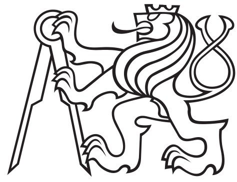
Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Diffusion models for path planning

Petr Zahradník

Supervisor: Ing. Vojtěch Vonásek, Ph.D.
Field of study: Open informatics
May 2024

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date 24. May 2024

Signature

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 24. May 2024

podpis autora práce

Abstract

The Denoising Diffusion Probabilistic Model (DDPM) is a recent generative neural network framework. We describe its application to feasible path planning. We analyze the current open-source implementations and their shortcomings blocking wider adoptation. We propose significant improvements to one of them and build a neural network architecture using our implementation. We evaluate the performance of the model on a variety of tasks and compare it to other state-of-the-art methods achieving success even in previously unseen environments. Furthermore, we propose methods to combine DDPMs with other path-planning algorithms to improve their performance.

Keywords: generative AI, diffusion, neural networks, path planning

Supervisor: Ing. Vojtěch Vonásek, Ph.D.

Abstrakt

Odšumovací difúzní probabilistický model (DDPM) je nedávno objevený způsob učení generativních neuronových sítí. V této práci popisujeme jeho aplikaci pro hledání průchozích cest pro roboty. Analyzujeme existující, veřejně dostupné implementace a jejich nedostatky znemožňující širší použití. Na základě námi navržených změn stavíme vylepšenou architekturu neuronové sítě. Dále testujeme model na různých úlohách a porovnáváme jej s jinými známými metodami. Náš přístup dosahuje úspěšných výsledků i v prostředích, která nebyla součástí trénovací množiny. V neposlední řadě navrhujeme, jak zkombinovat DDPM s jinými algoritmy pro plánování cest a dosáhnout ještě lepších výsledků.

Klíčová slova: generativní UI, difúze, neuronové sítě, plánování cesty

Překlad názvu: Difúzní modely pro robotické plánování

Contents

Thesis Assignment **1**

List of Notations **3**

List of Abbreviations **4**

1 Introduction **5**

2 Related work **7**

2.1 General planning 7

2.2 Sampling-based planning 7

2.3 Planning for robots 9

2.4 Transformer models 9

2.5 Denoising Diffusion Probabilistic Models 10

2.6 Path planning using neural networks 10

3 Diffusion models **13**

3.1 Forward diffusion 13

3.2 Reverse diffusion 14

3.3 Training 14

3.4 Inference 17

3.4.1 Inpainting 19

3.4.2 Steering 19

4 Analysis of existing implementations **21**

4.1 The Diffuser architecture 21

4.2 State of the implementation 22

4.3 Our contribution to the Diffusers 24

5 Diffuser Architecture **29**

5.1 Neural network architecture 29

5.2 Diffuser as denoiser 31

6 Results **35**

6.1 Simple shapes 35

6.1.1 Circles dataset 35

6.1.2 3D Spirals dataset 37

6.2 Navigation problems 38

6.2.1 Easy Maze dataset 41

6.2.2 Medium Maze dataset 43

6.2.3 Hard Maze dataset 43

6.3 Noise schedule 46

6.4 Comparison with other planners 47

6.5 Summary 47

7 Conclusion **51**

7.1 Conclusion 51

7.2 Future work 52

A List of attachements **53**

B Bibliography **55**

Figures

2.1 Comaprison of discrete and continuous planning 8
2.2 The unCLIP inference diagram . 10
2.3 The VQ-MPT diagram 11

3.1 Forward and reverse diffusion . . 15
3.2 DDPM training diagram 17
3.3 DDPM inference diagram 18
3.4 Inpainting example 19

4.1 Architecture diagram of the original U-Net 22
4.2 Diagram of the class composition in the Diffusers library 25
4.3 Diagram of the class composition after rewrite 26

5.1 Architecture diagram of the diffuser neural network 30
5.2 Diagrams of the U-Net blocks . . 31
5.3 Diagram of ResnetBlock1D 32
5.4 Diagram of Positional Embedding 33

6.1 Circles dataset 36
6.2 Training on Circles dataset 37
6.3 Generated Circles 37
6.4 Generated 3D Spirals 38
6.5 3D Spirals reverse diffusion 39
6.6 Easy Maze dataset 40
6.7 DDPM batch size scaling 41
6.8 Model in Hard Maze environment 42
6.9 Model in Medium Maze environment 44
6.10 Model in Hard Maze environment 45
6.11 Collision rate for various inference schedules 46
6.12 Path length rate for different inference schedules 46
6.13 Feasible path time comparison with OMPL planners 48
6.14 Collision rate comparison with OMPL planners 49

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Zahradník Petr**	Personal ID number:	**492286**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence**		

II. Master’s thesis details

Master’s thesis title in English:

Diffusion models for path planning

Master’s thesis title in Czech:

Difúzní modely pro robotické plánování

Guidelines:

1. Get familiar with denoising diffusion models [1,2]. Get familiar with path planning [5], and latent path planning.
2. Implement a path denoising diffusion model following [3,4]. Consider at least 3D configuration space. Design suitable environments and robots.
3. Investigate the ability to steer path according to a given value function [3]. The goal is generate feasible (collision-free) path between two configurations.
4. Train the diffusion model on a selected training dataset and experimentally verify its abilities on a set of simple problems, e.g. 2D path in maze, robotic manipulator.
5. Compare the diffusion model with other path planning algorithms e.g. RRT, RRT*, grid-based Q-learning. Use OMPL framework [6] for the comparison.

Bibliography / sources:

1. J. Ho, A. Jain, and P. Abbeel. Denoising Diffusion Probabilistic Models. 2020-12- doi: 10.48550/arXiv.2006.11239
2. A. Nichol and P. Dhariwal. Improved Denoising Diffusion Probabilistic Models. 2021-02-18. doi: 10.48550/arXiv.2102.09672
3. Jänner, M., Du, Y., Tenenbaum, J. B., & Levine, S. (2022). Planning with Diffusion for Flexible Bevavior Synthesis. arXiv (Cornell University). doi: 10.48550/arxiv.2205.09991
4. Carvalho, J. F., Le, A. T., Baierl, M., Koert, D., & Peters, J. (2023). Motion Planning Diffusion: Learning and Planning of Robot Motions with Diffusion Models. arXiv (Cornell University). doi: 10.48550/arxiv.2308.01557
5. S. M. LaValle, Planning algorithms, 2006, Cambridge press
6. Mark Moll, Ioan A. ucan, Lydia E. Kavraki, Benchmarking Motion Planning Algorithms: An Extensible Infrastructure for Analysis and Visualization, IEEE Robotics & Automation Magazine, 22(3):96–102, September 2015.

 © VUT v Praze, Design: VUT v Praze, VIC

Name and workplace of master’s thesis supervisor:

Ing. Vojt ch Vonásek, Ph.D. Multi-robot Systems FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **14.02.2024** Deadline for master's thesis submission: __________

Assignment valid until: **21.09.2025**

Ing. Vojt ch Vonásek, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

. Date of assignment receipt	Student’s signature

© VUT v Praze, Design: VUT v Praze, VIC

List of Notations

Symbol	Meaning
$\|v\|$	L^2-Norm of vector v
$[a;b)$	Interval $\{x \in \mathbb{R} \mid a \leq x < b\}$
$\mathbf{0}$	Zero vector
$\boldsymbol{I}$	Identity matrix
v_i	i-th element of v
t	Diffusion step, also known as diffusion time
x^t	Diffusion state after t diffusion steps
$\mathcal{S}$	State space
$\mathcal{S}_{\text{obstacle}}$	Obstacle state space
$\mathcal{S}_{\text{free}}$	Free state space
$p(x)$	Probability density function of a random variable x
$p(x \mid y)$	Conditional probability density function of random variable x given y
μ_θ	Mean of a normal distribution parameterized by θ
Σ_θ	Covariance matrix of a normal distribution parameterized by θ
$x \sim \mathcal{N}(\mu, \Sigma)$	Sample x is drawn from a normal distribution with mean μ and covariance Σ
$p(x) = \mathcal{N}(x; \mu, \Sigma)$	Probability density function of a normal distribution with mean μ and covariance Σ evaluated at x
$D_{KL}(p \parallel q)$	Kullback-Leibler divergence between probability distributions p and q
$\mathbb{E}_{x \sim p}[f(x)]$	Expectation of function f with respect to x drawn from distribution p

List of Abbreviations

Abbreviation	Meaning
A*	A-star, search algorithm
Adam	Adaptive Moment Estimation, optimization algorithm
DDPM	Denoising Diffusion Probabilistic Model, neural network training framework
ELBO	Evidence Lower Bound, also known as Variational Lower Bound, VAE loss
GAN	Generative adversarial network, neural network training framework
GELU	Gaussian Error Linear Unit, activation function
GPT	Generative Pre-trained Transformer, neural network architecture
GPU	Graphics Processing Unit, hardware accelerator for parallel numeric computation
GroupNorm	Group Normalization, normalization technique
LBKPIECE1	Lazy Bi-directional KPIECE with one level of discretization, planning algorithm
LoRA	Low-Rank Adaptation, approximation method
OMPL	Open Motion Planning Library, motion planning library
PRM	Probabilistic Roadmap, planning algorithm
ReLU	Rectified Linear Unit, activation function
ResNet	Residual Network, neural network architecture
RRT	Rapidly-exploring Random Tree, planning algorithm
VAE	Variational Auto-Encoder, neural network architecture
VQ-MPT	Vector Quantized-Motion Planning Transformer, neural network training framework

Chapter 1

Introduction

In this work, we will be solving the path planning problem using Denoising Diffusion Probabilistic Model (DDPM). Path planning is essential for mobile robots [1], robotic manipulators [2], navigation of drone swarms [3] and many more. Planning a path for a robot around obstacles (also known as the *generalized mover's problem*) is known to be PSPACE-hard [4].

Many algorithms can solve different instances of the path planning problem, such as Rapidly-exploring Random Trees (RRT) [5]. Still, the problem is challenging in general, and many instances are hard or unsolvable in practice. For example, the kinematics of the robot put constraints on the turning radius of the path, or the path is required to pass through a narrow passage, whose location is not known.

Path planning usually searches for a feasible, short or smooth path, or a collection of diverse paths. In this work, we will be showing a very general way of path planning, which can be used for all these purposes at the same time. Our method of choice is the Denoising Diffusion Probabilistic Model (DDPM) [6], which was only recently introduced to robotics.

We build upon the few works that have already used DDPM for path planning and bring a concise overview of the implementation of such a model.

In Chapter 2 we give a brief overview of methods and algorithms related to solving the path planning problem and mention several key works that influenced the development of DDPMs.

Chapter 3 contains a detailed explanation of the DDPM framework, its components, and the training process. To our knowledge, this is the first time the DDPM framework has been explained in such detail in one place. During the writing of this work, we wished for a similar resource, but we could not find one. The implementation details should be helpful for anyone who wants to understand DDPMs and use them in their work.

The core of this work is in Chapter 4 where we discuss the state of the open

source DDPM implementation and explain the troubles we encountered trying to use them. We present significant changes we made to one of the implementations fixing many errors and adding essential features needed for path planning. We give the improved implementation to the community in the hope that it will be useful for others.

In Chapter 5 we give a detailed overview of the neural network architecture we used for the DDPM. We explain in detail the building blocks and the choices we made during the design process.

Our last contribution is a series of experiments in Chapter 6 where we show the performance of the DDPM in many different task settings. We designed a series of experiments to show the ability to transfer learned knowledge to previously unseen environments. We compare our DDPM implementation to several baseline algorithms and discuss the strengths and weaknesses of the DDPM. We also suggest, how the DDPM could be used together with other algorithms to improve its performance.

Chapter 2

Related work

2.1 General planning

Planning is the problem of finding a sequence of steps required to get from a given start state to a given goal state. States are elements from a configuration space $\mathcal{S}$, a set of all configurations or states of the system. For example, we might want to find the movements to solve a Rubik's cube, escape a maze or guide a robotic hand to lift a cup of coffee. Historically, planning has been studied extensively, and many different algorithms exist for many problem settings.

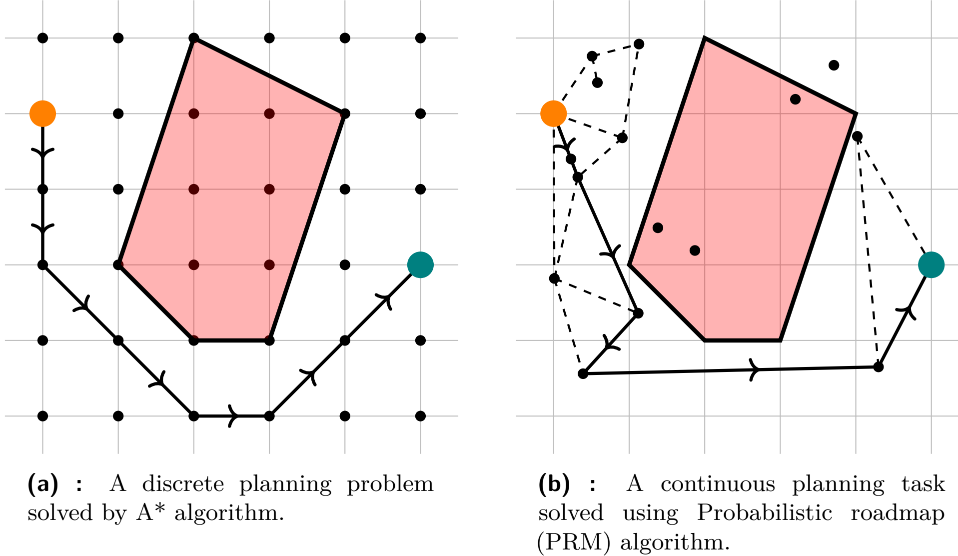
For planning in a discrete configuration space, an exhaustive search can be used (e.g., Breadth-first search), but it scales poorly with the number of possible states; for large domains, a search guided towards the goal (informed search, search with heuristics) is a better choice (e.g., A* [7], see the Figure 2.1a). The discrete-space planning is well-studied and contemporary research focuses mainly on improving the heuristics.

In continuous planning, we are interested in finding a path (for example for a robot's body) in a continuous configuration space. The problem is more challenging because the space contains infinite states. Usually, the space $\mathcal{S}$ is partitioned into two sets; obstacles $\mathcal{S}_{\text{obstacle}} \subset \mathcal{S}$ and the free space $\mathcal{S}_{\text{free}} = \mathcal{S} \setminus \mathcal{S}_{\text{obstacle}}$. We are trying to find a path that connects the start state to the goal state which is feasible (included in $\mathcal{S}_{\text{free}}$) or robustly feasible (included in the free space with some clearance $\delta > 0$).

2.2 Sampling-based planning

For the continuous domains, sampling-based planning is a common choice [8]. Random points are sampled and connected into a roadmap (PRM, Probabilis-

tic Roadmap [9], see the Figure 2.1) or a tree (RRT, Rapidly exploring Random Trees [5]). For continuous asymptotically optimal planning, informed variants of sampling-based algorithms were developed, such as PRM* and RRT* [10].



(a) : A discrete planning problem solved by A* algorithm.

(b) : A continuous planning task solved using Probabilistic roadmap (PRM) algorithm.

Figure 2.1: Comparison of discrete and continuous planning. A discrete planning algorithm searches an 8-connected grid. A continuous planning algorithm samples points in the free space and connects them to form a roadmap.

RRT* incrementally builds a tree through the free space. Random points are sampled iteratively and connected to the nearest point in the tree. After each extension of the tree, nearby points are reconnected to shorten the tree branches. See the Algorithm 1.

Algorithm 1: RRT*

Require: $x_{\text{start}} \in \mathcal{S}_{\text{free}}$ starting point, $K > 0$ step count, $\delta > 0$ step size

1: **procedure** RRT*$(x_{\text{start}}, K, \delta)$
2: AddVertex$(\text{T}, x_{\text{start}})$
3: **for** $k = 1$ to K **do**
4: $x_{\text{rand}} \leftarrow$ RandomState()
5: $x_{\text{near}} \leftarrow$ NearestNeighbor$(\text{T}, x_{\text{rand}})$
6: $u \leftarrow$ SelectInput$(x_{\text{rand}}, x_{\text{near}})$
7: $x_{\text{new}} \leftarrow$ NewState$(x_{\text{near}}, u, \delta)$
8: AddVertex$(\text{T}, x_{\text{new}})$
9: $X_{\text{near}} \leftarrow$ NearVertices$(\text{T}, x_{\text{new}})$
10: $x_{\text{min}} \leftarrow$ ChooseParent$(x_{\text{new}}, X_{near})$
11: Rewire$(\text{T}, x_{\text{min}}, x_{\text{new}}, X_{\text{near}})$
return T

It was proven that RRT* finds a robustly feasible path in a Euclidean space if some robustly feasible path exists in the limit of infinite samples (probabilis-

tic completeness) and that the path is optimal (asymptotic optimality) [10]. We will be using RRT* for generating some of our training datasets.

