Diffusion Models

Seminar Data Mining

Grazvydas Kuras
Department of Informatics
Technische Universität München

***Abstract*— Diffusion models are a quite new concept inside the generative learning landscape [1], but most of us already got a glimpse of their capabilities with AI systems such as OpenAI's Dall-E 2, but its use case goes way beyond the creation of artsy images. Reading this, you will not only get an idea of how it works, but also take away a working example to train and generate data by yourself. This review paper aims to provide a comprehensive summary of various papers regarding diffusion models, for people who are new to this topic. Focusing on the necessary details, it will give a categorical overview and look deeper into an example. We will work on the intuition and reinforce the knowledge with an actual implementation.**

***Keywords*— Generative models; Diffusion models; Computer vision; Medical imaging; Synthetic data; Data distribution; Diffusion process; Denoising diffusion models;**

I. Introduction

Diffusion models are the new state-of-the-art family of deep generative models, breaking the long-time dominance of previous systems in image synthesis. The potential of this concept triggered a significant increase in research in the past two years, making it challenging to stay up-to-date with the recent developments [2]. It can be applied in a variety of domains, such as computer vision, medical imaging, and natural language processing, as you can see in section V.

The idea behind diffusion models is to destroy the data by adding noise, and learning the reverse process with the given destruction [3]. The resulting reverse model can be applied to random noise, creating new synthetic examples.

This review paper aims to lay out the map around diffusion models, to give an overview of the concept. Then go down a specific path, to reinforce the idea with an example, that can be applied by yourself as a final takeaway. Since this is what I wished for, when starting to learn about diffusion models, but none of the papers was providing.

So this is for everyone who wants to pick up this concept, with minor prior knowledge needed. But willing to understand the functionality of the provided equations, which may not be straightforward. Some basics of probability theory would be helpful to know, such as the normal distribution and Markov chains. As well as some Python knowledge for the implementation.

Starting with section II, we will look at what a diffusion model is and the broader concept it belongs to. Then unfold a concrete formulation of a diffusion model in section III. There, we will discuss the mathematical expressions of its components, based on the approach of a popular paper [4]. At the same time, build intuition for these expressions and learn how everything works together.

This will be reinforced in section IV, introducing an actual application available on GitHub, that can be played around with. We will reflect our knowledge on the algorithms used and discuss possible implementation choices. Lastly, we will look at interesting applications for diffusion models in section V and wrap everything up in section VI.

II. Overview Diffusion Models

Generative models

A generative model samples synthetic data (as the name suggests). The model's key defining characteristic is that the distribution sampled from, is the same as the distribution of the training data [1]. Intuitively speaking, it generates new data with the same abstract characteristics as the training data.

Definition

Diffusion models are a class of generative models, capable of sampling from complex data distributions [1]. They consist of two steps: The forward diffusion process, which adds noise to the input data progressively, until nothing is left but Gaussian noise (or any simple prior distribution) [2]. And the reverse process, which learns to recover the input data from its noised version.

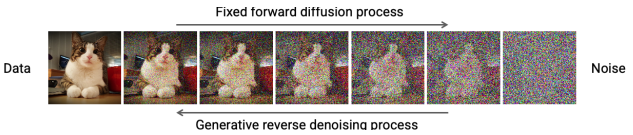

Fig. 1. Diffusion process example. Adapted from [5]

Therefore the first step will be used for training and the second for sample generation. Note that we will use images as an ongoing example, for its better illustrations, but it can be any data represented as $x \in \mathbb{R}^d$.

Types

There are three formulations commonly used in current research of diffusion models: denoising diffusion probabilistic models (DDPMs), score-based generative models (SGMs), and stochastic differential equations (Score SDEs) [2]. Out of those, we will continue with DDPMs for their simpler, yet effective implementation in the next section.

III. Denoising Diffusion Probabilistic Models (DDPMs)

A *denoising diffusion probabilistic model* (DDPM) specifies the forward and reverse process as two Markov chains, as you can see in Figure 3. Out of those, the forward process can be modeled accurately. This means the conditional probability distribution $q(x_t|x_{t-1})$, also known as transition kernel or Markov kernel, will correspond to the diffusion outcomes. Whereas the reverse process, also known as denoising [3], is parameterized by deep neural networks to approximate the forward process transitions, but in reverse [2]. To formulate our DDPM, we will specify both directions in detail.

