize the squared Euclidean distance as we did already for the naive diffusion.

What has changed from the previous algorithm? There we also took the output of the model and computed a squared distance. The difference is in what we compute the distance to. Previously, that was the sample z_{t-1} , Now, we compute the distance to the mean of the distribution on z_{t-1} . By not sampling z_{t-1} but looking instead at the mean of its distribution, we are reducing the variance of our loss.

ONE LAST TRICK: PREDICTING s_t INSTEAD OF μ_t

$$\begin{aligned} \text{loss}_\mu &\propto \|\bar{\mu}_{t-1} - \mu_\theta\|^2 \\ &= \left\| \frac{1}{\gamma_t} \left(z_t - \frac{1-\gamma_t^2}{\sqrt{1-\gamma_t^2}} s_t \right) - \mu_\theta \right\|^2 \\ \text{loss}_s &\propto \left\| \frac{1}{\gamma_t} \left(z_t - \frac{1-\gamma_t^2}{\sqrt{1-\gamma_t^2}} s_t \right) - \frac{1}{\gamma_t} \left(z_t - \frac{1-\gamma_t^2}{\sqrt{1-\gamma_t^2}} s_\theta \right) \right\|^2 \\ &\propto \left\| z_t - \frac{1-\gamma_t^2}{\sqrt{1-\gamma_t^2}} s_t - z_t + \frac{1-\gamma_t^2}{\sqrt{1-\gamma_t^2}} s_\theta \right\|^2 \\ &\propto \|s_\theta(z_t) - s_t\|^2 = \|s_t - s_\theta(z_t)\|^2 \end{aligned}$$



We are almost finished, but Ho et al have one last trick up their sleeve. By rewriting a little bit further we can have the model predict the noise that has been added to our image instead of the partially denoised image.

First we fill in our formula for computing the mean on z_{t-1} .

Next, we change our model output from a prediction of this mean, to a prediction of the noise s_t used to generate this mean. From that, we compute the mean by the same formula as we used in the other term.

The factor in front of the noises is the same on both sides, so we can take it out of the loss, and absorb it into the learning rate.

We've written s_θ as $s_\theta(z_t)$ here to emphasize that this is the output of the model when given z_t as an input.

GAUSSIAN DIFFUSION (HO ET AL 2020)

training	sampling
initialize model f	given f
for x in Data:	sample $z_T \sim N(\mathbf{0}, \mathbf{I})$
$t = \text{uniform}(0, T)$	for t from T to 2 :
$s_t \sim N(\mathbf{0}, \mathbf{I})$	sample $z_{t-1} \sim N(\bar{\mu}_t, \tau_t^2)$
$z_t = \gamma_t x + \sqrt{1-\gamma_t^2} s_t$	with $\bar{\mu}_t = \frac{1}{\gamma_t} \left(z_t - \frac{1-\gamma_t^2}{\sqrt{1-\gamma_t^2}} s_\theta(z_t) \right)$
$\text{loss} = \ s_t - f(z_t)\ ^2$	
brackprop & sgd on loss	return z_1 # $p(x z_1)$ not included



With that, here is our second Gaussian diffusion algorithm, which is exactly equal to the one used by Ho et al.

During training, we pick a random point t and sample z_t using some random standard noise s_t . We then have the model predict s_t .

During sampling, we sample z_{t-1} from the distribution $q(z_{t-1} | z_t, x)$ that we worked out earlier. You might think that in this case it's a problem that we don't have x , but that's where our model comes in. We changed the functional dependence on x by rewriting the mean as a function of s_t and this is a value that our model can predict.

The result is a kind of combination of the two algorithms we saw in the first part. There we had two options: we either predict z_{t-1} from z_t directly, or we predict x from z_{t-1} , and we denoise and renoise during sampling. Here, we do neither. We instead predict the noise vector s_t that describes z_t as a function of x . From this we can compute either our best bet for the fully denoised x , or for z_{t-1} . In short the first and second algorithms from the first part of the lecture are the same algorithm, but only if we use Gaussian noise, and we predict s_t .

Efficient sampling (DDIM)

Continuous time models

Conditional generation

Condition generation on the class of the image, or even a text description.

That should give you a basic idea of how Gaussian diffusion works. There are many variations on this idea and all of them require you to follow this derivation and to change some thing here or there, or to rewrite something in a slightly different way. Understanding all these models requires understanding this derivation in detail.

Some important things you may want to look into if you want to continue with diffusion models are:

- Ways to make the sampling more efficient. Ho et al required 1000 steps during sampling, which means 1000 forward passes to generate one batch of images. This puts diffusion models at a substantial disadvantage to VAEs and GANs which can generate images with a single forward pass. Several methods exist which allow us to reduce the number of sampling steps required (possibly at the cost of a slight drop in quality).
- If your sampling algorithm doesn't require you to visit all time steps in the diffusion process, then you can make those steps as small as possible. We saw this idea in part 1 in the denoising/renoising algorithm. In the limit of infinitely small time steps, we get a continuous value t . Theoretically, the models starts to look like a differential equation, but practically, not much changes.
- Finally, the most impressive uses of diffusion models are those that condition the generation on something, in particular on a text description of what the image should contain. Practically, this is as simple as feeding the text description to an LLM to get an embedding and then to add that embedding to the inputs of the model. However, it turns out that it's good to have a way to control how much the model tries to make the image fit the data distribution, and how much it tries to make it fit the description. This is usually done with a method called *classifier-free guidance*.