All the above-mentioned algorithms are general and require fine-tuning for the problem domain. For example, moving a robotic arm with multiple joints constrained by kinematics does not allow for uniform sampling and does not have a simple metric to be used in the RRT algorithm, thus violating the assumptions of the algorithm.

On the other hand, sampling-based planning algorithms can be used in spaces that would be impossible to efficiently discretize, for example, if the configuration space has a high dimensionality or is non-Euclidean. Also, non-uniform sampling can be used to guide the search toward the goal [8], around obstacles [11] or otherwise steer the search according to the user's needs.

Combining multiple planning algorithms for better initialization, faster convergence, or optimality is not unusual. For an overview of the sampling-based planning algorithms, see the survey [12].

2.3 Planning for robots

In this work, we are mainly concerned with planning for robots. Robots are generally specified by their physical model, and the generated plan is required to respect the robot's kinematics. Since there is no general solution to inverse kinematics [13], we must resort to other planning methods. A sampling-based planner is again a popular choice. For example, formulating the free space as a manifold in the configuration space, states can be sampled with Randomized Gradient Descend [14], by building a tangent space atlas [15] or by many more methods developed in recent years [16].

2.4 Transformer models

Transformer neural network architecture is currently the state of the art in sequence prediction. Shortly after its introduction in 2017 by Google Research team [17], Transformer models became used across many domains. Generative Pre-trained Transformer (GPT) [18] is currently the best-performing architecture for natural text generation. OpenAI Codex [19] is the most widely used code-completion language model. DINOv2 [20] showed that Transformers can extract information from images. AlphaFold [21] is a leading protein structure prediction model. Transformers were even shown to perform well on a wide range of out-of-distribution tasks without explicit pre-training [22].

2.5 Denoising Diffusion Probabilistic Models

Denoising Diffusion Probabilistic Model (DDPM) emerged in 2015 [6] as an alternative to Generative Adversarial Network (GAN) [23]. The general idea is to train a denoising model on a dataset with added Gaussian noise. The model can then generate output similar to its training dataset from pure noise.

DDPM is a very general framework suitable for many different domains. Following the success of VAEs, DDPMs soon achieved state-of-the-art results in image generation (DALL-E [24], DALL-E 2 [25], Stable Diffusion [26]), audio synthesis (DiffWave [27]), and video generation (Stable Video Diffusion [28], Sora [29]). See the Figure 2.2 for an example of a text-to-image DDPM model.

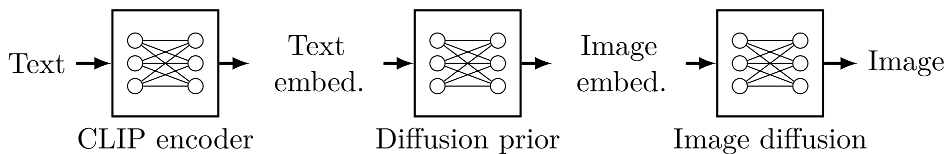


Figure 2.2: The unCLIP [25] inference process used in the DALL-E 2 model. The process contains a Contrastive Language-Image Pre-training (CLIP, [22]) embedding and two Denoising Diffusion models.

2.6 Path planning using neural networks

Recently, artificial neural networks have been successful in solving Reinforcement Learning (RL) tasks (robotic hand dexterity [30], Atari games [31], real-time multi-player game [32]). Furthermore, neural networks can be used to initialize, guide or even replace traditional planning algorithms [33].

Searchformer [34] is a Transformer network trained to imitate the steps of the A* algorithm and output a solution plan. Furthermore, the same network is used to bootstrap more training data, improving the plan length until achieving superior accuracy on 30×30 Sokoban mazes.

Vector Quantized-Motion Planning Transformers [35] use Vector Quantized model [36] to split the configuration space into discrete areas. A transformer model then guides a sampling algorithm (e.g., RRT) by selecting relevant areas and reconstructing them as probability distributions in the robot's configuration space, see the Figure 2.3.

Transformer models can also be trained to predict the following action given the past path. Janner et al. [37] provide a novel view on planning as a step-by-step prediction task.

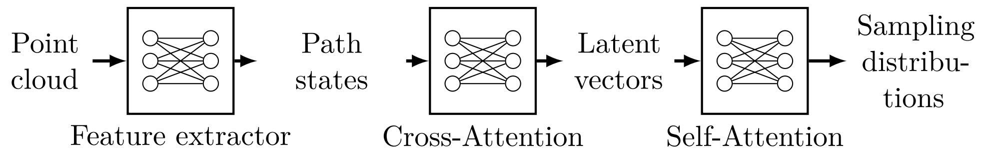


Figure 2.3: The Vector Qunatized-Motion Planning Transformer [35] use two Transformer models to suggest ideal sampling distribution from a dictionary of pre-trained areas.

Planning using neural networks is still a very active research area with many open questions. In this work, we will be working with one of the few neural network models that can generate a full plan from scratch and we will show different ways to use it for planning in continuous space.

Chapter 3

Diffusion models

In this section, we will explain the key points of Denoising Diffusion Probabilistic Models, mostly following definitions by Nichol [38].

The DDPM framework consists of two steps that achieve the opposite goals—forward diffusion and reverse diffusion. The forward diffusion step randomly adds noise to the data, while the reverse diffusion step tries to predict the noise and recover the original sample.

3.1 Forward diffusion

Forward diffusion is the process of taking a sample from the input distribution $x^0 \sim q(x)$ and performing T diffusion steps to obtain the output distribution

$$q(x^{1:T}) = q(x^0) \prod_{t=1}^{T} q(x^t \mid x^{t-1}). \tag{3.1}$$

Note that x can be a multidimensional tensor (e.g., path, image); therefore, we denote the diffusion step t (also called time or time step) using superscript and the tensor index using subscript.

The forward diffusion step q probability function can be arbitrary, Ho et al. [39] suggest a normal distribution

$$q(x^t|x^{t-1}) = \mathcal{N}(x^t; \sqrt{1-\beta^t}x^{t-1}, \beta^t \boldsymbol{I}) \tag{3.2}$$

for some constants $\beta^t \in (0,1)_{t=1}^{T}$, called the variance schedule. Setting $\alpha^t = 1-\beta^t$ and $\bar{\alpha}^t = \prod_{i=1}^{t} \alpha^i$ we can directly calculate the probability distribution after the first t forward diffusion steps as

$$q(x^t \mid x^0) = \mathcal{N}(x^t; \sqrt{\bar{\alpha}^t}x^0, (1-\bar{\alpha}^t)\boldsymbol{I}). \tag{3.3}$$

Therefore given a data point x^0 and a Gaussian noise sample $\epsilon^t \sim \mathcal{N}(0, I)$ we can calculate the output after the first t forward diffusion steps as

$$x^t = \sqrt{\bar{\alpha}^t} x^0 + \sqrt{1 - \bar{\alpha}^t} \epsilon^t. \tag{3.4}$$

This efficient calculation justifies the choice of the normal distribution for the forward diffusion step. The forward diffusion can be calculated in a closed form with no trainable parameters.

The forward diffusion step can be seen as adding a small amount of Gaussian noise to the input data. The output after T steps for $T \to \infty$ approaches isotropic Gaussian distribution, effectively replacing the initial sample x with pure noise. Under this assumption, the forward diffusion acts as an interpolation between the input data and the noise distribution. See the Figure 3.1a for an illustration.

3.2 Reverse diffusion

Knowing the forward diffusion step formula, we can build its inverse mapping – the reverse diffusion. Again, the usual choice due to Ho et al. [39] is a normal distribution

$$p_\theta(x^{t-1} \mid x^t) = \mathcal{N}(x^{t-1}; \mu_\theta(x^t, t), \Sigma_\theta(x^t, t)). \tag{3.5}$$

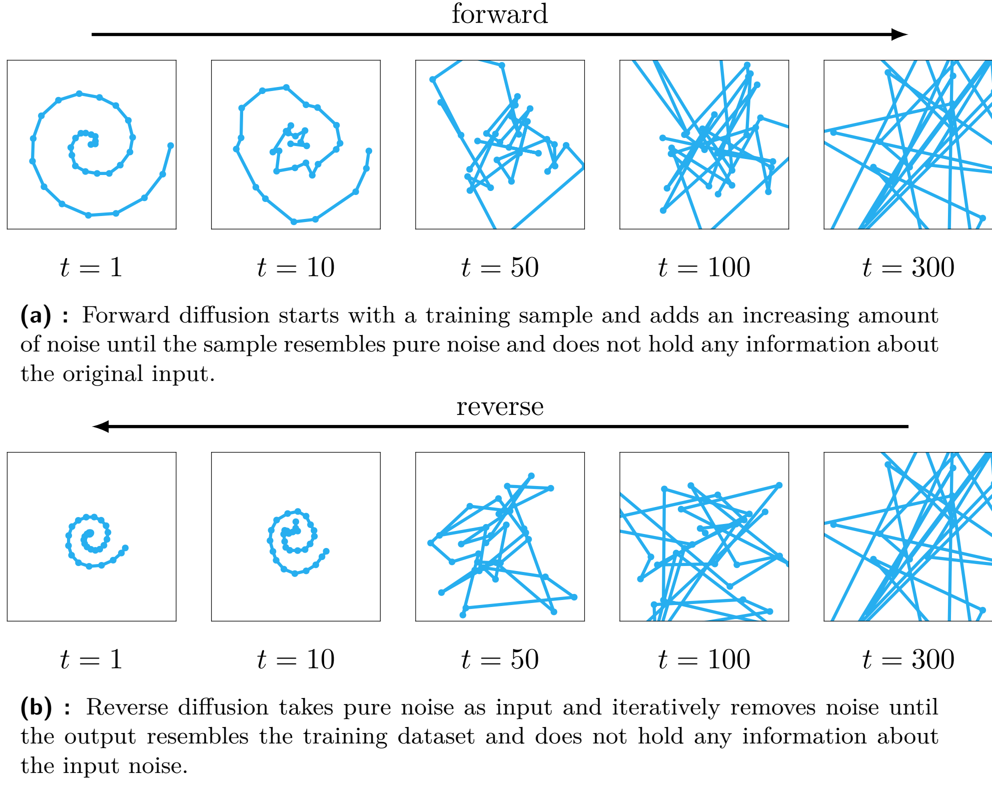
The distribution parameters μ_θ and Σ_θ cannot be calculated in closed form and must instead be approximated by a suitable model parametrized by θ.

The reverse diffusion can be seen as removing a small amount of noise from the input data. Given that the model is trained properly, the output after T reverse steps should resemble the training dataset distribution. See the Figure 3.1b for an illustration.

3.3 Training

Following [38], we can view the combination of p and q as a Variational Auto-Encoder and derive its variational lower bound (ELBO [40, 41]) as

$$\begin{aligned} L(x; \theta) &= \sum_{t=0}^{T} L^t \\ L^0 &= -\log p_\theta(x^0 \mid x^1) \\ L^{t-1} &= D_{KL}(q(x^{t-1} \mid x^t, x^0) \parallel p_\theta(x^{t-1} \mid x^t)) \quad \text{for } 2 \leq t \leq T \\ L^T &= D_{KL}(q(x^T \mid x^0) \parallel p(x^T)), \end{aligned} \tag{3.6}$$



(a) : Forward diffusion starts with a training sample and adds an increasing amount of noise until the sample resembles pure noise and does not hold any information about the original input.

(b) : Reverse diffusion takes pure noise as input and iteratively removes noise until the output resembles the training dataset and does not hold any information about the input noise.

Figure 3.1: Illustration of the forward and reverse diffusion processes.

where the posterior is

$$
\begin{aligned}
q(x^{t-1} \mid x^t, x^0) &= \mathcal{N}(x^{t-1}; \tilde{\mu}(x^t, x^0), \tilde{\beta}^t \boldsymbol{I}), \\
\tilde{\beta}^t &= \frac{1 - \bar{\alpha}^{t-1}}{1 - \bar{\alpha}t} \beta^t, \\
\tilde{\mu}^t(x^t, x^0) &= \frac{\sqrt{\bar{\alpha}^{t-1}}\beta^t}{1 - \bar{\alpha}^t} x^0 + \frac{\sqrt{\alpha^t}(1 - \bar{\alpha}^{t-1})}{1 - \bar{\alpha}^t} x^t.
\end{aligned}
\tag{3.7}
$$

The term L^0 can be calculated by discretizing the space as proposed by several authors [42, 43].

Example 1: Image pixel discretization

Ho et al. [39] rescale image pixel values to $[-1, 1]$ and discretize them into 256 bins. The normal distribution is integrated over the selected bin

in each of the D dimensions to obtain the probability product

$$
\begin{aligned}
p_\theta(x^0 \mid x^1) &= \prod_{i=1}^{D} \int_{\delta_-(x_i^0)}^{\delta_+(x_i^0)} \mathcal{N}(x; \mu_\theta^1(x^1,1)_i, \Sigma_\theta^1(x^1,1)_i)\, \mathrm{d}x\,, \\
\delta_-(x) &= \begin{cases} -\infty & \text{if } x = -1, \\ x - \frac{1}{255} & \text{if } x > -1, \end{cases} \\
\delta_+(x) &= \begin{cases} \infty & \text{if } x = 1, \\ x + \frac{1}{255} & \text{if } x < 1. \end{cases}
\end{aligned}
\tag{3.8}
$$

To summarize, L^0 can be calculated by discretizing the space, L^T does not depend on θ, and the rest of the terms are Kullback–Leibler divergences between two Gaussians.

Instead of predicting the $\tilde{\mu}^t(x^t, x^0)$ directly, we instead predict the noise $\epsilon_\theta(x^t, t)$ from the following step x^t and subtract it from the noisy data. Using the Equation 3.4 we can rewrite the mean

$$
\mu_\theta^t(x^t, t) = \frac{1}{\sqrt{\alpha^t}} \left(x^t - \frac{1 - \alpha^t}{\sqrt{1 - \bar{\alpha}^t}} \epsilon_\theta(x^t, t) \right) \tag{3.9}
$$

and use it to predict the previous step

$$
x^{t-1} \sim \mathcal{N}(x^{t-1}; \mu_\theta^t(x^t, t), \Sigma_\theta(x^T, t)). \tag{3.10}
$$

Following [39] for best results we fix $\Sigma_\theta(x^T, t) = \beta^t \boldsymbol{I}$ and ignore any additive and multiplicative constants to derive the simplified training loss

$$
\begin{aligned}
L_{\text{simple}}^t &= \mathbb{E}_{t, x^0, \epsilon^t} \left[||\epsilon^t - \epsilon_\theta(x^t, t)||^2 \right] \\
&= \mathbb{E}_{t, x^0, \epsilon^t} \left[\left\| \epsilon^t - \epsilon_\theta(\sqrt{\bar{\alpha}^t} x^0 + \sqrt{1 + \bar{\alpha}^t} \epsilon^t, t) \right\|^2 \right],
\end{aligned}
\tag{3.11}
$$

where $1 \leq t \leq T$ is a sampled time step.

Therefore the only model we need is a predictor of the noise $\epsilon_\theta(x^t, t)$ from the noisy data x^t and the time step t. We will be using a neural network model to predict the noise in this work.

The complete training loop is as follows. Sample x^0 from the training set, sample t uniformly, sample $\epsilon^t \sim \mathcal{N}(0, \boldsymbol{I})$, calculate x^t, predict $\epsilon_\theta(x^t, t)$, and take a gradient descend step with respect to L_{simple}^t from the Equation 3.11. See the complete training algorithm in the Algorithm 2 and a diagram in the Figure 3.2.

The training can be parallelized by sampling multiple training examples, multiple time steps and multiple noise samples at once. The gradient step can then be performed with stochastic gradient descent or replaced by any other optimization algorithm which decreases the model's loss. We use the Adam optimizer [44] in our experiments.