Multivariate normal distribution

Both transition kernels, forward (4) and reverse (8), will be modeled using the multivariate normal distribution $\mathcal{N}(x; \mu, \Sigma)$. So this section will show the underlying formula and provide a useful intuition, to make you familiar with it.

Let's look at the Gauss function for a given μ' and σ:

$$f(x') = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x'-\mu'}{\sigma}\right)^2}. \tag{1}$$

The value of this function depends on the distance from x' to μ'. Thus the smaller $(x' - \mu')^2$, the higher the resulting probability. We can apply the same intuition to the multivariate case, as seen in this bivariate example with $\mu = (-1.5, 0)$.

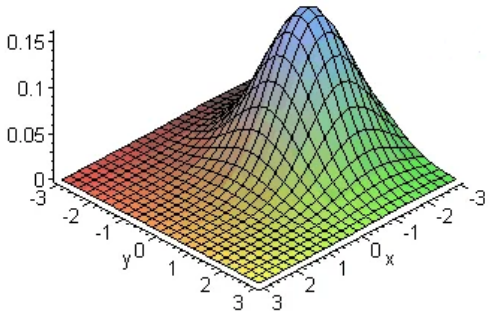

Fig. 2. Visualisation of a bivariate normal distribution for $\mu = (0, -1, 5)$. Adapted from https://www.statisticshowto.com/bivariate-normal-distribution/

We denote the probability density at a point $x \in \mathbb{R}^d$, with $\mu \in \mathbb{R}^d$ and the identity matrix $I \in \mathbb{R}^{d\times d}$, as

$$\mathcal{N}(x; \mu, \sigma I) := \frac{1}{\sigma(2\pi)^{1/d}} \exp\left(-\frac{(x-\mu)^T(x-\mu)}{2\sigma^2}\right). \tag{2}$$

Notice the new distance calculation $(x - \mu)^T(x - \mu)$ and the similarity to the classic Gauss function. Note that this expression (2) is simplified to the isotropic case, since our covariance matrix Σ is $\sigma^2 I$.

Forwards

In the forward process, we are adding independent standard Gaussians $\epsilon_t \sim \mathcal{N}(\vec{0}, I)$ to an image $x_0 \in \mathbb{R}^d$. The ratio between noise and previous data, is determined by a so-called variance schedule $\beta_1, \ldots, \beta_T \in [0, 1)$. It is typically hand-designed [2] [3], but can also be learned [4]. The resulting sequence $x_1, x_2, \ldots, x_T$ goes according to the rule

$$x_t = \sqrt{1-\beta_t} x_{t-1} + \sqrt{\beta_t}\epsilon_t. \tag{3}$$

Given the noise distribution $\mathcal{N}(\vec{0}, I)$ and variance schedule, we can specify the transition kernel of this corruption process (3) as

$$q(x_t|x_{t-1}) := \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I). \tag{4}$$

The mean is the previous point x_{t-1}, but nudged towards the standard normal distribution ($\mu = \vec{0}$), informally speaking. The joint distribution conditioned on x_0 is

$$q(x_{1:T}|x_0) = q(x_1, \ldots, x_T|x_0) := \prod_{t=1}^{T} q(x_t|x_{t-1}). \tag{5}$$

It is important to understand, that the joint distribution's (5) value is the highest (on average) for sequences, which result from our corruption process (3). That's why, we will try to match this value with our reverse formulation in *Backwards*, as a trajectory for training the neural network, since it should approximate the same sequences, but in reverse.

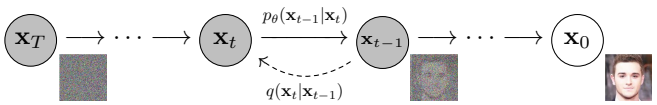

Fig. 3. Visualisation of Markov chain for forward and reverse process. Adapted from [4]

A property of this forward process is that we can sample x_t at an arbitrary timestep [4]. Using the notation $\alpha_t := 1 - \beta_t$ and $\bar{\alpha}_t := \prod_{s=1}^{t} \alpha_s$ we get

$$x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, \tag{6}$$

where ϵ is a single standard Gaussian [2] [3]. This will be helpful in further formulations. We can also write

$$q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)I). \tag{7}$$

This would be used in some later proofs, which are not covered in this review paper. But it's still helpful for the understanding of the forward transition kernel.

