# Advanced machine learning and data analysis for the physical sciences

Morten Hjorth-Jensen[1,2]

Department of Physics and Center for Computing in Science Education, University of Oslo, Norway[1]

Department of Physics and Astronomy and Facility for Rare Isotope Beams, Michigan State University, East Lansing, Michigan, USA[2]

May 7, 2024

# Plans for the week of May 6-10, 2024

## Generative models

1. Finalizing discussion of Generative Adversarial Networks
2. Mathematics of diffusion models and selected examples
3. Whiteboard notes

## Reading on diffusion models

1. A central paper is the one by Sohl-Dickstein et al, Deep Unsupervised Learning using Nonequilibrium Thermodynamics, https://arxiv.org/abs/1503.03585
2. See also Diederik P. Kingma, Tim Salimans, Ben Poole, Jonathan Ho, Variational Diffusion Models, https://arxiv.org/abs/2107.00630

# Reminder from last week, what is a GAN?

A GAN is a deep neural network which consists of two networks, a so-called generator network and a discriminating network, or just discriminator. Through several iterations of generation and discrimination, the idea is that these networks will train each other, while also trying to outsmart each other.
In its simplest version, the two networks could be two standard neural networks with a given number of hidden of hidden layers and parameters to train. The generator we have trained can then be used to produce new images.

# Labeling the networks

For a GAN we have:

1. a discriminator $D$ estimates the probability of a given sample coming from the real dataset. It attempts at discriminating the trained data by the generator and is optimized to tell the fake samples from the real ones (our data set). We say a discriminator tries to distinguish between real data and those generated by the abovementioned generator.
2. a generator $G$ outputs synthetic samples given a noise variable input $z$ ($z$ brings in potential output diversity). It is trained to capture the real data distribution in order to generate samples that can be as real as possible, or in other words, can trick the discriminator to offer a high probability.

At the end of the training, the generator can be used to generate for example new images. In this sense we have trained a model which can produce new samples. We say that we have implicitely defined a probability.

# Which data?

**GANs are generally a form of unsupervised machine learning**, although they also incorporate aspects of supervised learning. Internally the discriminator sets up a supervised learning problem. Its goal is to learn to distinguish between the two classes of generated data and original data. The generator then considers this classification problem and tries to find adversarial examples, that is samples which will be misclassified by the discriminator.

# Semi-supervised learning

One can also design GAN architectures which work in a semi-supervised learning setting. A semi-supervised learning environment includes both labeled and unlabeled data. See https://proceedings.neurips.cc/paper_files/paper/2016/file/8a3363abe792db2d8761d6403605aeb7-Paper.pdf for a further discussion.
Thus, GANs can be used both on labeled and on unlabeled data and are used in three most commonly used contexts, that is

1. with labeled data (supervised training)
2. with unlabeled data (unsupervised learning)
3. a with a mix labed and unlabeled data

# Improving functionalities

These two models compete against each other during the training process: the generator $G$ is trying hard to trick the discriminator, while the critic model $D$ is trying hard not to be cheated. This interesting zero-sum game between two models motivates both to improve their functionalities.

# Setup of the GAN

We define a probability $p_{\boldsymbol{h}}$ which is used by the generator. Usually it is given by a uniform distribution over the input $\boldsymbol{h}$. Thereafter we define the distribution of the generator which we want to train, $p_g$ This is the generator's distribution over the data $\boldsymbol{x}$. Finally, we have the distribution $p_r$ over the real sample $\boldsymbol{x}$

# Optimization part

On one hand, we want to make sure the discriminator $D$'s decisions over real data are accurate by maximizing $\mathbb{E}_{\boldsymbol{x} \sim p_r(\boldsymbol{x})}[\log D(\boldsymbol{x})]$.
Meanwhile, given a fake sample $G(\boldsymbol{h}), \boldsymbol{h} \sim p_{\boldsymbol{h}}(\boldsymbol{h})$, the discriminator is expected to output a probability, $D(G(\boldsymbol{h}))$, close to zero by maximizing $\mathbb{E}_{\boldsymbol{h} \sim p_{\boldsymbol{h}}(\boldsymbol{h})}[\log(1 - D(G(\boldsymbol{h})))]$.
On the other hand, the generator is trained to increase the chances of $D$ producing a high probability for a fake example, thus to minimize $\mathbb{E}_{\boldsymbol{h} \sim p_{\boldsymbol{h}}(\boldsymbol{h})}[\log(1 - D(G(\boldsymbol{h})))]$.