Algorithm 2: Diffuser training

Require: DDPM ϵ_θ parametrized by θ, $\mathcal{T}$ training set, $T \in \mathbb{N}$ diffusion length, $(\bar{\alpha}^t)_{t=1}^T$ variance schedule

1: **procedure** TRAIN($\epsilon_\theta, \mathcal{T}, T, (\bar{\alpha}^t)_{t=1}^T$)
2: **repeat**
3: $x^0 \sim \mathcal{T}$ ▷ *Sample from the training set*
4: $t \sim \text{Uniform}(1, \dots, T)$ ▷ *Sample a diffusion time*
5: $\epsilon^t \sim \mathcal{N}(0, \boldsymbol{I})$ ▷ *Sample noise*
6: $x^t \leftarrow \sqrt{\bar{\alpha}^t} x^0 + \sqrt{1 - \bar{\alpha}^t} \epsilon^t$ ▷ *Add noise to the data, Equation 3.4*
7: $\hat{\epsilon}^t \leftarrow \epsilon_\theta(x^t, t)$ ▷ *Predict the noise from the noisy data*
8: Take a gradient step on $\nabla_\theta \left\| \epsilon^t - \hat{\epsilon}^t \right\|^2$ ▷ *Update the model*
9: **until** convergence
10: **return** θ

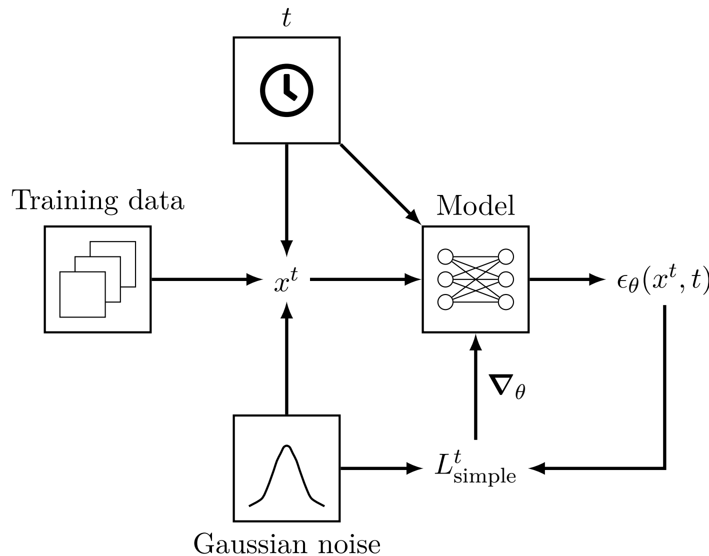


Figure 3.2: DDPM training diagram showing the relationship between the training data, noise and diffusion time as inputs, and predicted noise as output.

3.4 Inference

Having a trained noise predictor ϵ_θ, we can generate new data generalizing the training set. The inference formula cannot be calculated in a closed form and instead must be performed by T denoising steps.

Initialize the inference with a pure noise sample $x^T \sim \mathcal{N}(0, \boldsymbol{I})$ and then for $t = T, \dots, 2$ sample additional noise $z^t \sim \mathcal{N}(0, \boldsymbol{I})$. Rewriting the Equation 3.5 in terms of sampled data using the formulas for the forward diffusion,

we calculate the previous sample

$$x^{t-1} = \frac{1}{\sqrt{\alpha^t}} \left(x^t - \frac{1-\alpha^t}{\sqrt{1-\bar{\alpha}^t}} \epsilon_\theta(x^t, t) \right) + \sqrt{\beta^t} z^t. \tag{3.12}$$

After the last iteration, we make a final step with $t = 1$ and $z^1 = 0$ and return the output as the denoised sample. See the complete inference algorithm in the Algorithm 3 and a diagram in the Figure 3.3.

Algorithm 3: Diffuser inference

Require: DDPM ϵ_θ, $T \in \mathbb{N}$ diffusion length, $(\alpha^t)_{t=1}^T$ and $(\bar{\alpha}^t)_{t=1}^T$ variance schedule

1: **procedure** GENERATE($\epsilon_\theta, (\alpha^t)_{t=1}^T, (\bar{\alpha}^t)_{t=1}^T$)
2: $\quad x^T \sim \mathcal{N}(0, \boldsymbol{I})$ ▷ *Sample noise*
3: $\quad$ **for** $t = T, \dots, 2$ **do**
4: $\quad\quad z^t \sim \mathcal{N}(0, \boldsymbol{I})$ ▷ *Sample noise*
5: $\quad\quad x^{t-1} \leftarrow \frac{1}{\sqrt{\alpha^t}} \left(x^t - \frac{1-\alpha^t}{\sqrt{1-\bar{\alpha}^t}} \epsilon_\theta(x^t, t) \right) + \sqrt{\beta^t} z^t$ ▷ *Equation 3.12*
6: $\quad x^0 \leftarrow \frac{1}{\sqrt{\alpha^1}} \left(x^1 - \frac{1-\alpha^1}{\sqrt{1-\bar{\alpha}^1}} \epsilon_\theta(x^1, 1) \right)$ ▷ *Final denoising step*
7: $\quad$ **return** x^0

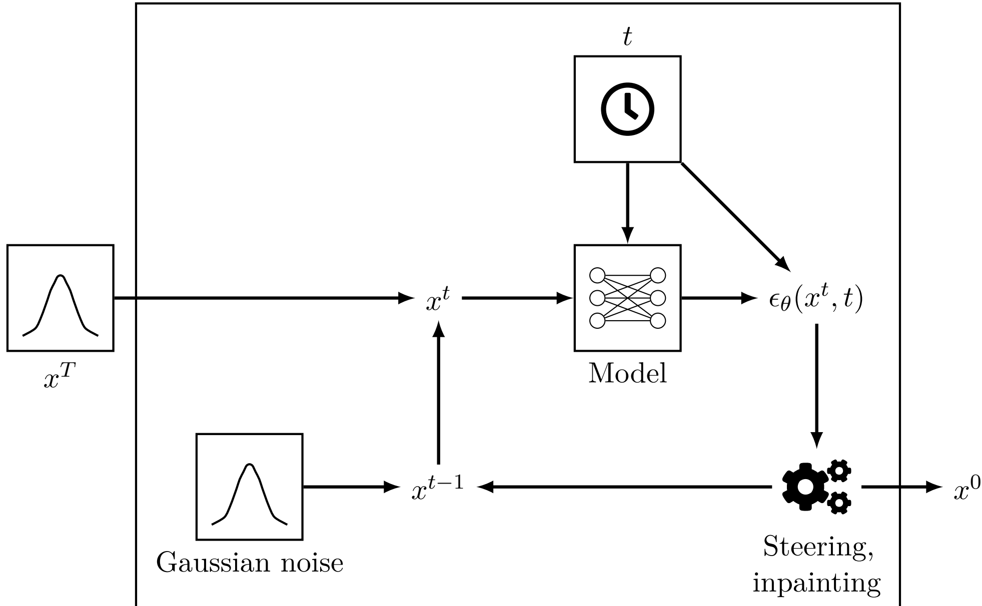


Figure 3.3: DDPM inference diagram showing the denoising loop iteratively removing noise until a final sample is returned.

Note that inference can also be parallelized by sampling multiple noise samples at once and calculating the denoising step for all of them in parallel. This method produces a batch of generated samples and is suitable for modern GPU architectures.

3.4.1 Inpainting

Inpainting is the process of filling in the missing parts of an image. The idea was first introduced by Pathak et al. [45] long before DDPMs. At that time, GANs with special layer architecture were used to capture the context of the image and fill in the damaged parts of photos with a realistic texture.

The first DDPM-based inpainting without any special architecture or training changes is RePaint [46]. They proposed to denoise the fixed and unknown parts of the image separately. The unknown parts follow the standard DDPM inference, while the fixed parts contain a noisy inpainting mask with a fixed schedule

$$x^{t-1}_{\text{fixed}} = \sqrt{\bar{\alpha}^t x_{\text{mask}}} + (1 - \bar{\alpha}^t)z. \tag{3.13}$$

This ensures that the fixed parts of the image converge to the intended mask at the end of the diffusion process. The image is obtained by combining the fixed and unknown parts after each denoising step.

A similar approach can be used for constraining the generated path in our setting. For example, we want to fix the start and goal points and let the model generate the path between them. Janner [47] proposed to set the desired points to the mask value after each denoising step; see an example in the Figure 3.4. In our experiments, this worked well.

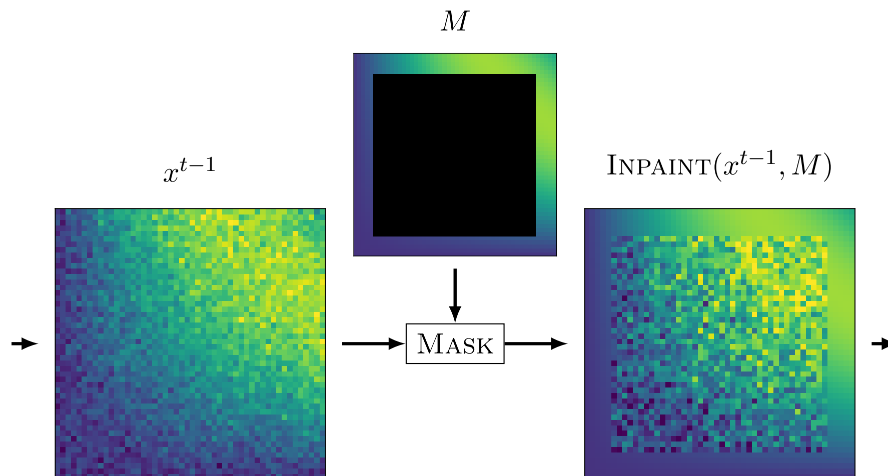


Figure 3.4: Inpainting example on 2D color gradient. Outer pixels are fixed in the mask M and reset using the Mask function. The pixels in the middle are taken from the model output.

3.4.2 Steering

Another important aspect of generative models is the ability to control the generated samples. For example, Choi et al. [48] propose a latent variable method that allows influencing the resulting image including a style

specification, image interpolation and converting paintings into photorealistic images.

We use a method called steering inspired by Carvalho et al. [49]. A steering function is a differentiable function that predicts the score of a sample. After each denoising step, we apply the steering function and take a gradient step to maximize the score. This way, we can guide the final output towards the desired direction.

Combining DDPMs with a steering function is a powerful tool since it allows us to generate samples that are both realistic and fulfill user-defined constraints. It can act as an optimization tool that optimizes both locally and globally. See the complete inference algorithm with steering in the Algorithm 4.

Algorithm 4: Diffuser inference with steering and inpainting

Require: DDPM ϵ_θ, $T \in \mathbb{N}$ diffusion length, $(\alpha^t)_{t=1}^T$ and $(\bar{\alpha}^t)_{t=1}^T$ variance schedule, STEER steering function, M inpainting mask

1: **procedure** GENERATE(ϵ_θ, $(\alpha^t)_{t=1}^T$, $(\bar{\alpha}^t)_{t=1}^T$, STEER, M)
2: $x^T \sim \mathcal{N}(0, \boldsymbol{I})$
3: **for** $t = T, \dots, 2$ **do**
4: $z^t \sim \mathcal{N}(0, \boldsymbol{I})$
5: $x^{t-1} \leftarrow \frac{1}{\sqrt{\alpha^t}} \left(x^t - \frac{1-\alpha^t}{\sqrt{1-\bar{\alpha}^t}} \epsilon_\theta(x^t, t) \right) + \sqrt{\beta^t} z^t$
6: $x^{t-1} \leftarrow$ STEER(x^{t-1})
7: $x^{t-1} \leftarrow$ INPAINT(x^{t-1}, M)
8: $x^0 = \frac{1}{\sqrt{\alpha^1}} \left(x^1 - \frac{1-\alpha^1}{\sqrt{1-\bar{\alpha}^1}} \epsilon_\theta(x^1, 1) \right)$
9: $x^0 \leftarrow$ STEER(x^0)
10: $x^0 \leftarrow$ INPAINT(x^0, M)
11: **return** x^0

Chapter 4

Analysis of existing implementations

In this chapter, we will analyze the existing implementations of the Diffuser architecture, their shortcomings, and the reasons for rewriting the Diffusers library. We will also describe the changes we made to the library and the new features we added.

4.1 The Diffuser architecture

The noise predictor $\epsilon_\theta(x^t, t)$ from the Algorithm 2 is usually chosen to be a neural network. A usual architecture is the U-Net, first introduced by Ronneberger [50]. U-Net is a residual convolutional neural network composed of down-scaling blocks, followed by a bottleneck (middle block) and up-scaling blocks. The output of each down-scaling block is added to the input of the respective up-scaling block, forming residual connections. Furthermore, the blocks themselves can contain residual connections allowing efficient state propagation through the network during the early training stages. See the diagram in the Figure 4.1.

The U-Net architecture was shown by its authors to improve learning in image segmentation tasks but can be adapted to other domains. It is important for our work that the U-Net is fully convolutional, meaning that the input and output shapes can vary, and the network can be applied to any input size. It is also symmetric in the sense that the input and output shapes are the same.

We will follow the architecture by Janner called the Diffuser [47]. In their work, the main building block is the Residual Temporal Block consisting of four convolution layers with Mish [51] activation function, residual connections [52] and group normalizations [53]. The diffusion time t is embedded into 64 dimensions and added between the convolutional layers. The Down-scaling Block consists of two Residual Temporal Blocks followed by a Convolution layer. The Middle Block consists of two Residual Temporal Blocks

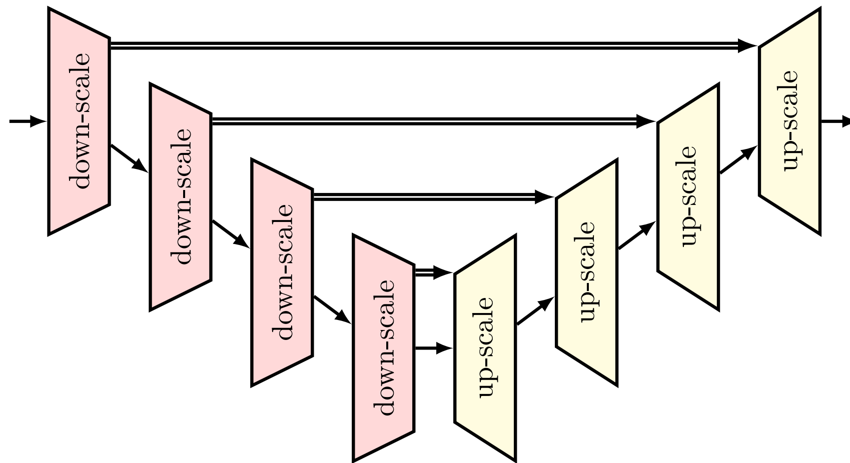


Figure 4.1: Simplified architecture diagram of the original U-Net. The input is passed through down-scaling blocks and then up-scaled back to the original size. The output of each down-scaling block is also added to the input of the respective up-scaling block, forming residual connections.

only. The Up-scaling Block consists of two Residual Temporal Blocks and a transpose convolution layer mirroring the convolution layer in the Down-scaling Block.

The whole architecture is a fully convolutional 1D network. 1D means that the intended output is a path with convolutions along the path length as opposed to the usual use in images, which are convolved along two axes. If the path is in a multi-dimensional space, the dimensions are represented as convolution channels. This representation allows, for example, to model a robotic manipulator with six degrees of freedom for a 32-step horizon as a path of shape 6×32. In our experiments, we will be mainly interested in path planning for a point robot in a 2D space, so the path will be of shape 2×32.

4.2 State of the implementation

At the time of writing this work, there were two publicly available implementations of the Diffuser.

First, a code from the paper's authors is available on GitHub[1]. It, however, contains many errors and software bugs and depends on old versions

[1] https://github.com/jannerm/diffuser

of libraries (namely `mujoco`[2], `gym`[3], `d4rl`[4]). Despite several complaints in the issue tracker, the authors have not updated the project for the last two years. The architecture is only briefly touched on in Appendix A [47], not mentioning details regarding the precise layer composition, settings, and dimensions. Many of them are also unassigned variables in the file. We did not find the implementation complete enough to recreate a full training setup.

Second, the project was adapted by Diffusers following the success of the conference talk about the Diffuser. Diffusers[5] [60] by Hugging Face is an Apache-licensed implementation of many different diffusion models in a unified framework. In 2022, Lambert adapted[6] Janner's work and added it to Diffusers with a pre-trained model. Again, this implementation did not catch up with the fast-paced ecosystem evolution and is neither complete nor easily reproducible. Lambert's adaptation closely followed the original code, resulting in a convoluted implementation that does not fit the rest of the Diffusers project well. It does not share much of its code with 2D and 3D diffusion architectures that form most of the Diffusers' core and are much more advanced regarding the number of features and hyper-parameters supported.

We decided to base our work on the second option, the Diffusers library. During initial testing with dummy datasets, we were struggling with dimensionality errors. The correct shape of the input and output data was not apparent, and we could not modify the size or remove layers. We never achieved a proof of concept run. After thoroughly inspecting the code, we found that instead of adapting the existing 2D Residual Blocks, a minimal handwritten implementation is used with hard-coded layers, hyper-parameters, and dimensions.

We decided that this implementation is insufficient for our experiments and needs updating. We started with planning the rewrite in the form of a Pull Request on GitHub[7], but Diffusers' maintainers quickly stopped us. Their project vision is to collect popular diffusion models, and a large refactor did not bring any value for them, only the burden of reviewing and maintaining our changes.

We do not share their view; in our opinion, the code could be much simpler but also more powerful, modular, and almost identical to the existing 2D diffusion implementation. A rewrite would make both the code review and

[2]MuJoCo [54] was heavily updated by Google's Deep Mind team and is difficult to install considering compatibility between system library and Python bindings. `https://mujoco.org`

[3]Gym [55] library by OpenAI is deprecated in favor of Gymnasium [56] by Farama Foundation. Both projects currently co-exist and maintain only partial compatibility. `https://www.gymlibrary.dev`, `https://gymnasium.farama.org`

[4]D4RL [57] by Farama Foundation was an experimental reinforcement learning benchmark library, now replaced by Gymnasium-Robotics [58] and Minari [59]. `https://github.com/Farama-Foundation/D4RL`, `https://robotics.farama.org`, `https://minari.farama.org`.