Backwards

The reverse process is also modeled using the multivariate normal distribution. By parameterizing the mean and variance, we can specify the reverse transition kernel as

$$p_\theta(x_{t-1}|x_t) := \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \Sigma_\theta(x_t, t)). \tag{8}$$

In simple words, this approximates the probability distribution towards the real data, undoing the corresponding corruption step. The initial input x_T is a standard Gaussian $p(x_T) = \mathcal{N}(x_T; \vec{0}, I)$. This is because the denoising should reverse the forward process, which is constructed such that $q(x_T) \approx$

$\mathcal{N}(x_T; \vec{0}, I)$ [2]. So we can model the joint distribution with our reverse transition kernel as

$$p_\theta(x_{0:T}) := p(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t). \tag{9}$$

This allows us a direct comparison with the forward process (5) for any data input x_0 and its corresponding noising sequence $x_1, x_2, \ldots, x_T$. With that in hand, we can model a loss function for our goal of reversing the forward process; and improving our parameterization.

Training objective

For the scope of this paper, you don't have to fully understand the equations given in this section. But it's important to recognize the key ideas discussed, that will lead to our training objective.

As already mentioned in *Forwards*, we want to achieve the same value with our reverse joint distribution (9), as the forward joint distribution (5) [2], given any noising sequence using (3). As a result, we will obtain our desired reverse model. Because matching the reverse joint distribution (9) means evaluating transitions towards the real data as most likely, starting from pure noise x_T. Since this is what the forward joint distribution (5) does, but for the other direction, starting from x_0.

To achieve that, we optimise the variational lower bound,

$$\begin{aligned} &\mathbb{E}[-\log p_\theta(x_0)] \leq \mathbb{E}_q \left[-\log \frac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)} \right] = \\ &\mathbb{E}_q \left[-\log p(x_T) - \sum_{t \geq 1} \log \frac{p_\theta(x_{t-1}|x_t)}{q(x_t|x_{t-1})} \right] =: L. \end{aligned} \tag{10}$$

Notice how matching the joint distributions results in the minimization of the negative log-likelihood. It turns out that this match is hard to achieve, as it is still a topic of research. So we will look at the currently most popular approach, proposed by Ho et al. [4], which is justified by its simplicity and empirical results of quality samples. First, we can rebuild our loss term (10) to

$$\begin{aligned} &\mathbb{E}_q \Bigg[\underbrace{D_{KL}(q(x_T|x_0) \parallel p(x_T))}_{L_T} + \\ &\sum_{t>1} \underbrace{q(x_{t-1}|x_t, x_0) p_\theta(x_{t-1}|x_t)}_{L_{t-1}} \underbrace{- \log p_\theta(x_0|x_1)}_{L_0} \Bigg]. \end{aligned} \tag{11}$$

When using a fixed variance schedule, L_T has no learnable parameters and can be ignored, since it becomes a constant during training. Using the so-called posterior term

$$\begin{aligned} &q(x_{t-1}|x_t, x_0) = \mathcal{N}(x_{t-1}; \tilde{\mu}_t(x_t, x_0), \tilde{\beta}_t I), \\ &\text{where} \quad \tilde{\mu}_t(x_t, x_0) = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}} \epsilon \right). \end{aligned} \tag{12}$$

Together with a simplified backward transition kernel (8), after setting the variance $\Sigma_\theta(x_t, t) = \sigma_t^2 I$ as fixed constants,

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \sigma_t^2 I), \tag{13}$$

we can rewrite L_{t-1} as

$$L_{t-1} = \mathbb{E}_q \left[\frac{1}{2\sigma_t^2} \|\tilde{\mu}_t(x_t, x_0) - \mu_\theta(x_t, t)\|^2 \right] + C. \tag{14}$$

Now a mere comparison between the means is achieved. So we see that our loss can be improved by parameterizing μ_θ to predict $\tilde{\mu}_t$. We don't even have to consider the whole corruption sequence, because the means only correspond to a single step t. Therefore, t will be chosen uniformly between 1 and T while training.