# Minimax game

When combining both aspects together, $D$ and $G$ are playing a **minimax game** in which we should optimize the following loss function:

$$\min_G \max_D L(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_r(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{h} \sim p_{\boldsymbol{h}}(\boldsymbol{h})}[\log(1 - D(G(\boldsymbol{h})))]$$
$$= \mathbb{E}_{\boldsymbol{x} \sim p_r(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{x} \sim p_g(\boldsymbol{x})}[\log(1 - D(\boldsymbol{x})]$$

where $\mathbb{E}_{\boldsymbol{x} \sim p_r(\boldsymbol{x})}[\log D(\boldsymbol{x})]$ has no impact on $G$ during gradient descent updates.

# Optimal value for $D$

Now we have a well-defined loss function. Let's first examine what is the best value for $D$.

$$L(G, D) = \int_{\boldsymbol{x}} \Big( p_r(\boldsymbol{x}) \log(D(\boldsymbol{x})) + p_g(\boldsymbol{x}) \log(1 - D(\boldsymbol{x})) \Big) d\boldsymbol{x}$$

# Best value of $D$

Since we are interested in what is the best value of $D(\boldsymbol{x})$ to maximize $L(G, D)$, let us label

$$\tilde{\boldsymbol{x}} = D(\boldsymbol{x}), A = p_r(\boldsymbol{x}), B = p_g(\boldsymbol{x})$$

# Integral evaluation

The integral (we can safely ignore the integral because $\boldsymbol{x}$ is sampled over all the possible values) is:

$$
\begin{aligned}
f(\tilde{\boldsymbol{x}}) &= A \log \tilde{\boldsymbol{x}} + B \log (1 - \tilde{\boldsymbol{x}}) \\
\frac{df(\tilde{\boldsymbol{x}})}{d\tilde{\boldsymbol{x}}} &= A \frac{1}{\tilde{\boldsymbol{x}}} - B \frac{1}{1 - \tilde{\boldsymbol{x}}} \\
&= \frac{A - (A + B)\tilde{\boldsymbol{x}}}{\tilde{\boldsymbol{x}}(1 - \tilde{\boldsymbol{x}})}.
\end{aligned}
$$

# Best values

If we set

$$\frac{df(\tilde{\boldsymbol{x}})}{d\tilde{\boldsymbol{x}}} = 0,$$

we get the best value of the discriminator:

$$D^*(\boldsymbol{x}) = \tilde{\boldsymbol{x}}^* = \frac{A}{A+B} = \frac{p_r(\boldsymbol{x})}{p_r(\boldsymbol{x}) + p_g(\boldsymbol{x})} \in [0, 1].$$

Once the generator is trained to its optimal, $p_g$ gets very close to $p_r$. When $p_g = p_r$, $D^*(\boldsymbol{x})$ becomes $1/2$. We will observe this when running the code from last week (see jupyter-notebook from week 15).