[5]`https://huggingface.co/docs/diffusers`

[6]`https://github.com/huggingface/diffusers/pull/105`

[7]`https://github.com/huggingface/diffusers/pull/5482`

further expansions easier. However, we respected their viewpoint and continued our work in a separate repository. We forked the library on December 17th, 2023, at the commit `9cef07`[8] in Diffusers version 0.24.0. When this work was published, more diffusion models had been added to the Diffusers library, and several rounds of refactoring had already taken place in the 1D code. However, most of the functionality is still in the state described above.

4.3 Our contribution to the Diffusers

The main reason to rewrite the Diffusers library was to address the lack of hyper-parameter tuning (e.g., the number of group normalization groups was hard-coded) and the removal of hand-written blocks where standardized implementations exist.

The Diffusers library contained the following classes for the 1D diffusion model:

- `UNet1DModel`: the main class representing a U-Net model. It takes the configuration of the other blocks as parameters.
- `DownResnetBlock1D`, `UpResnetBlock1D`: containers for a sequence of Residual Temporal Blocks with some additional layers.
- `UNetMidBock1D`: a special block for the middle of the U-Net containing optional attention.
- `ResidualTemporalBlock1D`: a contained block with two convolutions, group normalization, and activation.
- `ResConvBlock`: a special block for the middle of the U-Net.

First, we made sure that all primary building blocks existed. Notably, a LoRA-compatible 1D convolution layer was missing as opposed to the 2D diffusion module. Low-Rank Adaptation (LoRA) [61] is a technique to speed up model fine-tuning by replacing layers with their respective low-rank approximation with much fewer parameters. Hugging Face ecosystem has a package called PEFT [62] for automatic LoRA usage, but a manual compatibility layer is still present for potential usage without PEFT. We split the existing class `LoRACompatibleConv` into `LoRACompatibleConv2d` and `LoRACompatibleConv1d`, each providing the LoRA convolution layer of their respective type. The Diffusers library is heavily tested, so we added unit tests for both layers.

Having LoRA-compatible convolution layers, we updated both `Upsample1D` and `Downsample1D` layers to match their 2D counterparts. The existing behavior was retained and compatibility with LoRA and configurable padding was added.

Next, we found that while 2D models used a generic `ResnetBlock2D`, the 1D model used a much simpler `ResConvBlock` with hard-coded layers, GELU activation function, kernel sizes, and padding. See the diagram in the Figure 4.2.

[8] `https://github.com/huggingface/diffusers/tree/9cef07da5a1f28ddfeadd232ede9a815c7945aec`

The most problematic part was the omnipresent assumption of the network architecture. There were specific branches for the first and last blocks in the U-Net with fixed input and output shapes. Changing the shape or the number of layers in the `ResConvBlock` was impossible.

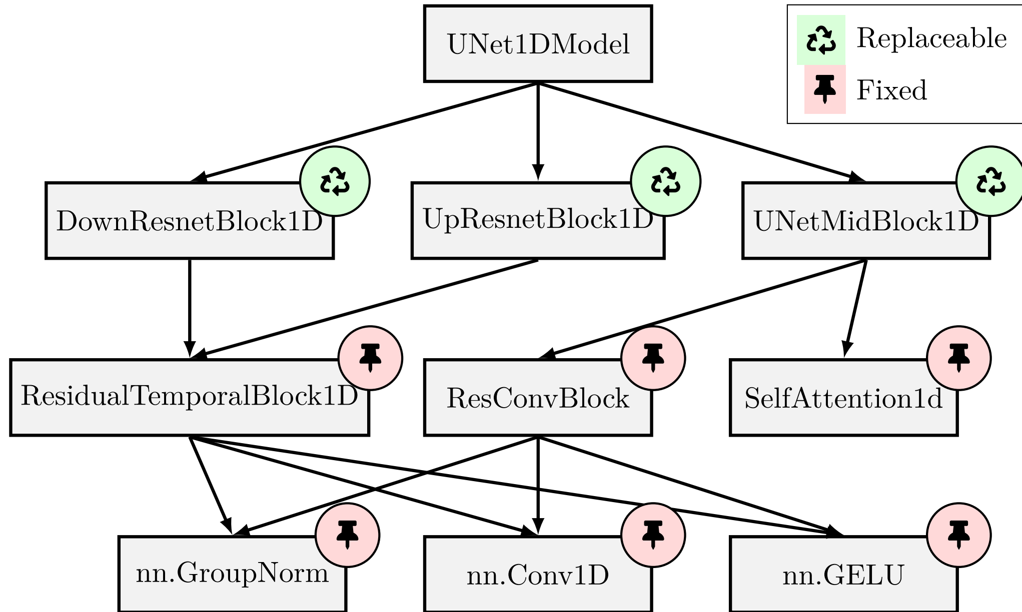


Figure 4.2: Simplified diagram of the original class composition in Diffusers library. Note that several blocks, such as `ResidualTemporalBlock1D` and `ResConvBlock`, are similar but not identical and used in different contexts. Also note that most of the depicted blocks contain additional convolutions, activation, and normalization layers, which cannot be configured from the outside.

We, therefore, created a new `ResnetBlock1D` mirroring the parameters and usage of the 2D variant, allowing more flexible and concise usage. A universal Resnet block allowed us to improve the up- and down-blocks. Initially, the implementation contained several customized block types following Janner's code. Convoluted block architecture does not scale well if the shape of the data is not known beforehand. Taking the library as-is, we could not run a single forward with any shape different from the one provided in the example without a runtime error. We instead refactored the code to contain a single `UpBlock1D` and a single `DownBlock1D` modeled after the 2D implementation in Diffusers. The tensor shapes are now calculated dynamically, and the blocks can be used in any order.

Previously, only the number of input and output channels was settable via constructor parameters. After our refactor, one could specify temporal embedding channels, number of Resnet Blocks, custom activation function, dropout rate, number of normalization groups, output rescaling, up-/down-sampling, and padding. Most of the code again was borrowed from the 2D implementation and reused with only minor changes. We did not need some advanced and dimension-sensitive 2D features (e.g., interpolation kernels),

so we did not port them to the 1D implementation; see Figure 4.3 for an updated architecture diagram.

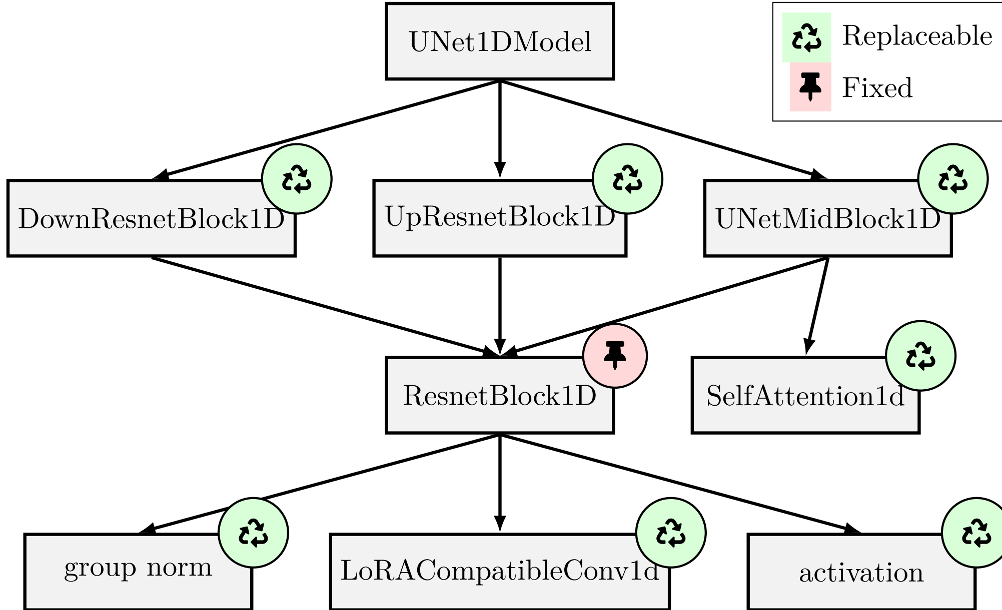


Figure 4.3: Simplified diagram of the class composition in Diffusers library after our rewrite. All residual blocks were merged into a parametric `ResnetBlock1D`. Also note that there are no more hard-coded layers; convolutions, activation, and normalizations are configurable in each block and ultimately shared throughout the whole network.

Citing Janner [47], the horizon of the path is not bounded by the model architecture because the model is fully convolutional. Still, their model and the code of Diffusers can optionally use Attention in their layers. Attention [63] is a central layer to the Transformer architecture [17] as well as many U-Net inspired architectures [64]. We do not aim to use Attention in our experiments as we want a model with a dynamic horizon, but the refactoring was also easy enough to add proper Attention support.

Finally, the public class `UNet1DModel` was updated to properly support middle and output block configurations, layer counting, dropout, group normalization, Attention, and scale shifting. Correctly passing all the parameters to all the layers greatly simplified the code and removed many previously hard-coded constants. The updated 1D model is comparable to a 2D diffusion model regarding features and API; see the Table 4.1. The only parts lacking proper support are interpolation kernels and some Attention variants irrelevant to our research.

The full source code of the Diffusers including our changes is available in the attachment and on GitHub[9].

[9] `https://github.com/petrzjunior/diffusers`

Feature	Original TemporalResnetBlock	Original ResnetBlock2D	Our rewritten ResnetBlock1D
input channels	✓	✓	✓
output channels	✓	✓	✓
temporal channels	✓	✓	✓
GroupNorm groups		✓	✓
GroupNorm epsilon	✓	✓	✓
dropout		✓	✓
activation function		✓	✓
time embedding normalization		✓	✓
Self-Attention		✓	✓
Cross-Attention		✓	
output scaling		✓	✓
skip-connection		✓	✓
skip-conn. bias		✓	✓
upsampling		✓	✓
downsampling		✓	✓
interpolation		✓	

Table 4.1: Comparison of features configurable in the original `TemporalResnetBlock` (1D) versus the original `ResnetBlock2D`. Our rewritten `ResnetBlock1D` takes inspiration from `ResnetBlock2D` and supports nearly all major features.

Chapter 5

Diffuser Architecture

This chapter explains the architecture of the neural network trained to denoise paths. We described the building blocks used in the Diffusers library and our modifications. to our knowledge, this is the most complete publicly accessible description of a diffuser model architecture.

5.1 Neural network architecture

The final architecture chosen for this work follows a one-dimensional U-Net as described previously. It consists of three down-scaling blocks, a middle block, three up-scaling blocks, and an output block, connected with skip connections in a symmetrical shape; see the diagram in the Figure 5.1.

We found experimentally that three blocks are enough for our experiments, and adding more blocks does not improve the accuracy. Instead, the performance increases with the number of parallel channels and the quality of the training dataset.

The down-scaling part of the U-Net is implemented using `DownBlock1D` followed by a stridden convolution, which halves the path length; see the Figure 5.2a. The up-scaling part of the U-Net is built using `DownBlock1D` followed by a stridden transpose convolution, which doubles the path length; see the Figure 5.2b. The middle block `UNetMidBlock1D` does not change the tensor shape as it contains only three ResNet blocks with no additional convolutions; see the Figure 5.2c. This block can optionally include an Attention layer between each ResNet block, but we did not use Attention for experiments in this work. The output block consists of two convolutions to reshape the output state into the same shape as the network input; see the Figure 5.2d.

The skip connections copy the output of the ResNet down-blocks and concatenate them before the corresponding ResNet up-block. In Janner's original architecture, there was also a skip connection after each downscaling block

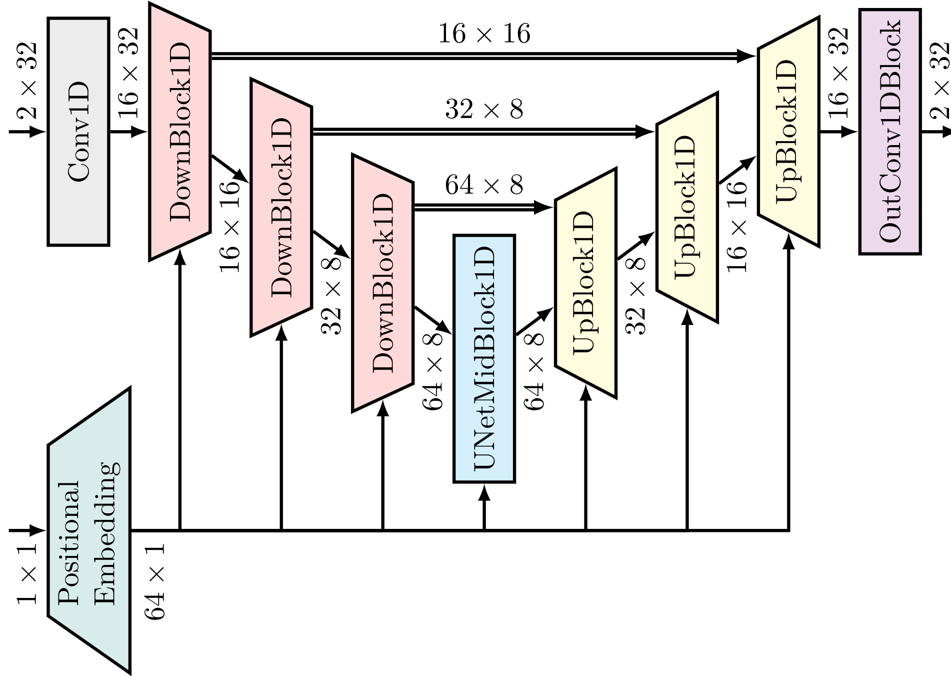
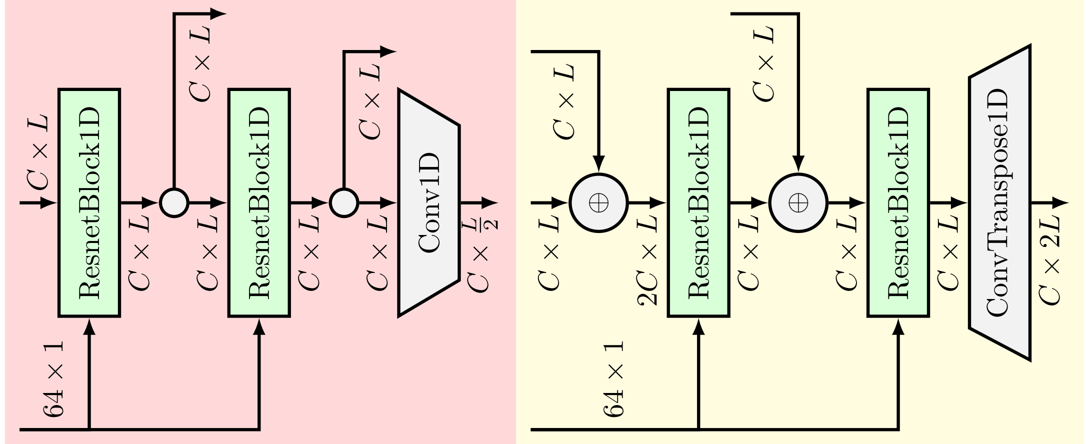


Figure 5.1: High-level architecture diagram of the diffuser neural network. The diagram shows the configuration with three down- and three up-blocks used in this work. Skip connections are depicted with double arrows. Connections are annotated with tensor sizes for a two-channel input of length 32 and a single diffusion timestep t. The denoised output has the same shape as the input.

connected to an additional ResNet up-block. We found this detail to complicate the implementation and break the network symmetry. We removed the extra block and the skip connection and observed insignificant accuracy loss.

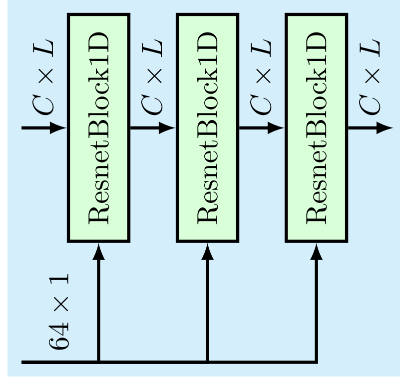
The `ResnetBlock1D` block is a typical ResNet block [52]. The block contains Group Normalization, Mish activation, and Convolutional layer. Then, the input time embedding is transformed with a single Linear layer and added to the output. Another Mish activation layer, optional Dropout, and the final Convolutional layer follow this. Finally, a shortcut with a single convolution is added to the output of the last Convolutional layer. See the detailed diagram in the Figure 5.3.

In the Diffusers library, after our rewrite, we can choose the activation function, the Group Normalization parameters, the Dropout rate, and the Attention layers; the mentioned configuration follows the choices we made for our experiments. We chose not to use Attention and instead made the model fully convolutional to remove the dependence on the path length during training. However, we usually fix the path length during training to allow efficient batching. Still, it is not fixed in the model, and we can generate paths of different lengths during the inference phase.