A crucial observation is that we can approximate $\tilde{\mu}_t(x_t, x_0)$ with a parameterization of $\epsilon_\theta(x_t, t)$. Since the formulation in (12) doesn't rely on x_0, which we don't have in the reverse process, we can specify

$$\begin{aligned} \mu_\theta(x_t, t) &= \tilde{\mu}_t \left(x_t, \frac{1}{\sqrt{\bar{\alpha}_t}} (x_t - \sqrt{1-\bar{\alpha}_t} \epsilon_\theta(x_t)) \right) \\ &= \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}} \epsilon_\theta(x_t, t) \right). \end{aligned} \tag{15}$$

So with this background, we can rebuild our loss (14) using an approximation of $\epsilon_\theta(x_t, t)$ and (7) into

$$\mathbb{E}_{x_0, \epsilon} \left[\frac{\beta_t^2}{2\sigma_t^2 \alpha_t (1-\bar{\alpha}_t)} \parallel \epsilon - \epsilon_\theta(\underbrace{\sqrt{\bar{\alpha}_t} x_0 + \sqrt{1-\bar{\alpha}_t} \epsilon}_{x_t}, t) \parallel^2 \right], \tag{16}$$

which the authors simplified even further into

$$L_{\text{simple}}(\theta) := \mathbb{E}_{t, x_0, \epsilon} \left[\left\| \epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t} x_0 + \sqrt{1-\bar{\alpha}_t} \epsilon, t) \right\|^2 \right], \tag{17}$$

leaving us with a mere approximation of ϵ. Sampling each reverse step can then be performed using (15) and some white noise, resembling Langevin dynamics. All the details leading to these results can be found in [4]. Now that we discussed all the components, we can look into the summary in section VI, to get an overview, before we continue with the implementation.

IV. Implementation

Algorithms

Looking at the results from section III, we can use (17) in our training algorithm, since we know that its minimization will improve our reverse transition model. Algorithm 1 takes some data x_0, a random but uniformly distributed timestep $t \in 1, ..., T$ and Gaussian noise e. Then gets the corresponding x_t by adding the noise using (7) and takes a gradient descent step, based on the prediction of the added noise.

Algorithm 1 Training

1: **repeat**
2: $x_0 \sim q(x_0)$
3: $t \sim \text{Uniform}(\{1, \ldots, T\})$
4: $\epsilon \sim \mathcal{N}(0, I)$
5: Take gradient descent step on
$\nabla_\theta \|\epsilon - \epsilon_\theta(\sqrt{\bar{a}_t}\, x_0 + \sqrt{1-\bar{a}_t}\, \epsilon, t)\|^2$
6: **until** converged

Sampling will be an iterative process, starting with Gaussian noise $x_T \sim \mathcal{N}(0, I)$. Each next step x_{t-1} is the approximated mean (15), in addition to white noise $\sigma_t z$, inspired by Langevin dynamics [6]. No white noise is added in the last step, so we display our result $\mu_\theta(x_1, 1)$ [4].

Algorithm 2 Sampling

1: $x_T \sim \mathcal{N}(0, I))$
2: **for** $t = T, \dots, 1$ **do**
3: $\quad z \sim \mathcal{N}(0, I)$ if $t > 1$, else $z = 0$
4: $\quad x_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}} \epsilon_\theta(x_t, t) \right) + \sigma_t z$
5: **end for**
6: **return** x_0

Implementation choices

You have plenty of options to play around with, when building a DDPM for a specific distribution, as long as the dimensions of input and output remain the same [7]. We will look into some common approaches for implementation.

Starting with the total number of steps T, you generally need more steps for more complex distributions. For example $T = 40$ was enough to learn a 2D-Swiss-roll distribution [8], whereas $T = 500$ was used to learn 32x32 black-and-white images from the MNIST dataset [3] and Ho et al. [4] achieved novel samples for 256x256 colored images, using $T = 1000$. Many small steps result in coarser and more varied interpolations [4], avoiding overfitting.