# At their optimal values

When both $G$ and $D$ are at their optimal values, we have $p_g = p_r$ and $D^*(\boldsymbol{x}) = 1/2$, the loss function becomes

$$
\begin{aligned}
L(G, D^*) &= \int_{\boldsymbol{x}} \Big( p_r(\boldsymbol{x}) \log(D^*(\boldsymbol{x})) + p_g(\boldsymbol{x}) \log(1 - D^*(\boldsymbol{x})) \Big) d\boldsymbol{x} \\
&= \log \frac{1}{2} \int_{\boldsymbol{h}} p_r(\boldsymbol{x}) d\boldsymbol{x} + \log \frac{1}{2} \int_{\boldsymbol{x}} p_g(\boldsymbol{x}) d\boldsymbol{x} \\
&= -2 \log 2
\end{aligned}
$$

# What does the Loss Function Represent?

The JS divergence between $p_r$ and $p_g$ can be computed as:

$$
\begin{aligned}
D_{JS}(p_r \| p_g) =& \frac{1}{2} D_{KL}(p_r || \frac{p_r + p_g}{2}) + \frac{1}{2} D_{KL}(p_g || \frac{p_r + p_g}{2}) \\
=& \frac{1}{2} \left( \log 2 + \int_x p_r(\boldsymbol{x}) \log \frac{p_r(\boldsymbol{x})}{p_r + p_g(\boldsymbol{x})} d\boldsymbol{x} \right) + \\
& \frac{1}{2} \left( \log 2 + \int_x p_g(\boldsymbol{x}) \log \frac{p_g(\boldsymbol{x})}{p_r + p_g(\boldsymbol{x})} d\boldsymbol{x} \right) \\
=& \frac{1}{2} \left( \log 4 + L(G, D^*) \right)
\end{aligned}
$$

# What does the loss function quantify?

We have

$$L(G, D^*) = 2D_{JS}(p_r \| p_g) - 2\log 2.$$

The loss function of GANs quantifies the similarity between the generative data distribution $p_g$ and the real sample distribution $p_r$ by the so-called JS divergence when the discriminator is optimal. The best $G^*$ that replicates the real data distribution leads to the minimum $L(G^*, D^*) = -2\log 2$.

# Problems with GANs

Although GANs have achieved great success in the generation of realistic images, the training is not easy; The process is known to be slow and unstable.

## Hard to reach equilibrium.

Two models are trained simultaneously to an equilibrium to a two-player non-cooperative game. However, each model updates its cost independently with no respect to another player in the game. Updating the gradient of both models concurrently cannot guarantee a convergence.

# Vanishing Gradient

When the discriminator is perfect, we are guaranteed with $D(\boldsymbol{x}) = 1, \forall \boldsymbol{x} \in p_r$ and $D(\boldsymbol{x}) = 0, \forall \boldsymbol{x} \in p_g$.
Then, the loss function $L$ falls to zero and we end up with no gradient to update the loss during learning iterations. One can encouter situations where the discriminator gets better and the gradient vanishes fast.
As a result, training GANs may face the following problems

1. If the discriminator behaves badly, the generator does not have accurate feedback and the loss function cannot represent the real data
2. If the discriminator does a great job, the gradient of the loss function drops down to close to zero and the learning can become slow

# Improved GANs

One of the solutions to improved GANs training, is the introduction of what is called the Wasserstein diatance, which is a way to compute the difference/distance between two probability distribitions. For those interested in reading more, we recommend for example chapter 17 of Rashcka's et al textbook, Machine Learning with PyTorch and Scikit-Learn, chapter 17, see https://github.com/rasbt/python-machine-learning-book-3rd-edition/tree/master/ch17
For a definition of the Wasserstein distance, see for example https://arxiv.org/pdf/2103.01678

# More codes

1. For codes and background, see Raschka et al, Machine Learning with PyTorch and Scikit-Learn, chapter 17, see https://github.com/rasbt/python-machine-learning-book-3rd-edition/tree/master/ch17 for codes
2. Babcock and Bali, Generative AI with Python and TensorFlow2, chapter 6 and codes at https://github.com/raghavbali/generative_ai_with_tensorflow/blob/master/Chapter_6/conditional_gan.ipynb
3. See also Foster's text Generative Deep Learning and chapter 4 with codes at https://github.com/davidADSP/Generative_Deep_Learning_2nd_Edition/tree/main/notebooks/04_gan

# Diffusion models, basics

Diffusion models are inspired by non-equilibrium thermodynamics. They define a Markov chain of diffusion steps to slowly add random noise to data and then learn to reverse the diffusion process to construct desired data samples from the noise. Unlike VAE or flow models, diffusion models are learned with a fixed procedure and the latent variable has high dimensionality (same as the original data).