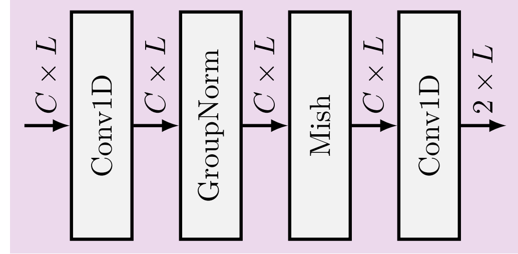


(a) : The `DownBlock1D` block with two ResNet blocks. The down-sampling convolution with stride two is included in all down-blocks in the network except the last one before mid-block.

(b) : The `UpBlock1D` block with two ResNet blocks. The up-sampling transpose convolution with stride two is included in all down-blocks in the network except the last one before the output layer.



(c) : The `UNetMidBlock1D` block with three ResNet blocks and no attention layer.



(d) : The `OutConv1DBlock` block with two convolutional layers. The second convolutional layer has kernel size one, which only reduces the number of channels to the desired output.

Figure 5.2: Diagrams of the individual blocks of the U-Net architecture in the configuration used for the experiments in this work. The blocks are formed by repeated ResNet blocks followed by a convolutional layer to increase or decrease the tensor length. The blocks receive input from the previous

5.2 Diffuser as denoiser

We take the U-Net described in the previous section and use it as a denoiser in the DDPM framework. DDPMs usually operate on two-dimensional images or three-dimensional video. In our case, we want to generate a path given as a tensor of concatenated point coordinates. A path with L points

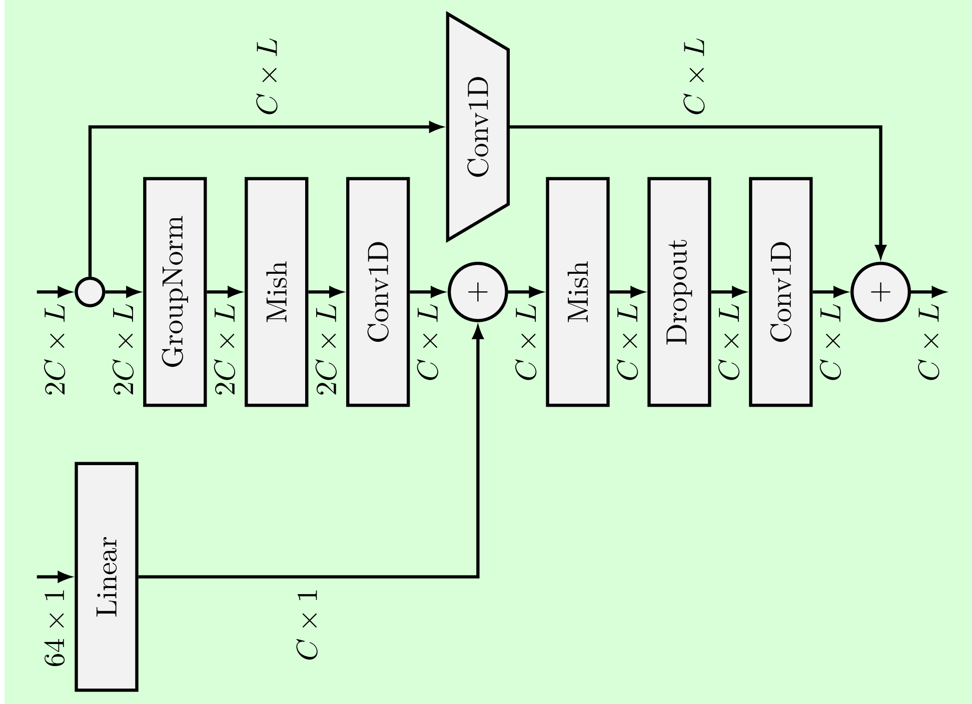


Figure 5.3: Diagram of the `ResnetBlock1D` block in the configuration used in this work. This block contains two convolutions, two activation layers, Group Normalization, optional Dropout, and a convolutional shortcut. Temporal embedding is transformed with a Linear layer and added between the convolutions.

in C dimensions is represented with a tensor of shape $C \times L$. The input of the network is the noisy path. The network processes the noisy data using 1D convolutions along the length axis. The network output is a tensor of the same shape $C \times L$, representing the noise prediction. The noise is subtracted from the noisy path following the Equation 3.12 and fed again into the network input. This process is repeated for T steps. The diffusion step is tracked as $T \geq t \geq 1$ and added as the second input into the network. We transform the step using the Positional Encoding inspired by Vaswani et al. [17] with 64 channels, as shown in the Figure 5.4. The embedded step is then fed into all ResNet blocks in the network as depicted in the Figure 5.3.

During training, we first sample a batch of paths from the training dataset and a random time step. Then, we add random noise to the path following the Equation 3.4. Finally, the network predicts the noise; we calculate the training loss according to the Equation 3.11 and take a gradient step. This way, we train the network to predict the noise distribution and serve as an approximation for the reverse diffusion step in the Equation 3.5.

During inference, the path is initialized with random Gaussian noise and then iteratively denoised by the network. After the last step, the output is returned as the denoised path. Since we trained the network to predict the noise distribution up to a near-normal distribution, the denoised path

Figure 5.4: The Positional Embedding block with embedding dimension 64. Half of the outputs use the cosine function, and the other half use the sine function. We chose the period of 10,000 as in the original Attention paper.

should be a sample from the approximated training distribution.

Chapter 6

Results

In this chapter, we experimentally demonstrate the capabilities of our implementation of the DDPM. It is important to consider that DDPM is both a novel approach in the field of robotics and a flexible model. The are no established benchmarks or standard datasets for this model. Also, our work is a proof of concept building the foundation for future research. There are many possible ways to extend and improve the model, we present only a few of them.

Images presented here were generated using the Algorithm 4 with different inpainting and steering functions as specified further.

6.1 Simple shapes

To demonstrate that the code1 is working correctly, we first tested the ability to learn and reproduce simple shapes in two and three dimensions. This demonstrates that both the training and inference processes are working correctly and the model hyperparameters are set up properly. Also, this already shows that our rewrite of the Diffusers code is successful, it was previously not possible to train the model on data with a different number of dimensions than the only dataset provided by Janner et al.

All models in this section were trained for 25 epochs with a batch size of 64, Adam optimizer [44] with a learning rate of 0.001, and a cosine learning rate schedule with 500 warm-up steps.

6.1.1 Circles dataset

First, we generated 3,000 circles with a uniformly sampled radius $r \in (0.25, 1)$ centered at the origin and all oriented in the same direction, see the Figure 6.1.

The first and the last points of the circle are the same and lie on the right semi-axis.

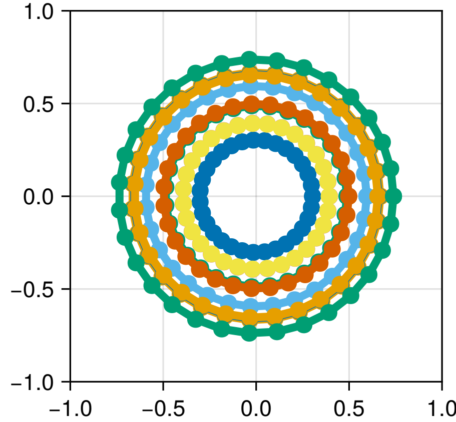


Figure 6.1: Ten samples from the Circles dataset.

To improve the results, we employed the following steering strategy.

Strategy 1: Length steering

For $20 > t \geq 0$ we calculate the pairwise squared distances between neighboring points of the path

$$d^t = \sum_{i=1}^{N-1} \left\| x_i^t - x_{i+1}^t \right\|^2 \tag{6.1}$$

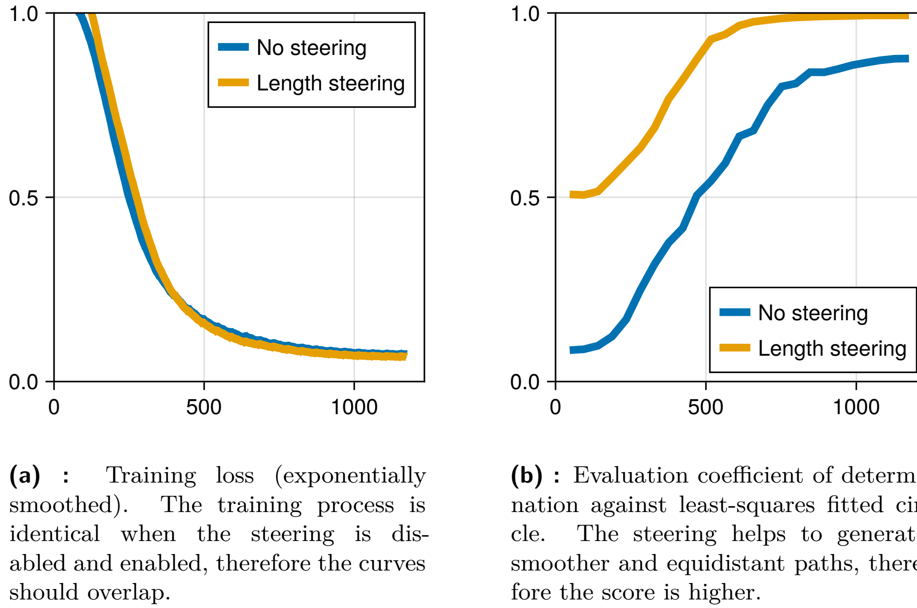
and make a gradient descent step

$$\hat{x}^t = x^t - \alpha \frac{\partial d^t}{\partial x^t} \tag{6.2}$$

after each denoising step. For our experiments, we selected $\alpha = 0.1$.

The training loss is shown in the Figure 6.2a. For evaluation, we fitted each output sample with a circle using least squares and calculated the R^2 coefficient of determination. The results are in the Figure 6.2b.

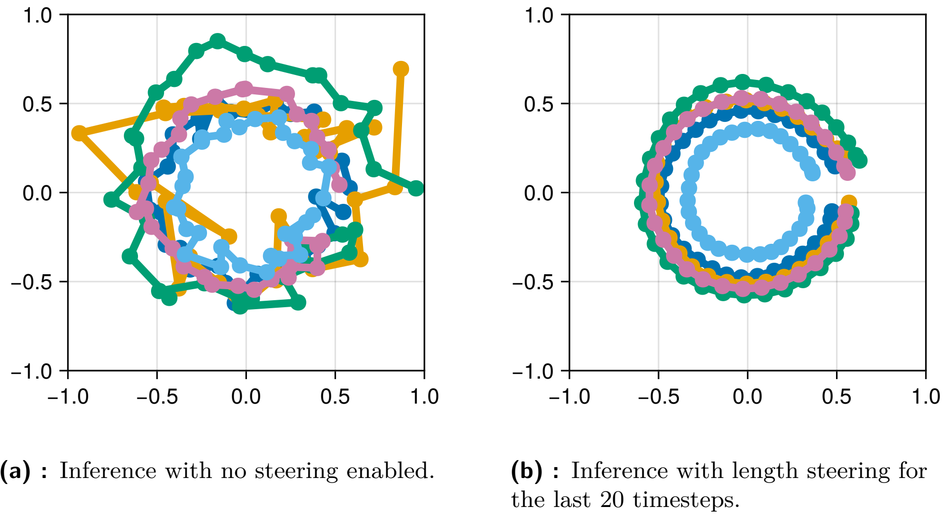
We can see that when the steering is enabled, the R^2 score is significantly higher. Note that in the Figure 6.3 the paths are smoother and more circular. However, the paths generated with steering (Figure 6.3b) are not closed. This is due to the steering strategy preferring shorter paths; there was no constraint on the distance between the first and last points. Also, the convolution is not cyclic, the first and last points are far apart in the autoencoder spatial dimension.



(a) : Training loss (exponentially smoothed). The training process is identical when the steering is disabled and enabled, therefore the curves should overlap.

(b) : Evaluation coefficient of determination against least-squares fitted circle. The steering helps to generate smoother and equidistant paths, therefore the score is higher.

Figure 6.2: Training metrics on the Circles dataset during the 25 epochs.



(a) : Inference with no steering enabled.

(b) : Inference with length steering for the last 20 timesteps.

Figure 6.3: Samples generated by the model trained on the Circles dataset.

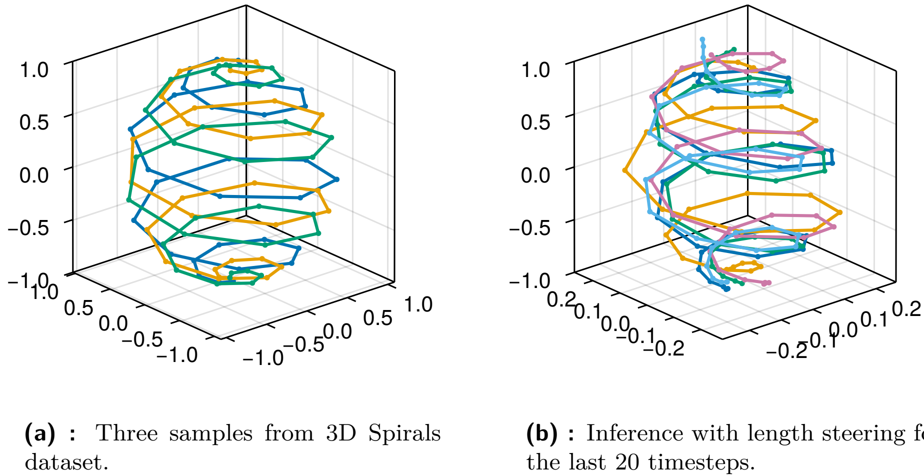
6.1.2 3D Spirals dataset

The second simple data set contains 3,000 spherical spirals in three dimensions with four revolutions. The spirals are generated by the parametric equations

$$\begin{aligned} x &= \sin(\theta)\cos(8\theta + \theta_0), \\ y &= \sin(\theta)\sin(8\theta + \theta_0), \\ z &= \cos(\theta), \end{aligned} \tag{6.3}$$

where $0 \leq \theta \leq \pi$ are 32 equally spaced points and $\theta_0 \in [0, 2\pi)$ is uniformly sampled starting angle. See the dataset in the Figure 6.4a.

For inference, we used the same steering Strategy 1 as for the Circles dataset. We can see some generated results in the Figure 6.4b.



(a) : Three samples from 3D Spirals dataset.

(b) : Inference with length steering for the last 20 timesteps.

Figure 6.4: Samples generated by the model trained on the 3D Spirals dataset.

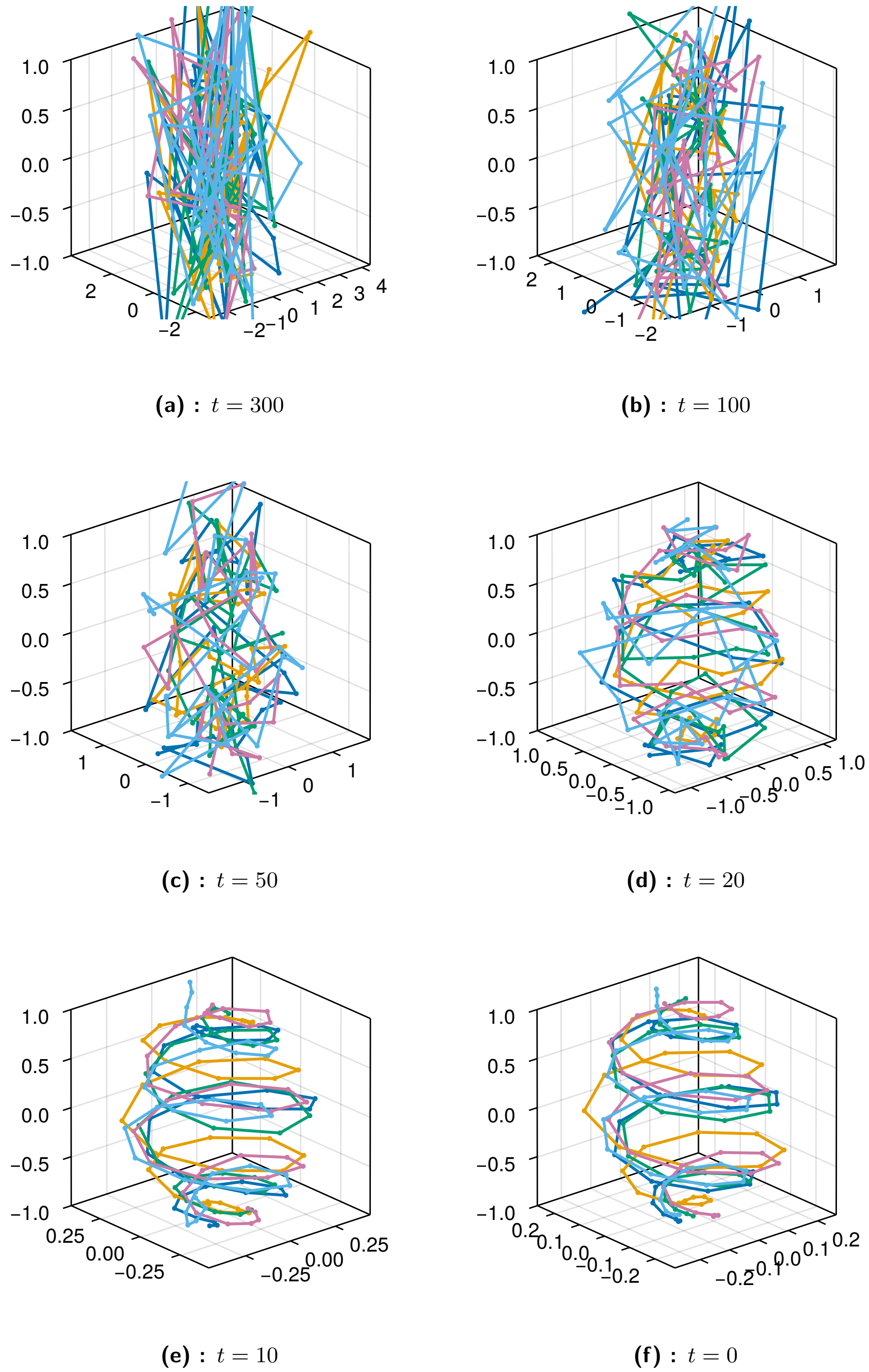
A visualization of the reverse diffusion process is in the Figure 6.5. The process starts with pure noise and iteratively refines the path until a relatively smooth spiral is generated.