As already mentioned, our variance schedule $\beta_1, \dots, \beta_T$ is learnable by reparametrization [4], but we usually keep it fixed. Ho et al. [4] uses a linear schedule from $\beta_1 = 10^-4$ to $\beta_T = 0.02$, since we can ignore L_T this way. Then there is the question of what to approximate for our reverse process. Based on the loss functions in *Training objective*, we see that a model could approximate $\tilde{\mu}_t(x_t, x_0)$ in (14) and ϵ in (17). A direct approximation of x_0 would be also possible, but yields worse results [4]. We need to specify a neural model for the approximation. It could be any, with the same output dimensions as the input. But most commonly, a U-Net-shaped convolutional neural network is used.

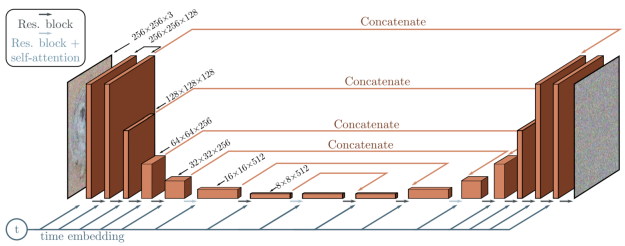

Fig. 4. Example structure of a U-Net. Adapted from [6]

Explaining U-Nets is out of scope for this review paper, therefore we will rely on sufficient sample implementations. Lastly, we make sure that the data is scaled linearly to $[-1, 1]$, for consistency within our neural network [4].

We will cover two implementations with code examples in the next subsection *Code*. Both will use a linear variance schedule and approximate the added noise, applying the proposed algorithms in *Algorithms*. The first one will be trained to sample from the 2D-Swiss-roll distribution. For that we will use $T = 40$, taking an example from [8] and because of the simpler data, we can use a more lightweight neural network. After some experimenting, a feed forward neural network with time embedding did the job.

The second one is for training on images. Due to the increased data size and a heavier neural model, access to a GPU is recommended. Training over long and intense periods on a CPU can increase the heat significantly, reaching levels that shorten the lifespan or cause damage in the worst case. We will use a U-Net and $T = 1000$, inspired by Ho et al. [4]. Since images, such as 32x32 images of flowers, have a more complex underlying distribution.

Code

Now let's apply our knowledge. I provided a GitHub repository at `https://github.com/Grazvy/Diffusion_Models_demonstration.git`. Most of the infrastructure, which is not relevant for diffusion models specifically, was hidden in the *utils* folder. You can find a Jupyter Notebook in the root directory, which walks through all the components leading to a functioning DDPM. This subsection will guide you through those steps and how to use the provided functionality. You will see how to use an already trained model, which is provided, and how to train one by yourself. You can use predefined, but also custom datasets with some adjustments in the dataloader. But keep in mind that a GPU is recommended for images. The environment needs to be set up, before you can run the code. All the details are provided in the README.md file. If you don't have time for this, you can also look at the provided cell outputs. Note that

The following demonstration will be for the 2D-Swiss-roll distribution, but the process looks the same using other datasets; since all of them can be described as a vector. You should run the corresponding code cells in sequence, when their functionality and definitions are being discussed.

The applied dataset is specified in *BaseConfig* and samples of it can be visualized using *visualize_data()*. You can look at the currently implemented variance schedule using the according method. This will be applied in the *forward_diffusion()*, an implementation of our forward diffusion (6), where the variance schedule is provided via *ConstantsManager*. As you can see, this can also be visualized for a given dataset and timesteps.

The applied dataset is specified in *BaseConfig*. There are provided methods for the visualization of the datasets, variance schedule, and the resulting forward diffusion, which applies both. You can look into the corresponding definitions, to see possible visualization adjustments. The *forward_diffusion()* implements the noising process (6), which will be used during training, as specified in Algorithm 1. This is implemented

in *train_one_epoch()*, where the data loader provides x_0, *forward_diffusion()* turns it into x_t and the model gets optimised to predict the induced noise e. Algorithm 2 is implemented in *generate_samples()*. The results are displayed using *InferenceLogger*, which manages them according to *InferType*. Before you can run the training, you need to specify the number of diffusion steps T and the sample generation during training. This is done in *TrainingConfig*, besides some basic options such as batch size and the amount of epochs. And of course, you need to specify the model being used. Two architectures are provided in the folder *neural_networks*, but you can add your own as well. The pre-trained model is a feed forward neural network. You can see the corresponding configuration *FNNConfig* and if you want to work with image data, *UnetConfig* provides an example configuration for a U-Net. Now with all the configurations specified, we can load our objects and start training or skip to sample generation.