# Problems with probabilistic models

Historically, probabilistic models suffer from a tradeoff between two conflicting objectives: *tractability* and *flexibility*. Models that are *tractable* can be analytically evaluated and easily fit to data (e.g. a Gaussian or Laplace). However, these models are unable to aptly describe structure in rich datasets. On the other hand, models that are *flexible* can be molded to fit structure in arbitrary data. For example, we can define models in terms of any (non-negative) function $\phi(\boldsymbol{x})$ yielding the flexible distribution $p(\boldsymbol{x}) = \frac{\phi(\boldsymbol{x})}{Z}$, where $Z$ is a normalization constant. However, computing this normalization constant is generally intractable. Evaluating, training, or drawing samples from such flexible models typically requires a very expensive Monte Carlo process.

# Diffusion models

Diffusion models have several interesting features

- extreme flexibility in model structure,
- exact sampling,
- easy multiplication with other distributions, e.g. in order to compute a posterior, and
- the model log likelihood, and the probability of individual states, to be cheaply evaluated.

# Original idea

In the original formulation, one uses a Markov chain to gradually convert one distribution into another, an idea used in non-equilibrium statistical physics and sequential Monte Carlo. Diffusion models build a generative Markov chain which converts a simple known distribution (e.g. a Gaussian) into a target (data) distribution using a diffusion process. Rather than use this Markov chain to approximately evaluate a model which has been otherwise defined, one can explicitly define the probabilistic model as the endpoint of the Markov chain. Since each step in the diffusion chain has an analytically evaluable probability, the full chain can also be analytically evaluated.

# Diffusion learning

Learning in this framework involves estimating small perturbations to a diffusion process. Estimating small, analytically tractable, perturbations is more tractable than explicitly describing the full distribution with a single, non-analytically-normalizable, potential function. Furthermore, since a diffusion process exists for any smooth target distribution, this method can capture data distributions of arbitrary form.

# Mathematics of diffusion models

Let us go back our discussions of the variational autoencoders from last week, see https://github.com/CompPhysics/AdvancedMachineLearning/blob/main/doc/pub/week15/ipynb/week15.ipynb. As a first attempt at understanding diffusion models, we can think of these as stacked VAEs, or better, recursive VAEs.
Let us try to see why. As an intermediate step, we consider so-called hierarchical VAEs, which can be seen as a generalization of VAEs that include multiple hierarchies of latent spaces.

# Chains of VAEs

As an intermediate step, we introduce what is often called Markovian VAE where the generative process is a Markov chain and build a hierarchy of VAEs.
Each transition down the hierarchy is Markovian, where we decode each latent set of variables $\boldsymbol{h}_t$ in terms of the previous latent variable $\boldsymbol{h}_{t-1}$. Intuitively, and visually, this can be seen as simply stacking VAEs on top of each other (see figure next slide).
One can think of such a model as a recursive VAE.

# Mathematical representation

Mathematically, we represent the joint distribution and the posterior of a Markovian VAE as

$$p(\boldsymbol{x}, \boldsymbol{h}_{1:T}) = p(\boldsymbol{h}_T) p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{h}_1) \prod_{t=2}^{T} p_{\boldsymbol{\theta}}(\boldsymbol{h}_{t-1}|\boldsymbol{h}_t)$$

$$q_{\boldsymbol{\phi}}(\boldsymbol{h}_{1:T}|\boldsymbol{x}) = q_{\boldsymbol{\phi}}(\boldsymbol{h}_1|\boldsymbol{x}) \prod_{t=2}^{T} q_{\boldsymbol{\phi}}(\boldsymbol{h}_t|\boldsymbol{h}_{t-1})$$