We see that the model can generate paths that resemble the original shape and are smooth. We also see that without additional constraints, the model prefers to generate spirals with one particular orientation. If a diversity of outputs is required, a different steering function or an impainting strategy could be used.

6.2 Navigation problems

To test the model on more complex problems, we chose a navigation task in an environment with obstacles. Many algorithms can solve this task in different ways as mentioned earlier. We will compare them against our diffusion model.

We generate the maze using a contour of Perlin noise, a procedural noise generator developed by Ken Perlin [65]. We use the Python library called `perlin_noise` [66] developed by Ildar Salakhiev and licensed under the MIT license to achieve this. We generate the Perlin noise and threshold it at a certain level to create $\mathcal{S}_{\text{obstacle}}$, the threshold value determines the size of the obstacles.



(a) : $t = 300$

(b) : $t = 100$

(c) : $t = 50$

(d) : $t = 20$

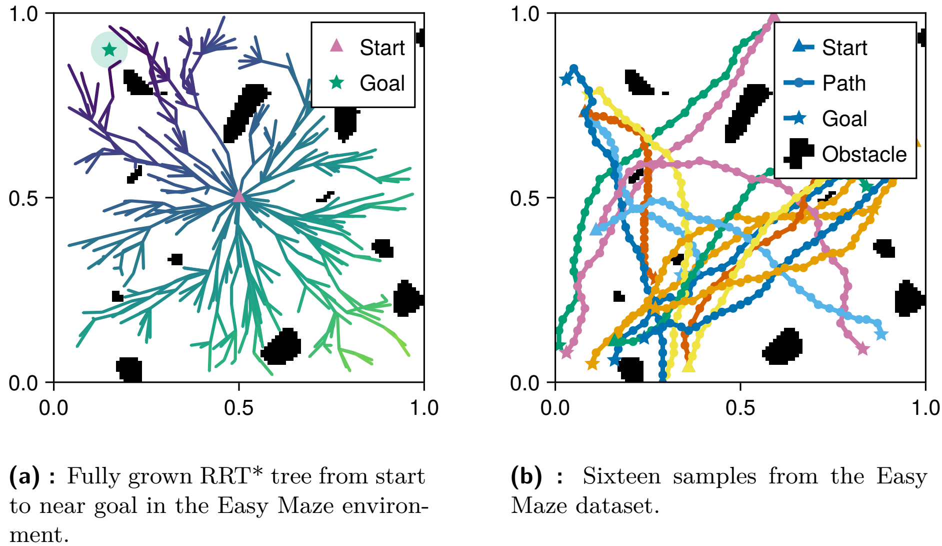
(e) : $t = 10$

(f) : $t = 0$

Figure 6.5: Visualization of several steps of the reverse diffusion process during the 3D Spirals inference. Length-based steering begins at $t = 19$

All models in this section were trained for 50 epochs with a batch size of 64, Adam optimizer with a learning rate of 0.001, and a cosine learning rate schedule with 500 warm-up steps.

To create a training dataset, 200 random start and goal configurations are uniformly sampled in the two-dimensional free space. The start and goal configurations are always at least half the maze size apart. For each start and goal pair, we run the RRT* algorithm [10] to generate a path for a point robot. As described in the Algorithm 1, the RRT* algorithm is a sampling-based motion planning algorithm that incrementally builds a search tree while also reconnecting the branches to make the paths final path shorter, see the Figure 6.6a. We resample the found path to a polygonal chain containing 32 equally spaced points. RRT* is restarted 50 times generating a total of 10,000 feasible paths. A sample from such a dataset is shown in the Figure 6.6b.



(a) : Fully grown RRT* tree from start to near goal in the Easy Maze environment.

(b) : Sixteen samples from the Easy Maze dataset.

Figure 6.6: Easy Maze dataset consisting of RRT* paths between random configurations in the free space.

Strategy 2: Obstacle steering

For $20 > t \geq 0$ we calculate a vector from each colliding point to the nearest free configuration and from each feasible point to the nearest colliding configuration

$$\Delta(x) = \begin{cases} -x + \left(\operatorname{argmin}_{p \in \mathcal{S}_{\text{free}}} \|p - x\|\right) & \text{if } x \in \mathcal{S}_{\text{obstacle}}, \\ x - \left(\operatorname{argmin}_{p \in \mathcal{S}_{\text{obstacle}}} \|p - x\|\right) & \text{if } x \in \mathcal{S}_{\text{free}} \end{cases} \tag{6.4}$$

using a breadth-first search on the Perlin noise grid.

We then take the total squared norm of these vectors

$$d^t = \sum_{i=1}^{N} \|\Delta(x)\|^2 \tag{6.5}$$

and make a gradient descent step

$$\hat{x}^t = x^t - \alpha \frac{\partial d^t}{\partial x^t} \tag{6.6}$$

after each denoising step. For our experiments, we selected $\alpha = 0.1$ using grid search.

For the navigation problems, we used two steering strategies, the Strategy 1 and the Strategy 2. We measured the time to generate a path for each steering strategy and batch size. The results are shown in the Figure 6.7. We see that generating a single path takes about 1.5 s on a single NVIDIA Quadro RTX 4000 while generating them in batches of 1,024 is a magnitude faster in terms of proportional time per path.

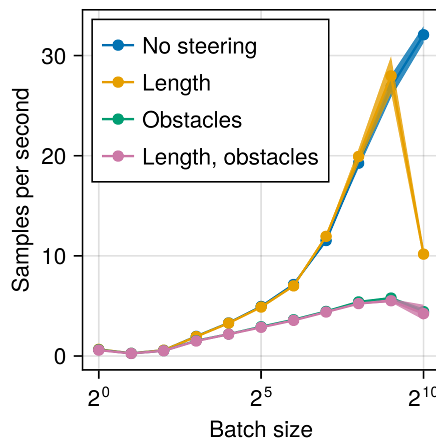


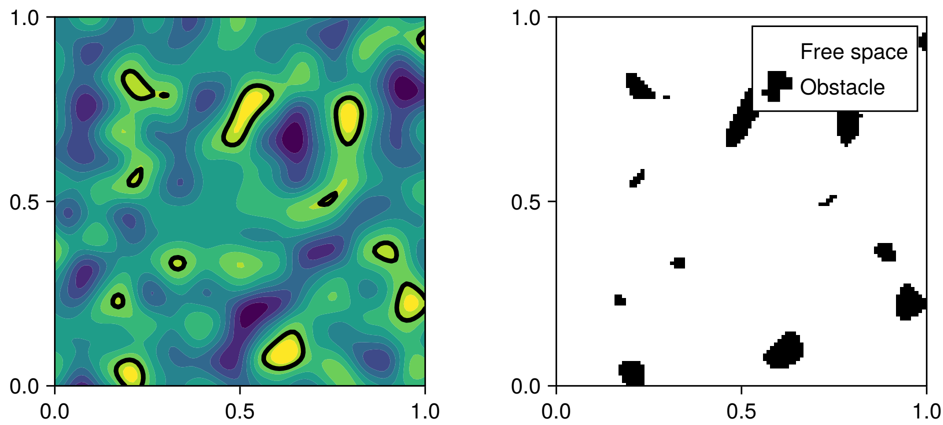
Figure 6.7: Speed scaling of the DDPM with the batch size. Total inference time divided by the batch size is reported. The average and standard deviation over 10 runs are shown for each steering strategy.

We also see that the performance degrades at higher batch sizes. CPU usage dominated GPU in most of our experiments. This is due to CPU and RAM saturation by surrounding code and collision checks. Much of the code can potentially be optimized, for example, collision checks could be rewritten in a compiled language with explicit memory management instead of Python.

6.2.1 Easy Maze dataset

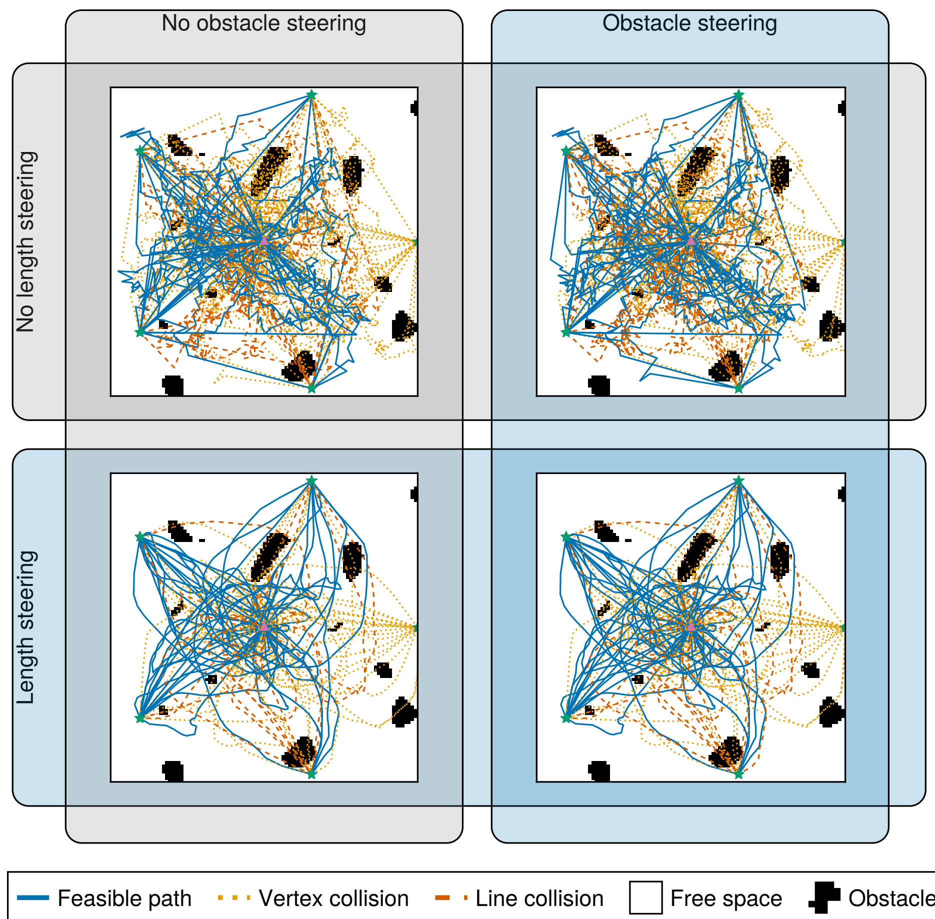
The Easy Maze is generated by thresholding the Perlin noise at 0.3. The Perlin noise is shown in the Figure 6.8a and the final maze in the Figure 6.8b.

See the comparison between the steering strategies for inpainting on 5 different goal positions in the Figure 6.8c. The model is successful in finding smooth and short paths for four of the five goals.



(a) : Contour plot of the computed Perlin noise with 7 octaves, contour on level 0.3 is highlighted in black.

(b) : The Easy Maze obstacle map generated by thresholding the Perlin noise at contour level 0.3.



(c) : Samples generated by the model in the Easy Maze environment.

Figure 6.8: Model trained on the Easy Maze dataset model performing in the Easy Maze environment.

We can see that the obstacle avoidance strategy generates very bumpy paths, but works well with the length strategy. A combination of both strategies can be used to generate smoother paths which are better than the paths generated by early-stopping the RRT* algorithm which were present in the training dataset.

6.2.2 Medium Maze dataset

To test the ability to generalize to more complex environments, we took the same model from the Easy Maze dataset and changed the evaluation environment. The Medium Maze environment is generated by thresholding the same Perlin noise curve as before at 0.2 therefore creating more obstacles. The Perlin noise is shown in the Figure 6.9a and the final maze in the Figure 6.9b.

As expected, the model is much less successful in the environment which is both more complex and different from the training dataset. The generated paths are shown in the Figure 6.9c.

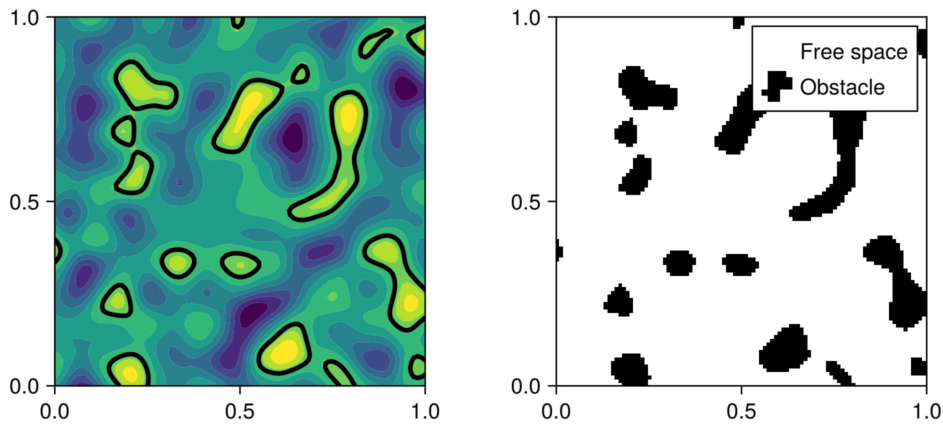
Out of the hundreds of generated paths, about 30 % do not collide with the obstacles in any vertex and only 20 % are feasible along the whole path. We see, that the model learned that some regions are present in a lot of training examples and generate paths forcing some of the points to go through these regions despite new obstacles. On the other hand, many paths follow the free space and are feasible even in the new environment.

6.2.3 Hard Maze dataset

We generated the Hard Maze environment by thresholding the same Perlin noise curve as before at 0.15. The Perlin noise is shown in the Figure 6.10a and the final maze in the Figure 6.10b. There is now a significant number of obstacles in the way and the task is challenging for all kinds of planners.

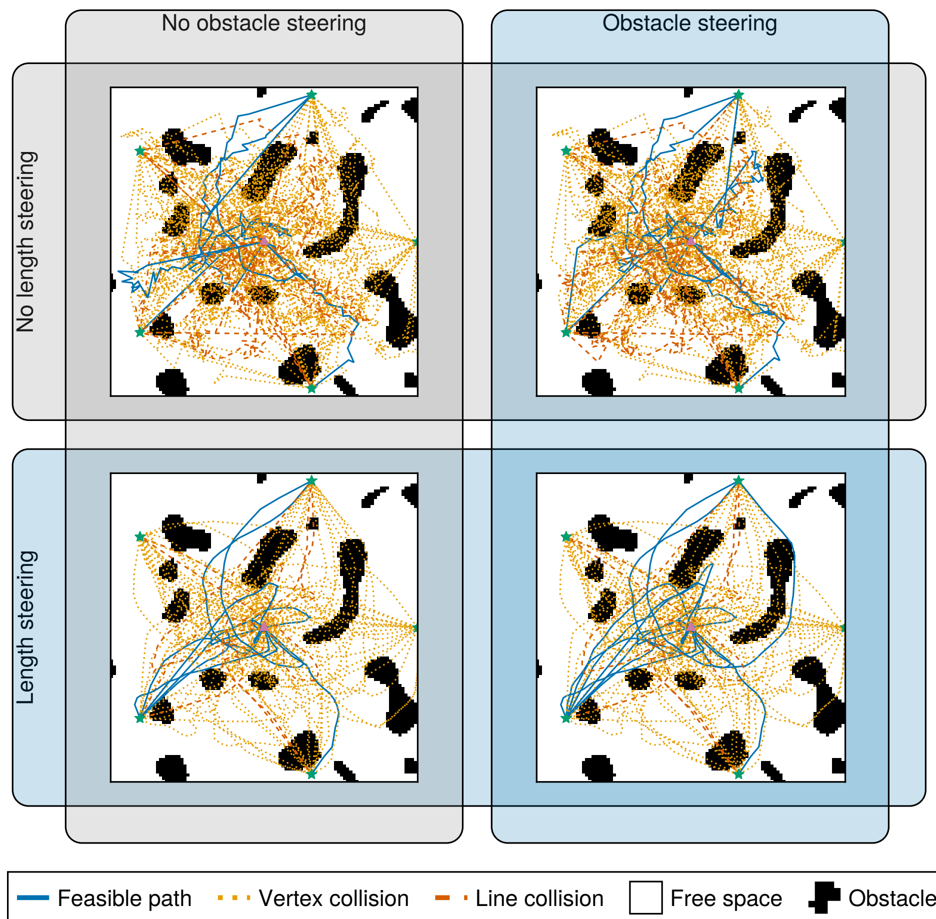
Still, our model generated 10 feasible paths to two of the five goals. Several other paths collide only partially with the obstacles and their parts can be easily fixed by another algorithm, for example as initialization for the RRT* algorithm.

