

Bachelor's Thesis in Robotics, Cognition, Intelligence

Evolving Soft Robot Morphologies with Generative Diffusion Models

Entwicklung von Morphologien weicher Roboter
mit Diffusionsmodellen

Supervisor	Prof. Dr.-Ing. habil. Alois C. Knoll
Advisor	Erdi Sayar, M.Sc.
Author	John Stewardson
Date	October 15, 2024 in Munich

Disclaimer

I confirm that this Bachelor's Thesis is my own work and I have documented all sources and material used.

Munich, October 15, 2024



(John Stewardson)

Abstract

Co-design of soft robots focuses both on controlling the robot, and designing the mechanical structure of the robot itself. While the former is well-researched in the robotics community, there is much to explore in the generative design of robots.

In this thesis, a new design optimization method is introduced, using diffusion models to learn the underlying reward distribution of the design space. The generational diffusion algorithm divides the task of learning the entire reward distribution into learning only part of it every generation, while moving closer to the global maximum in the process.

This method is evaluated in the benchmark environment Walker-v0 of Evogym [Bha+21] and compared to the genetic algorithm and CPPN-NEAT, two of the most prevalent design algorithms. Finally, the method is positioned in the context of design optimization, giving an outlook on what can be improved and researched further.

The corresponding code implementation in Python can be accessed on GitHub:
https://github.com/JohnStewardson/evolving_robots_diffusion

Kurzfassung

Während in der Robotik-Forschung bereits viel Aufmerksamkeit auf die intelligente Regelung von Robotern gerichtet ist, ist die generative Gestaltung der Struktur noch weitgehend unerforscht. In dieser Arbeit wird eine neue Methode zur Gestaltung von Robotern mittels Diffusionsmodellen beschrieben. Das Lernen der gesamten Verteilung der Belohnungsfunktion über den möglichen Roboterkonfigurationen wird hierbei auf mehrere Generationen aufgeteilt. Jede Generation lernt einen Teil des Gestaltungsraums, mit Robotern, dessen Belohnungen höher und näher am globalen Maximum liegen als die der vorherigen Generation. Anschließend wird die Methode in der Simulationsumgebung Walker-v0 von Evogym [Bha+21] evaluiert und mit dem etablierten genetischen Algorithmus und CPPN-NEAT Algorithmus verglichen. Der Schluss der Arbeit gibt einen Ausblick darüber, wie die Methode zukünftig verbessert werden kann.

Die Implementierung des Codes in Python ist auf GitHub hinterlegt:
https://github.com/JohnStewardson/evolving_robots_diffusion

Contents

1	Automated Design Revolutionizing Robotics	1
2	Background	3
2.1	Diffusion Models	3
2.2	Evogym	6
2.2.1	Robot Representation	6
2.2.2	Task Representation	6
2.2.3	Structure of Co-Design Algorithms	7
2.3	Reinforcement Learning	8
2.3.1	Markov Decision Process	8
2.3.2	Model-Free Reinforcement Learning	9
2.3.3	Policy Gradient Methods	10
2.3.4	Trust Region Policy Optimization	11
2.3.5	Proximal Policy Optimization	11
2.3.6	Algorithm Implementation	12
3	Related Work	15
3.1	Genetic Algorithm	15
3.2	CPPN-NEAT	16
3.2.1	NeuroEvolution of Augmenting Topologies (NEAT)	16
3.2.2	Compositional Pattern Producing Networks (CPPNs)	19
3.2.3	Combining CPPN and NEAT	20
4	Designing Robot Structures with Diffusion Models	21
4.1	Concept: Generational Diffusion Algorithm	21
4.2	Forward Process	23
4.3	Model Architecture	25
4.3.1	Down-Sampling	26
4.3.2	Up-Sampling	26
4.3.3	Self Attention	26
4.3.4	Double Convolution	27
4.4	Hyperparameter Tuning	27
4.5	Evaluating Diversity of Generated Robots	29
5	Experiments	33
5.1	Walker-v0	33
5.2	Simplified Reward Environment	34
6	Positioning Diffusion Models in Design Optimization	39
6.1	Comparison to GA and CPPN-NEAT	39
6.2	Weaknesses of Diffusion Models	39

6.3	Possible Improvements for Future Work	40
6.4	Advantages of Diffusion Models	41
6.5	Conclusion and Outlook	42
A	Appendix 1	43
	Bibliography	47