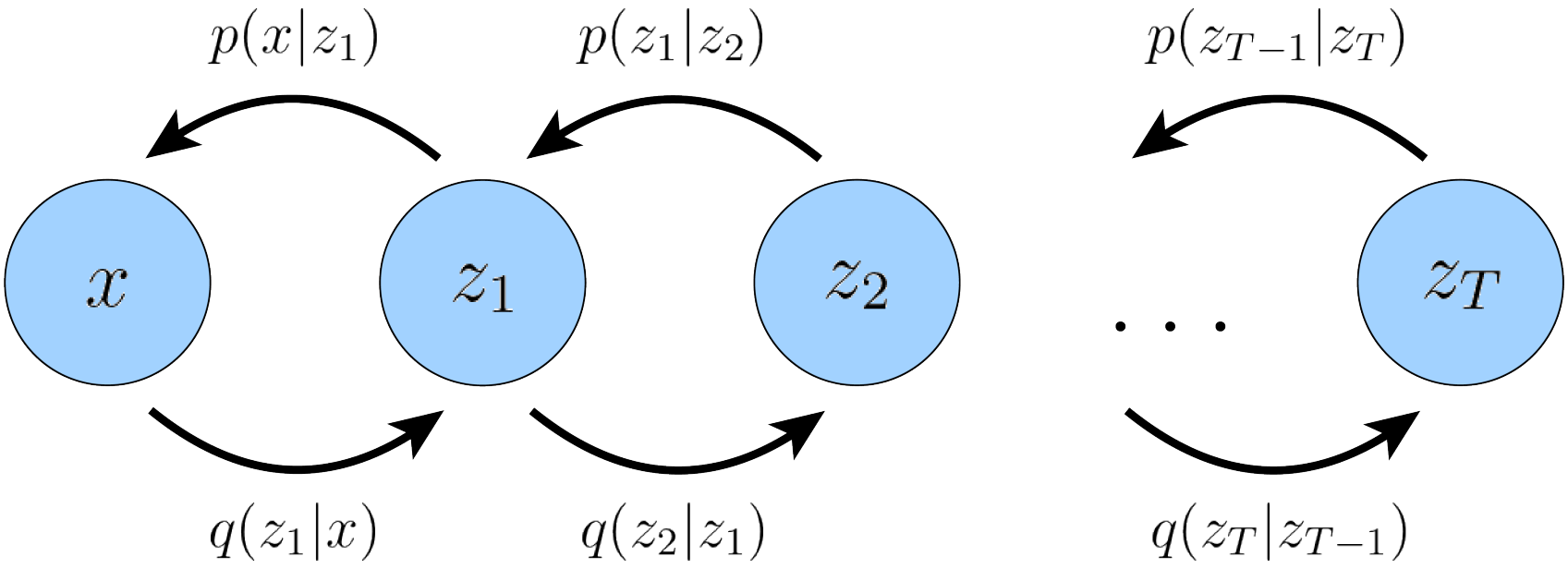
# Back to the marginal probability

We can then define the marginal probability we want to optimize as

$$\begin{aligned}
\log p(\boldsymbol{x}) &= \log \int p(\boldsymbol{x}, \boldsymbol{h}_{1:T}) d\boldsymbol{h}_{1:T} \\
&= \log \int \frac{p(\boldsymbol{x}, \boldsymbol{h}_{1:T}) q_\phi(\boldsymbol{h}_{1:T}|\boldsymbol{x})}{q_\phi(\boldsymbol{h}_{1:T}|\boldsymbol{x})} d\boldsymbol{h}_{1:T} && \text{(Multiply by 1 = } \frac{q_\phi(\boldsymbol{h}_{1:T}|}{q_\phi(\boldsymbol{h}_{1:T}|} \\
&= \log \mathbb{E}_{q_\phi(\boldsymbol{h}_{1:T}|\boldsymbol{x})} \left[ \frac{p(\boldsymbol{x}, \boldsymbol{h}_{1:T})}{q_\phi(\boldsymbol{h}_{1:T}|\boldsymbol{x})} \right] && \text{(Definition of Expectation)} \\
&\geq \mathbb{E}_{q_\phi(\boldsymbol{h}_{1:T}|\boldsymbol{x})} \left[ \log \frac{p(\boldsymbol{x}, \boldsymbol{h}_{1:T})}{q_\phi(\boldsymbol{h}_{1:T}|\boldsymbol{x})} \right] && \text{(Discussed last week)}
\end{aligned}$$