For the top right goal in the Figure 6.10c, the model found three topologically distinct paths. This is a remarkable result as planning algorithms usually search either for the shortest path or a broad set of paths, but not for a diverse set of short paths. We saw this behavior appearing consistently in our experiments.



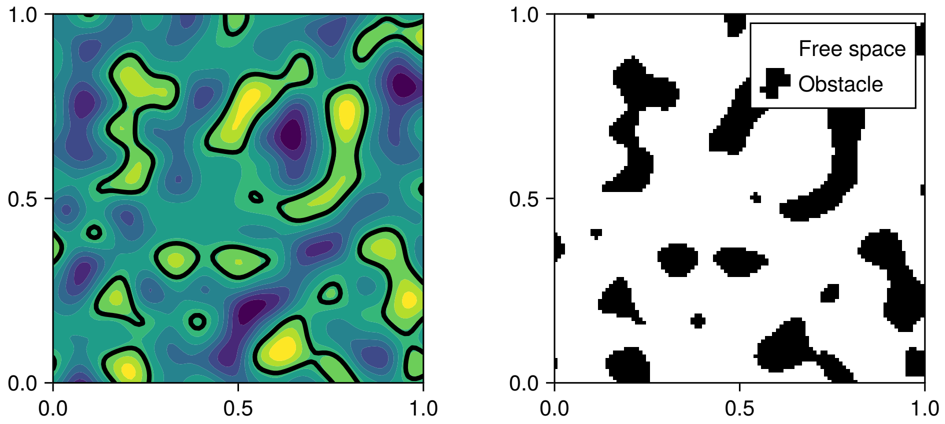
(a) : Contour plot of the computed Perlin noise with 7 octaves, contour on level 0.2 is highlighted in black.

(b) : The Medium Maze obstacle map generated by thresholding the Perlin noise at contour level 0.2.



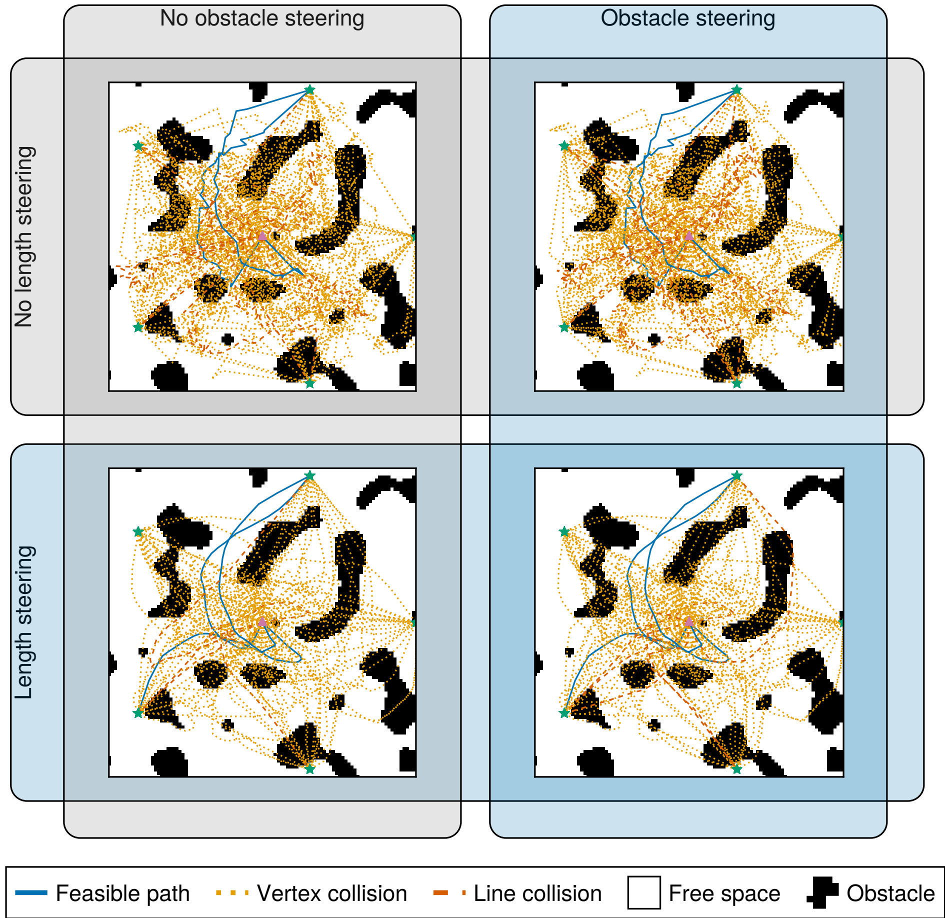
(c) : Samples generated by the model in the Medium Maze environment.

Figure 6.9: Model trained on the Easy Maze dataset model performing in the Medium Maze environment.



(a) : Contour plot of the computed Perlin noise with 7 octaves, contour on level 0.15 is highlighted in black.

(b) : The Hard Maze obstacle map generated by thresholding the Perlin noise at contour level 0.15.



(c) : Samples generated by the model in the Hard Maze environment.

Figure 6.10: Model trained on the Easy Maze dataset model performing in the Hard Maze environment.

6.3 Noise schedule

We learned the model with a linear 300-step noise schedule and generated all the above samples with the same schedule. It is, however, possible to use a different schedule, usually with fewer steps to reduce the inference time. For example, Sabour et al. [67] have shown that it is possible to generate high-quality images and video with only 10 to 20 diffusion steps.

We tested the model with several different schedule lengths to compare the results. We calculate the collisions by tracing the path using Bresenham's algorithm [68]. The path is colliding if at least one point on the path is colliding. You can see the average path collision rate in the Figure 6.11 and the average path length in the Figure 6.12. We see that skipping most of the steps does not significantly affect the collision rate, but increases the path length slightly. Depending on the exact implementation of the inference loop and steering strategy, reducing the number of steps from 300 to 20 can lead to 2x to 10x speedup.

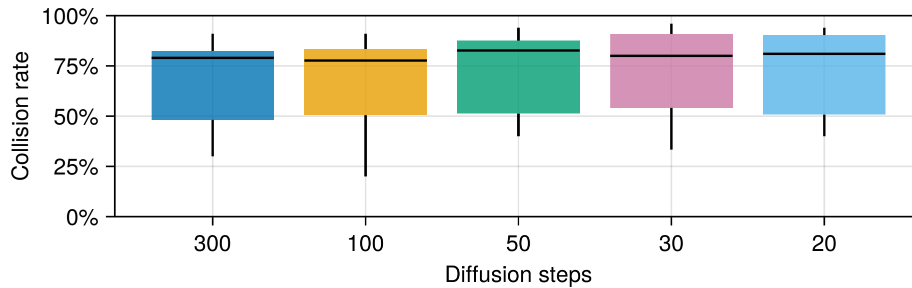


Figure 6.11: Average rate of colliding paths generated by various inference schedules. A linear schedule with different numbers of steps is used. The median and interquartile range over 12 different Maze environments are highlighted.

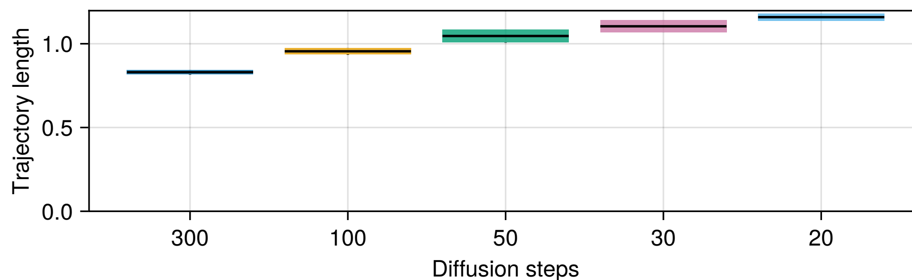


Figure 6.12: Average path length generated by various inference schedules. A linear schedule with different numbers of steps is used. The environment is a 1 by 1 square. The median and interquartile range over 12 different Maze environments are highlighted.

6.4 Comparison with other planners

To compare the performance of our model with other planners, we used the Open Motion Planning Library (OMPL) [69] which is a widely used motion planning library containing many planning algorithms. We compared the performance of the RRT, RRT-connect [70], and LBKPIECE1 [71, 72] planners with our model in the Easy, Medium, and Hard Maze environments. The results are shown in the Figure 6.13 and Figure 6.14. All the selected OMPL planners are sampling-based, which means they generate a path by sampling the configuration space and connecting the samples. Therefore their performance varies highly as we see in the figures.

The median runtime is two orders of magnitude higher for our model than for the OMPL planners. This is expected as the OMPL planners are highly optimized and our model is a proof of concept. However, it is a reasonable trade-off for the ability to generate diverse and smooth paths. Note that the runtime of the DDPM is almost constant with the Length steering strategy, it does not depend on the complexity of the environment. On the other hand, the Obstacle steering strategy slows down the model significantly in the Hard Maze environment.

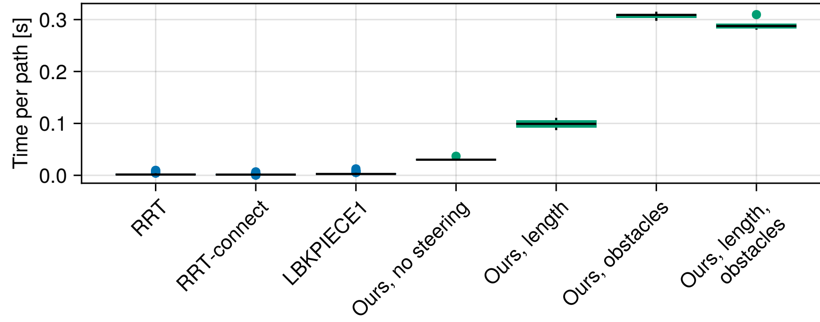
The number of collision checks is also constant (640), our model performs a single collision for each path vertex in each of the final 20 denoising steps. The OMPL planners perform a variable number of collision checks, depending on the random sampling of the configuration space. The number of collision checks is comparable between the OMPL planners and our model.

6.5 Summary

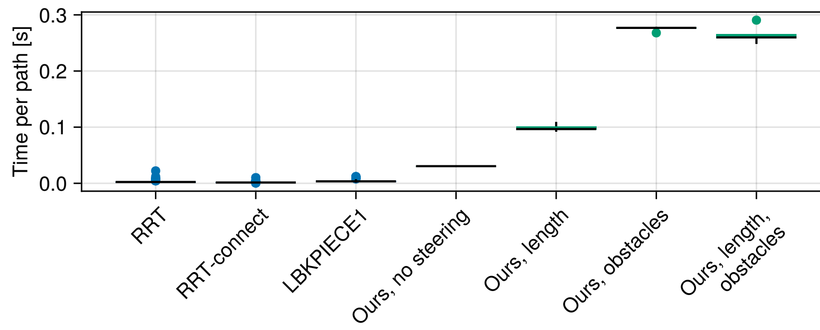
We have successfully demonstrated that the model can learn the distribution of various shapes and paths and generate new samples from it. We described two simple steering strategies that can be used to generate smooth and short paths, which is a unique feature of generative models. In combination with inpainting, it can be used to generate paths for a robot.

Despite learning in a simple environment, the model performs well in an altered and more complex environment, proving its ability to generalize and generate out-of-distribution samples.

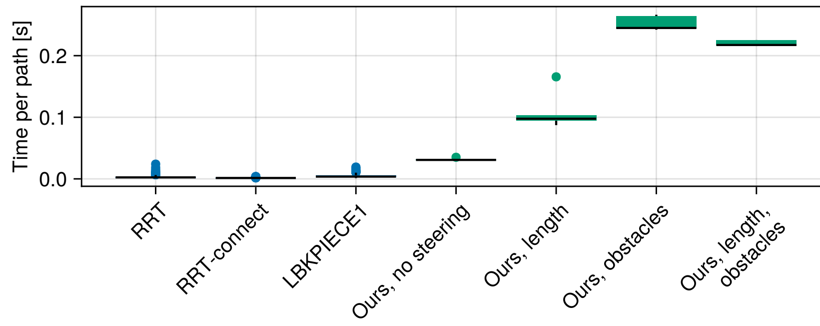
The model is slower than the state-of-the-art planners, but this is a reasonable trade-off for the ability to generate diverse and smooth paths. Also, the runtime is almost constant with the Length steering strategy, it does not depend on the complexity of the environment and can be improved by optimizing the code or using a different noise schedule.



(a) : Easy Maze environment.

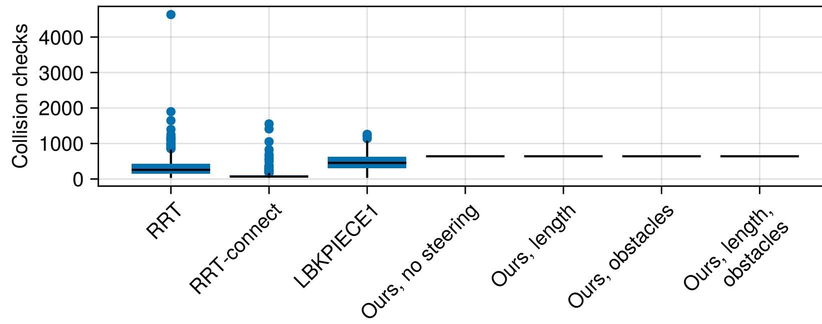


(b) : Medium Maze environment.

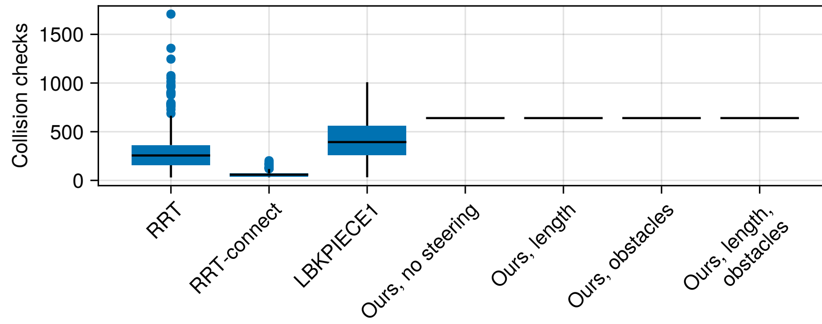


(c) : Hard Maze environment.

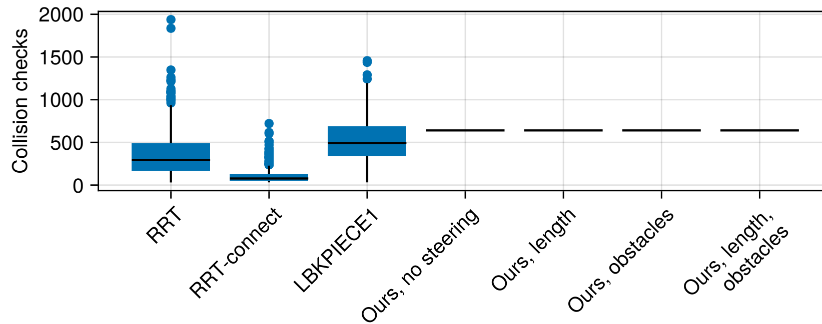
Figure 6.13: Comparison of various OMPL planners with our DDPM in different Maze environments. Time to find a feasible path is reported for OMPL planners. Inference time per one sample from the batch of 1,024 is reported for the DDPM.



(a) : Easy Maze environment.



(b) : Medium Maze environment.



(c) : Hard Maze environment.

Figure 6.14: Comparison of various OMPL planners with our DDPM in different Maze environments. Path collision rate reported.

Chapter 7

Conclusion

7.1 Conclusion

In this work, we have presented a recent approach to the path planning problem using Denoising Diffusion Probabilistic Model (DDPM). Denoising diffusion neural networks are currently the state of the art in many domains including image and video generation.

We explained how to create a DDPM, how to train it and how to sample new data from it. Our main concern was to build upon the existing work and apply it to the path planning problem, the approach used only in a few papers so far. We reviewed the existing work and open-source implementation of DDPMs for path planning and we identified their limitations.