We load and save our neural network using *ModelManager*. Since the *SwissRoll* folder is already defined in *models*, including the trained model, we would load this model when creating a *ModelManager* object. If you want to create a new one from scratch, just change *model_name* to a new filename, *ModelManager* will create a new folder and load it automatically with the next instance. The corresponding neural network will be created using *get_model()*.

After creating the *dataloader* and *constants* objects, which hold our dataset and variance schedule respectively, we can start the training loop. This will call *train_one_epoch()*, store the progress as a checkpoint, and create a test sample, as specified in *TrainingConfig*. When you're happy with the results, you can use the model to generate samples in the *Inference* section. Again, by specifying the configurations accordingly and running *generate_samples()*.

According to the dataset, you can use different ways to display your results, which will be saved. For images, it can be either the resulting image using *InferType.IMAGE* or the whole transition process, starting with noise, as a gif using (InferType.VIDEO). The equivalent for points would be *InferType.PLOT* and *InferType.PLOT_PROCESS* respectively.

If you want to add custom data for training, you can adjust *dataLoader.py* accordingly. The simplest way to add images would be to copy the approach of the *Flowers* dataset or *swissRollLoader.py*. But as well as for the neural network, you have to make sure that the tensor dimensions are compatible with components such as the variance schedule. Speaking of neural networks, you can add your custom architectures in the respective folder and make sure to adjust *get_model()* in *ModelManager*. Also helpful to know is that you can create and load custom checkpoints using *create_checkpoint()* and *load_checkpoint()* in *ModelManager*. As well as save your progress using *update_checkpoint()*.

V. Applications

Computer Vision

The capabilities of diffusion models to generate unique high-quality images were hard to miss in the last years, through sophisticated tools such as Dall-E or Midjourney. Dall-E 2 for example, proposes a two-stage approach for Text-to-Image generation: A prior model that generates a so-called CLIP-based image embedding for a text input. And a diffusion-based decoder that generates an image conditioned on the provided image embedding [2].

But there are more applications related to computer vision, such as super resolution, a process that increases the size of an image without compromising on its quality. One approach is to connect multiple diffusion models in sequence, increasing the resolution with each step [2].

Fig. 5. Example of super resolution. Adapted from [9]

Image Inpainting, also known as Image Completion [10] is another interesting use case, which fills out missing regions; while maintaining harmony with the surroundings.

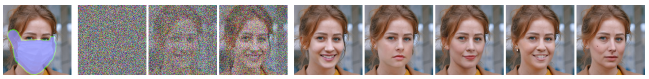

Fig. 6. Inpainting example, where a mask gets replaced with fitting illustrations of a mouth. Adapted from [10]

Making it a powerful tool for image manipulation.

Medical Imaging

A survey in the medical image analysis journal [1] describes the clinical importance of generative models, specifically diffusion models. For which, image generation is the primary objective.

The synthetic images help with a severe class imbalance in databases, which may occur due to the rare nature of some pathologies. In education, such images can be used for teaching and practice. A study was even conducted by Moghadam et al. [11], in which two pathologists with different levels of expertise couldn't distinguish real from synthetic data (Figure 7), besides some with lower confidence levels. Because of their quality, the images can be used as annotated data. This is very useful, since the annotation of data is a lengthy and expensive process with the assistance of an expert. It was also shown by Chen et al. [12] and Akrout et al. [13], that including synthetic data in the training of classifiers improves the performance, in contrast to using real data only.

Some other use cases mentioned in the survey include segmentation, which is concerned with decomposing an image into multiple meaningful image segments. Denoising, which handles noise in measurements. And anomaly detection, which

translates the input image into a healthy one, detecting anomalies through the difference between input and output.