# Diffusion models for hierarchical VAE, from https://arxiv.org/abs/2208.11970

A Markovian hierarchical Variational Autoencoder with $T$ hierarchical latents. The generative process is modeled as a Markov chain, where each latent $\boldsymbol{h}_t$ is generated only from the previous latent $\boldsymbol{h}_{t+1}$



# Equation for the Markovian hierarchical VAE

We obtain then

$$\mathbb{E}_{q_\phi(\boldsymbol{h}_{1:T}|\boldsymbol{x})}\left[\log \frac{p(\boldsymbol{x},\boldsymbol{h}_{1:T})}{q_\phi(\boldsymbol{h}_{1:T}|\boldsymbol{x})}\right] = \mathbb{E}_{q_\phi(\boldsymbol{h}_{1:T}|\boldsymbol{x})}\left[\log \frac{p(\boldsymbol{h}_T)p_\theta(\boldsymbol{x}|\boldsymbol{h}_1)\prod_{t=2}^{T} p_\theta(\boldsymbol{h}_{t-1}|\boldsymbol{h}_t)}{q_\phi(\boldsymbol{h}_1|\boldsymbol{x})\prod_{t=2}^{T} q_\phi(\boldsymbol{h}_t|\boldsymbol{h}_{t-1})}\right]$$

We will modify this equation when we discuss what are normally called Variational Diffusion Models.

# Variational Diffusion Models

The easiest way to think of a Variational Diffusion Model (VDM) is as a Markovian Hierarchical Variational Autoencoder with three key restrictions:

1. The latent dimension is exactly equal to the data dimension
2. The structure of the latent encoder at each timestep is not learned; it is pre-defined as a linear Gaussian model. In other words, it is a Gaussian distribution centered around the output of the previous timestep
3. The Gaussian parameters of the latent encoders vary over time in such a way that the distribution of the latent at final timestep $T$ is a standard Gaussian

The VDM posterior is (simply rewrite)

$$q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0) = \prod_{t=1}^{T} q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$$

# Second assumption

The distribution of each latent variable in the encoder is a Gaussian centered around its previous hierarchical latent. Here then, the structure of the encoder at each timestep $t$ is not learned; it is fixed as a linear Gaussian model, where the mean and standard deviation can be set beforehand as hyperparameters, or learned as parameters.
We parameterize the Gaussian encoder with mean $\boldsymbol{\mu}_t(\boldsymbol{x}_t) = \sqrt{\alpha_t}\boldsymbol{x}_{t-1}$, and variance $\boldsymbol{\Sigma}_t(\boldsymbol{x}_t) = (1 - \alpha_t)\mathbf{I}$, where the form of the coefficients are chosen such that the variance of the latent variables stay at a similar scale; in other words, the encoding process is variance-preserving. Note that alternate Gaussian parameterizations are allowed, and lead to similar derivations. The main takeaway is that $\alpha_t$ is a (potentially learnable) coefficient that can vary with the hierarchical depth $t$, for flexibility.