Because we were unable to directly use any of the existing implementations to reproduce the results, we decided to fix one of them. We took the Hugging Face Diffusers library and rewrote large parts of it. Apart from fixing the implementation bugs, we also added a great number of features and improvements. In our version, the user can configure Group Normalization, Dropout, activation function, time embedding, Attention, skip connections and other features. We achieved parity with Diffusers' two-dimensional DDPMs which are one of the most used implementations.

After reimplementing the DDPM library, we designed a neural network architecture for path planning. We demonstrated that the model can learn from a dataset of paths and generate new paths in previously unseen environments.

We also showed the flexibility of using the model together with a steering strategy, for example smoothing the path or guiding it outside of obstacles. We discussed that steering is a unique feature of DDPMs and that it can be used to greatly improve the quality of the generated paths.

7.2 Future work

Our works serves mainly as a learning material and a proof of concept for using DDPM in path planning and can be extended in many ways.

We generated the training data by early stopping a path-planning algorithm between random pairs of points. to improve the quality of the generated paths, we could generate a more specific dataset, for example using Dubins paths or robotic manipulator paths. For example, Liang et al. [73] propose to generate additional data based on the generated paths.

For training the algorithm, we used a simple training loop with a cosine learning schedule and Adam optimizer. The training procedure could be improved by choosing different hyperparameters, using a different optimizer or a different learning rate schedule based on recent research.

We used a relatively shallow neural network architecture. We could experiment with more layers and different layer types. We did not observe overfitting in our experiments, but we did not perform a proper analysis. Also, we did not perform an ablation study of the architecture.

We saw that the steering strategy has a great impact on the quality of the generated paths. We see that using a fast but accurate steering strategy is crucial to achieve good results. We think that combining DDPM with a suitable steering strategy, for example, a local sampling-based planner, could be the key to achieving state-of-the-art results and outperforming existing path-planning algorithms.

We see DDPM as a great addition to the path-planning toolbox, mainly as a supporting method to use together with other path-planning algorithms.

Apart from using an existing path planning algorithm as a steering strategy, we can generate a set of partially infeasible paths using the DDPM and initialize the path planning algorithm with them. This is straightforward with sampling-based algorithms such as RRT or PRM. Also, sampling-based algorithms can sample from a non-uniform distribution, biased towards the DDPM-generated paths.

These are just a few ideas for future work. We are excited to see how the field of path planning will evolve with the use of DDPMs in the same way DDPM revolutionized image and video generation.

List of attachements

- **images.zip** A collection of visualization images from this work in vector format.
- **sourcecode.zip** Source code of the improved Diffusers library.

Bibliography

[1] K. M. Hasan, Abdullah-Al-Nahid, and K. J. Reza. “Path planning algorithm development for autonomous vacuum cleaner robots.” In: 2014 International Conference on Informatics, Electronics & Vision (ICIEV). 2014-05, pp. 1–6. DOI: 10.1109/ICIEV.2014.6850799.

[2] Z. Li et al. “A Fault-Tolerant Method for Motion Planning of Industrial Redundant Manipulator.” In: *IEEE Transactions on Industrial Informatics* 16.12 (2020-12), pp. 7469–7478. ISSN: 1941-0050. DOI: 10.1109/TII.2019.2957186.

[3] Y. Zhou, B. Rao, and W. Wang. “UAV Swarm Intelligence: Recent Advances and Future Trends.” In: *IEEE Access* 8 (2020), pp. 183856–183878. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3028865.

[4] J. H. Reif. “Complexity of the mover’s problem and generalizations.” In: 20th Annual Symposium on Foundations of Computer Science (sfcs 1979). ISSN: 0272-5428. 1979-10, pp. 421–427. DOI: 10.1109/SFCS.1979.10.

[5] S. LaValle. *Rapidly-exploring random trees: A new tool for path planning*. 9811. Department of Computer Science: Iowa State University, 1998. URL: https://msl.cs.illinois.edu/~lavalle/papers/Lav98c.pdf (visited on 03/11/2024).

[6] J. Sohl-Dickstein et al. *Deep Unsupervised Learning using Nonequilibrium Thermodynamics*. 2015-11-18. DOI: 10.48550/arXiv.1503.03585.

[7] P. E. Hart, N. J. Nilsson, and B. Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths.” In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968-07), pp. 100–107. ISSN: 2168-2887. DOI: 10.1109/TSSC.1968.300136.

[8] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006-05-29. 1362 pp. ISBN: 978-1-139-45517-6.

[9] L. Kavraki et al. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces." In: *IEEE Transactions on Robotics and Automation* 12.4 (1996-08), pp. 566–580. ISSN: 2374-958X. DOI: 10.1109/70.508439.

[10] S. Karaman and E. Frazzoli. *Sampling-based Algorithms for Optimal Motion Planning*. 2011-05-05. DOI: 10.48550/arXiv.1105.1186.

[11] Rodriguez et al. "An obstacle-based rapidly-exploring random tree." In: *Proceedings 2006 IEEE International Conference on Robotics and Automation*. ICRA 2006. ISSN: 1050-4729. 2006-05, pp. 895–900. DOI: 10.1109/ROBOT.2006.1641823.

[12] M. Elbanhawi and M. Simic. "Sampling-Based Robot Motion Planning: A Review." In: *IEEE Access* 2 (2014), pp. 56–77. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2014.2302442.

[13] D. Manocha and J. Canny. "Efficient inverse kinematics for general 6R manipulators." In: *IEEE Transactions on Robotics and Automation* 10.5 (1994-10), pp. 648–657. ISSN: 2374-958X. DOI: 10.1109/70.326569.

[14] Z. Yao and K. Gupta. "Path planning with general end-effector constraints: using task space to guide configuration space search." In: *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems. 2005-08, pp. 1875–1880. DOI: 10.1109/IROS.2005.1545305.

[15] L. Jaillet and J. M. Porta. "Path Planning with Loop Closure Constraints Using an Atlas-Based RRT." In: *Robotics Research : The 15th International Symposium ISRR*. Ed. by H. I. Christensen and O. Khatib. Springer Tracts in Advanced Robotics. Cham: Springer International Publishing, 2017, pp. 345–362. ISBN: 978-3-319-29363-9. DOI: 10.1007/978-3-319-29363-9_20.

[16] L. G. D. O. Veras, F. L. L. Medeiros, and L. N. F. Guimaraes. "Systematic Literature Review of Sampling Process in Rapidly-Exploring Random Trees." In: *IEEE Access* 7 (2019), pp. 50933–50953. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2908100.

[17] A. Vaswani et al. *Attention Is All You Need*. 2023-08-01. DOI: 10.48550/arXiv.1706.03762.

[18] T. B. Brown et al. *Language Models are Few-Shot Learners*. 2020-07-22. DOI: 10.48550/arXiv.2005.14165.

[19] M. Chen et al. *Evaluating Large Language Models Trained on Code*. 2021-07-14. DOI: 10.48550/arXiv.2107.03374.

[20] M. Oquab et al. *DINOv2: Learning Robust Visual Features without Supervision*. 2024-02-02. DOI: 10.48550/arXiv.2304.07193.

[21] J. Jumper et al. “Highly accurate protein structure prediction with AlphaFold.” In: *Nature* 596.7873 (2021-08), pp. 583–589. ISSN: 1476-4687. DOI: 10.1038/s41586-021-03819-2.

[22] A. Radford et al. *Learning Transferable Visual Models From Natural Language Supervision*. 2021-02-26. DOI: 10.48550/arXiv.2103.00020.

[23] I. J. Goodfellow et al. *Generative Adversarial Networks*. 2014-06-10. DOI: 10.48550/arXiv.1406.2661.

[24] A. Ramesh et al. *Zero-Shot Text-to-Image Generation*. 2021-02-26. DOI: 10.48550/arXiv.2102.12092.

[25] A. Ramesh et al. *Hierarchical Text-Conditional Image Generation with CLIP Latents*. 2022-04-12. URL: http://arxiv.org/abs/2204.06125 (visited on 01/13/2024).

[26] R. Rombach et al. *High-Resolution Image Synthesis with Latent Diffusion Models*. 2022-04-13. DOI: 10.48550/arXiv.2112.10752.

[27] Z. Kong et al. *DiffWave: A Versatile Diffusion Model for Audio Synthesis*. 2021-03-30. DOI: 10.48550/arXiv.2009.09761.

[28] A. Blattmann et al. *Stable Video Diffusion: Scaling Latent Video Diffusion Models to Large Datasets*. 2023-11-25. DOI: 10.48550/arXiv.2311.15127.

[29] T. Brooks et al. “Video generation models as world simulators.” In: (2024). URL: https://openai.com/research/video-generation-models-as-world-simulators (visited on 03/11/2024).

[30] OpenAI et al. *Learning Dexterous In-Hand Manipulation*. 2019-01-18. DOI: 10.48550/arXiv.1808.00177.

[31] L. Kaiser et al. *Model-Based Reinforcement Learning for Atari*. 2020-02-19. DOI: 10.48550/arXiv.1903.00374.

[32] OpenAI et al. *Dota 2 with Large Scale Deep Reinforcement Learning*. 2019-12-13. DOI: 10.48550/arXiv.1912.06680.

[33] Z. Wang, J. J. Hunt, and M. Zhou. *Diffusion Policies as an Expressive Policy Class for Offline Reinforcement Learning*. 2023-08-25. DOI: 10.48550/arXiv.2208.06193.

[34] L. Lehnert et al. *Beyond A*: Better Planning with Transformers via Search Dynamics Bootstrapping*. 2024-02-21. URL: http://arxiv.org/abs/2402.14083 (visited on 02/25/2024).

[35] J. J. Johnson, A. H. Qureshi, and M. Yip. *Learning Sampling Dictionaries for Efficient and Generalizable Robot Motion Planning with Transformers*. 2023-09-26. DOI: 10.48550/arXiv.2306.00851.

[36] A. v. d. Oord, O. Vinyals, and K. Kavukcuoglu. *Neural Discrete Representation Learning*. 2018-05-30. URL: http://arxiv.org/abs/1711.00937 (visited on 03/11/2024).

[37] M. Janner, Q. Li, and S. Levine. *Offline Reinforcement Learning as One Big Sequence Modeling Problem*. 2021-11-28. DOI: 10.48550/arXiv.2106.02039.

[38] A. Nichol and P. Dhariwal. *Improved Denoising Diffusion Probabilistic Models*. 2021-02-18. DOI: 10.48550/arXiv.2102.09672.

[39] J. Ho, A. Jain, and P. Abbeel. *Denoising Diffusion Probabilistic Models*. 2020-12-16. DOI: 10.48550/arXiv.2006.11239.

[40] D. P. Kingma and M. Welling. *Auto-Encoding Variational Bayes*. 2013-12-20. DOI: 10.48550/arXiv.1312.6114.

[41] A. Shekhovtsov, D. Schlesinger, and B. Flach. *VAE Approximation Error: ELBO and Exponential Families*. 2022-04-11. DOI: 10.48550/arXiv.2102.09310.

[42] D. P. Kingma et al. *Improving Variational Inference with Inverse Autoregressive Flow*. 2017-01-30. DOI: 10.48550/arXiv.1606.04934.

[43] T. Salimans et al. *PixelCNN++: Improving the PixelCNN with Discretized Logistic Mixture Likelihood and Other Modifications*. 2017-01-19. DOI: 10.48550/arXiv.1701.05517.

[44] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. 2017-01-29. DOI: 10.48550/arXiv.1412.6980.

[45] D. Pathak et al. *Context Encoders: Feature Learning by Inpainting*. 2016-11-21. DOI: 10.48550/arXiv.1604.07379.

[46] A. Lugmayr et al. *RePaint: Inpainting using Denoising Diffusion Probabilistic Models*. 2022-08-31. DOI: 10.48550/arXiv.2201.09865.

[47] M. Janner et al. *Planning with Diffusion for Flexible Behavior Synthesis*. 2022-12-20. DOI: 10.48550/arXiv.2205.09991.

[48] J. Choi et al. *ILVR: Conditioning Method for Denoising Diffusion Probabilistic Models*. 2021-09-15. DOI: 10.48550/arXiv.2108.02938.

[49] J. Carvalho et al. *Motion Planning Diffusion: Learning and Planning of Robot Motions with Diffusion Models*. 2023-08-03. DOI: 10.48550/arXiv.2308.01557.

[50] O. Ronneberger, P. Fischer, and T. Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015-05-18. URL: http://arxiv.org/abs/1505.04597 (visited on 01/14/2024).

[51] D. Misra. *Mish: A Self Regularized Non-Monotonic Activation Function*. 2020-08-13. DOI: 10.48550/arXiv.1908.08681.

[52] K. He et al. *Deep Residual Learning for Image Recognition*. 2015-12-10. DOI: 10.48550/arXiv.1512.03385.

[53] Y. Wu and K. He. *Group Normalization*. 2018-06-11. DOI: 10.48550/arXiv.1803.08494.

[54] E. Todorov, T. Erez, and Y. Tassa. "MuJoCo: A physics engine for model-based control." In: 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2012). IEEE, 2012-10, pp. 5026–5033. ISBN: 978-1-4673-1736-8. DOI: 10.1109/IROS.2012.6386109.

[55] G. Brockman et al. *OpenAI Gym*. 2016-06-05. DOI: 10.48550/arXiv.1606.01540.

[56] M. Towers et al. *Gymnasium*. 2024-03-25. DOI: 10.5281/zenodo.8127026.

[57] *Farama-Foundation/D4RL*. 2024-03-25. URL: https://github.com/Farama-Foundation/D4RL (visited on 03/25/2024).

[58] Gymnasium-Robotics Contributors. *Gymnasium-Robotics: A a collection of robotics simulation environments for Reinforcement Learning*. 2022-01. URL: https://github.com/Farama-Foundation/Gymnasium-Robotics (visited on 03/25/2024).

[59] Minari Contributors. *Minari: A dataset API for Offline Reinforcement Learning*. 2023-05. URL: https://github.com/Farama-Foundation/Minari (visited on 03/25/2024).

[60] P. v. Platen et al. *Diffusers: State-of-the-art diffusion models*. Version 0.12.1. 2024-01-14. URL: https://github.com/huggingface/diffusers (visited on 01/14/2024).

[61] E. J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021-10-16. DOI: 10.48550/arXiv.2106.09685.

[62] *huggingface/peft*. 2024-03-25. URL: https://github.com/huggingface/peft (visited on 03/25/2024).

[63] D. Bahdanau, K. Cho, and Y. Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016-05-19. DOI: 10.48550/arXiv.1409.0473.

[64] O. Oktay et al. *Attention U-Net: Learning Where to Look for the Pancreas*. 2018-05-20. DOI: 10.48550/arXiv.1804.03999.

[65] K. Perlin. "An image synthesizer." In: *ACM SIGGRAPH Computer Graphics* 19.3 (1985), pp. 287–296. ISSN: 0097-8930. DOI: 10.1145/325165.325247.

[66] I. SALAKHIEV. *salaxieb/perlin_noise*. original-date: 2020-10-01T15:36:24Z. 2024-04-19. URL: https://github.com/salaxieb/perlin_noise (visited on 05/05/2024).

[67] A. Sabour, S. Fidler, and K. Kreis. *Align Your Steps: Optimizing Sampling Schedules in Diffusion Models*. 2024-04-22. DOI: 10.48550/arXiv.2404.14507.

[68] J. E. Bresenham. "Algorithm for computer control of a digital plotter." In: *IBM Systems Journal* 4.1 (1965), pp. 25–30. ISSN: 0018-8670. DOI: 10.1147/sj.41.0025.

[69] I. A. Sucan, M. Moll, and L. E. Kavraki. "The Open Motion Planning Library." In: *IEEE Robotics & Automation Magazine* 19.4 (2012-12), pp. 72–82. ISSN: 1558-223X. DOI: 10.1109/MRA.2012.2205651.

[70] J. Kuffner and S. LaValle. "RRT-connect: An efficient approach to single-query path planning." In: *Proceedings 2000 ICRA*. IEEE International Conference on Robotics and Automation. Vol. 2. ISSN: 1050-4729. 2000-04, 995–1001 vol.2. DOI: 10.1109/ROBOT.2000.844730.

[71] I. A. Şucan and L. E. Kavraki. "Kinodynamic Motion Planning by Interior-Exterior Cell Exploration." In: *Algorithmic Foundation of Robotics VIII*. Ed. by G. S. Chirikjian et al. Red. by B. Siciliano, O. Khatib, and F. Groen. Vol. 57. Series Title: Springer Tracts in Advanced Robotics. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 449–464. ISBN: 978-3-642-00311-0 978-3-642-00312-7. DOI: 10.1007/978-3-642-00312-7_28.

[72] R. Bohlin and L. Kavraki. "Path planning using lazy PRM." In: *Proceedings 2000 ICRA*. IEEE International Conference on Robotics and Automation. Vol. 1. ISSN: 1050-4729. 2000-04, 521–528 vol.1. DOI: 10.1109/ROBOT.2000.844107.

[73] Z. Liang et al. *AdaptDiffuser: Diffusion Models as Adaptive Self-evolving Planners*. 2023-05-12. DOI: 10.48550/arXiv.2302.01877.