Chapter 1

Automated Design Revolutionizing Robotics

Artificial Intelligence (AI) and Machine Learning (ML) have advanced countless industries, replacing or alleviating manual labor and solving previously unsolvable tasks. In robotics, reinforcement learning is commonly used to solve complex control problems that are not well-defined.[Kim+21]

Traditionally, the design of robots has been a labor-intensive process, heavily reliant on human effort. However, AI is now being utilized to automate the entire design process, from initial concept to prototype, significantly reducing the time and effort required from human designers. This exciting development is set to revolutionize the field of robotics. [Smi+22] [Lip14]

Automated design, when coupled with simulations, significantly reduces the number of physical iterations required and accelerates the development process. Furthermore, robots designed by AI are not bound by human imagination, leading to the creation of innovative and high-performing machines. These advantages translate into a more cost-effective and superior product, developed in a shorter time frame.

State-of-the-art design algorithms rely on evolutionary principles, for example, survival of the fittest, where the ideal design is found through random mutations over multiple generations, only allowing the best robots of every generation to reproduce[SM02][KCK21]. However, these algorithms still perform poorly in some tasks, warranting further research and discovery of new methods.[Bha+21]

A novel approach utilizes diffusion models, known for their ability to produce realistic images and videos, to generate robot structures. While diffusion models work in pixel space to create images, the prevailing method for soft robot generation, is to model the robot as a combination of many cells. This concept is inspired by genetics, where living organisms are composed of numerous cells, and the "design problem" reduces to determining what type of cell each cell should be (for example muscle, bone, or skin cell). In this new method, the diffusion model works with cells instead of pixels and cell types instead of pixel values. The robot learns to move, i.e., to control the actuators (the muscles of robots) with a separate reinforcement learning algorithm. By embedding the task of learning the movement in the task of learning the robot structure, a holistic generation of a functioning robot is achieved. To understand how this works, we first need to understand the working principle behind diffusion models.[HJA20]

Chapter 2

Background

2.1 Diffusion Models

Diffusion models are a deep, unsupervised learning method, developed by Jonathan Ho, Ajay Jain, and Pieter Abbeel at the University of California, Berkeley. The deep generative model became widely known for its use in realistic image generation, as the sampled images are more diverse, i.e., better capture the data distribution, compared to Generative Adversarial Networks (GANs), which was the established method before the introduction of diffusion models.[HJA20]

There are many valid interpretations of diffusion models. The most intuitive, is that during training, noise is added to the training data, and the model predicts the added noise (or equivalently learns the original input). Then, gradient descent is used to optimize this noise prediction. Once the model can predict what noise was added to the input, it can remove the noise to obtain the original. Moreover, given complete random noise as input, the trained model can denoise it to generate an output that resembles the training data. This denoising is split up into many steps, making the training easier. This is illustrated in Figure 2.1. [HJA20]

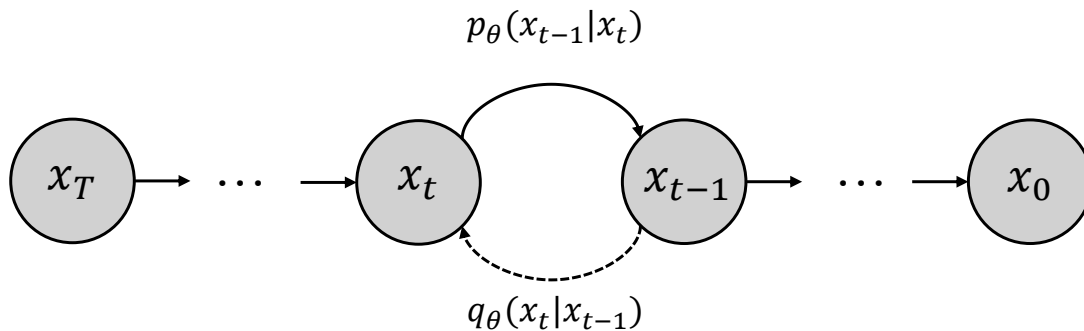


Figure 2.1: Denoising schematic [HJA20]

In the forward diffusion process, Gaussian noise is added to x_{t-1} to obtain x_t . x_T denotes the fully deconstructed input (that is now Gaussian noise), and x_0 is the original input. One forward step (adding noise) to x_{t-1} leads to a probability distribution of x_t , as the noise is sampled from a Gaussian distribution, and is not deterministic. The probability distribution of x_t given x_{t-1} is given by [HJA20]:

$$q(x_t | x_{t-1}) = \mathcal{N}(\sqrt{1 - \beta_t} x_{t-1}, \beta_t I) \quad (2.1)$$