# Encoder transitions

Mathematically, the encoder transitions are defined as

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) = \mathcal{N}(\boldsymbol{x}_t; \sqrt{\alpha_t}\boldsymbol{x}_{t-1}, (1-\alpha_t)\mathbf{I})$$

# Third assumption

From the third assumption, we know that $\alpha_t$ evolves over time according to a fixed or learnable schedule structured such that the distribution of the final latent $p(\boldsymbol{x}_T)$ is a standard Gaussian. We can then update the joint distribution of a Markovian VAE to write the joint distribution for a VDM as

$$p(\boldsymbol{x}_{0:T}) = p(\boldsymbol{x}_T) \prod_{t=1}^{T} p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$$

where,

$$p(\boldsymbol{x}_T) = \mathcal{N}(\boldsymbol{x}_T; 0, \mathbf{I})$$

# Noisification

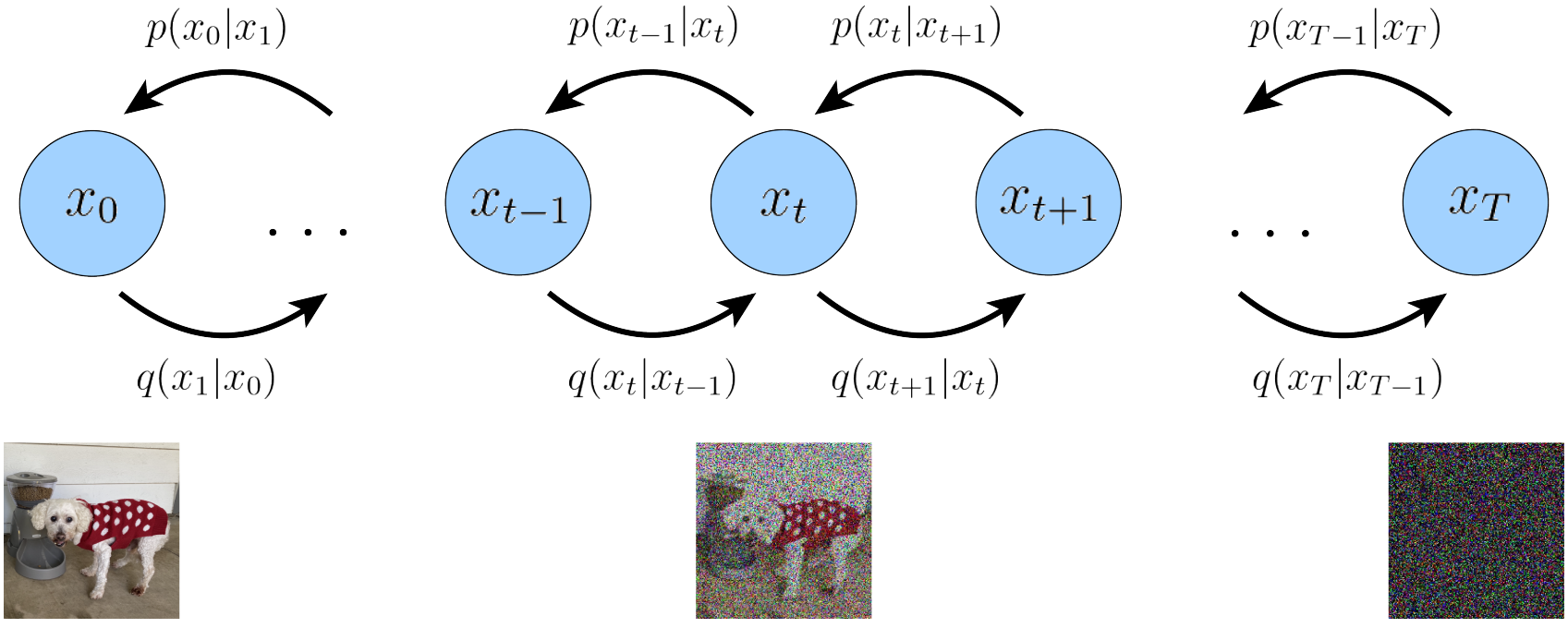
Collectively, what this set of assumptions describes is a steady noisification of an image input over time. We progressively corrupt an image by adding Gaussian noise until eventually it becomes completely identical to pure Gaussian noise. See figure on next slide.