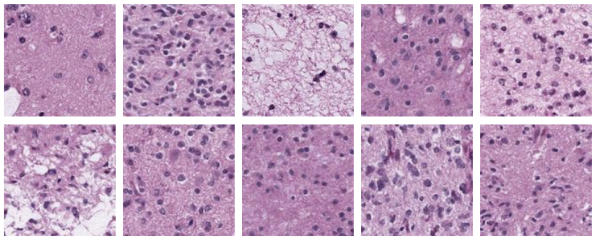

Fig. 7. Ten synthetic histopathology images generated by MFDPM. Adapted from [11]

Other

Also worth mentioning is segmentation, which labels each pixel to an object category. And anomaly detection, which was mentioned in medical imaging, but can be applied in any context. Furthermore, diffusion models are used in classification, material design, and natural language generation. The details and more examples are available in [2].

VI. Summary

This review paper presented how image synthesis is performed, using the promising new concept of diffusion models. In summary: To sample from the distribution of the training data, diffusion models aim to reverse the forward diffusion process. To achieve this, a DDPM specifies both directions as two Markov chains. We look at the approach of Ho et al. [4], to train the corresponding reverse model, because of the promising results shown in their popular paper. It proposes approximating the mean (14) or noise (17) for each step.

Using all this knowledge, we formalized the algorithms for training and sampling, considered different implementation choices, and finally looked into an actual code example, which turns the theory into practice.

As a last takeaway, we saw examples of diffusion models being applied in various real applications. And the utility that synthetic data can have.

References

[1] M. H. R. A. M. F. I. H. D. M. Amirhossein Kazerouni, Ehsan Khodapanah Aghdam, "Diffusion models in medical imaging: A comprehensive survey," 2023. [Online]. Available: https://doi.org/10.1016/j.media.2023.102846

[2] Y. S. S. H. R. X. Y. Z. W. Z. B. C. Ling Yang, Zhilong Zhang and M.-H. Yang, "Diffusion models: A comprehensive survey of methods and applications," 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3626235

[3] P. G. C. Catherine F. Higham, Desmond J. Higham, "Diffusion models for generative artificial intelligence: An introduction for applied mathematicians," 2023. [Online]. Available: https://arxiv.org/pdf/2312.14977

[4] P. A. Jonathan Ho, Ajay Jain, "Denoising diffusion probabilistic models," 2020. [Online]. Available: https://arxiv.org/abs/2006.11239

[5] A. V. Karsten Kreis, Ruiqi Gao, "Denoising diffusion-based generative modeling: Foundations and applications," 2022. [Online]. Available: https://cvpr2022-tutorial-diffusion-models.github.io/

[6] V. Singh, "An in-depth guide to denoising diffusion probabilistic models ddpm – theory to implementation," 2023. [Online]. Available: https://learnopencv.com/denoising-diffusion-probabilistic-models/

[7] R. O'Connor, "Introduction to diffusion models for machine learning," 2022. [Online]. Available: https://www.assemblyai.com/blog/diffusion-models-for-machine-learning-introduction/

[8] N. M. S. G. Jascha Sohl-Dickstein, Eric A. Weiss, "Deep unsupervised learning using nonequilibrium thermodynamics," 2015. [Online]. Available: https://arxiv.org/abs/1503.03585

[9] W. C. T. S. D. J. F. M. N. Chitwan Saharia, Jonathan Ho, "Image super-resolution via iterative refinement," 2021. [Online]. Available: https://arxiv.org/pdf/2104.07636

[10] A. R. F. Y. R. T. L. V. G. Andreas Lugmayr, Martin Danelljan, "Repaint: Inpainting using denoising diffusion probabilistic models," 2022. [Online]. Available: https://arxiv.org/abs/2201.09865

[11] K. C. M. J. L. S. Y. H. F. A. B. Puria Azadi Moghadam, Sanne Van Dalen, "A morphology focused diffusion probabilistic model for synthesis of histopathology images," 2022. [Online]. Available: https://arxiv.org/abs/2209.13167

[12] T. Y. C. D. F. K. W. F. M. Richard J. Chen, Ming Y. Lu, "Synthetic data in machine learning for medicine and healthcare," 2021. [Online]. Available: https://doi.org/10.1038/s41551-021-00751-8

[13] A. I. A.-R. C.-B. S. L. Z. Yijun Yang, Huazhu Fu, "Diffmic: Dual-guidance diffusion network for medical image classification," 2023. [Online]. Available: https://arxiv.org/abs/2303.10610