# Gaussian modeling

Note that our encoder distributions $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$ are no longer parameterized by $\phi$, as they are completely modeled as Gaussians with defined mean and variance parameters at each timestep. Therefore, in a VDM, we are only interested in learning conditionals $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$, so that we can simulate new data. After optimizing the VDM, the sampling procedure is as simple as sampling Gaussian noise from $p(\boldsymbol{x}_T)$ and iteratively running the denoising transitions $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ for $T$ steps to generate a novel $\boldsymbol{x}_0$.

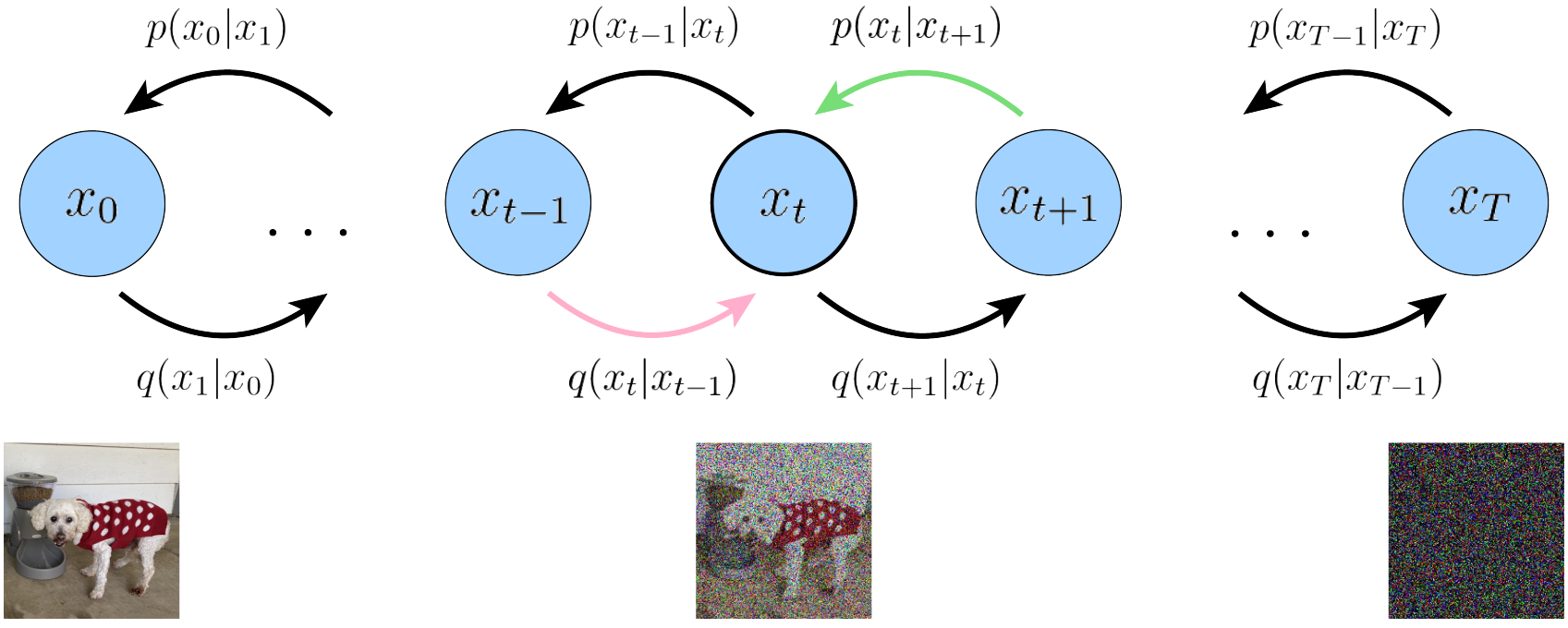
# Diffusion models, from

https://arxiv.org/abs/2208.11970



# Diffusion models, part 2, from

https://arxiv.org/abs/2208.11970



# Diffusion models, part 3, from

https://arxiv.org/abs/2208.11970