Where the variance is given by the noise schedule $\beta(t)$, which is a linear interpolation from β_{start} to β_{end} , such that in the beginning, the noise added is minimal and gradually increases every step. [HJA20]

The forward steps from x_0 to x_t are simply the multiplication of the individual steps [HJA20]:

$$q(x_t|x_0) = \prod_{s=1}^t q(x_s|x_{s-1}) \quad (2.2)$$

$$= \prod_{s=1}^t \mathcal{N}(\sqrt{1-\beta_s}x_{s-1}, \beta_s I) \quad (2.3)$$

$$= \prod_{s=1}^t \mathcal{N}(\sqrt{\alpha_s}x_{s-1}, (1-\alpha_s) I) \quad (2.4)$$

with $\alpha_s = 1 - \beta_s$. This means that every step, x is scaled down with α_t and noise is added, as illustrated for a one-dimensional input in Figure 2.2.

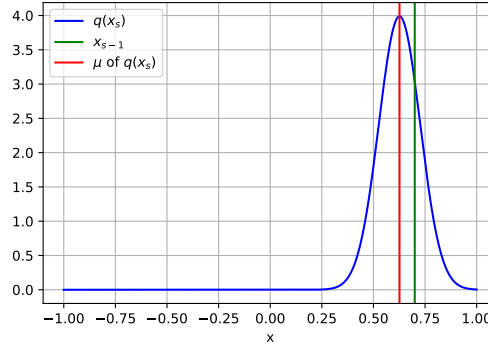


Figure 2.2: Adding Gaussian noise

Using $\bar{\alpha}_t = \prod_{s=1}^t \alpha_s$ and $\epsilon = \mathcal{N}(0, I)$ we can write [HJA20]:

$$q(x_t|x_0) = \prod_{s=1}^t q(x_s|x_{s-1}) \quad (2.5)$$

$$= \mathcal{N}(\sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t) I) \quad (2.6)$$

$$= \sqrt{\bar{\alpha}_t}x_0 + (1-\bar{\alpha}_t) \mathcal{N}(0, I) \quad (2.7)$$

$$= \sqrt{\bar{\alpha}_t}x_0 + (1-\bar{\alpha}_t) \epsilon \quad (2.8)$$

Therefore the forward process is done by sampling x_t from $\sqrt{\bar{\alpha}_t}x_0 + (1-\bar{\alpha}_t) \epsilon$. During training, the diffusion model θ predicts the noise $\epsilon_\theta = \epsilon_\theta(x_t, t)$. The predicted noise is then compared with the actual noise ϵ , and the mean square error (MSE) 2.9 between the two is computed. [HJA20]

$$\text{MSE}(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2 \quad (2.9)$$

The model aims to minimize this error, thus a gradient descent step is done on this loss (to improve future predictions). This is the core principle behind the Training Algorithm 1 [HJA20], where this is done with different randomly sampled timesteps until the loss converges. [HJA20]

Algorithm 1 Training [HJA20]

```

repeat
   $x_0 \sim q(x_0)$ 
   $t \sim \text{Uniform}(\{1, \dots, T\})$ 
   $\epsilon \sim \mathcal{N}(0, I)$ 
  Take gradient descent step on
     $\nabla_{\theta} \|\epsilon - \epsilon_{\theta}(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon, t)\|^2$ 
until converged

```

Denoising is the reverse process of noise addition. Here the predicted noise is removed one step at a time. Meaning the probability distribution of x_0 given x_T is [HJA20]:

$$p_{\theta}(x_0|x_T) = \prod_{t=1}^T p_{\theta}(x_{t-1}|x_t) \quad (2.10)$$

$$= \prod_{t=1}^T \mathcal{N}(\mu_{\theta}(x_t, t), \Sigma_{\theta}(x_t, t)) \quad (2.11)$$

Instead of directly learning $\mu_{\theta}(x_t, t)$, it is parameterized, such that ϵ_{θ} is learned (as seen in the training algorithm) [HJA20]:

$$\mu_{\theta}(x_t, t) = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{1 - \alpha_t}{\sqrt{1 - \alpha_t}} \epsilon_{\theta} \right) \quad (2.12)$$

In theory, one could also learn μ directly, but experiments from Jonathan Ho et al. have shown this approach to be less efficient [HJA20]. [HJA20]

This forms the basis for the Sampling Algorithm 2 [HJA20]. The data additionally gets scaled up by $\frac{1}{\sqrt{\alpha_t}}$ which is the reverse to the shrinking by $\sqrt{\alpha_t}$ in the forward process. If the denoising step is not the final one, some additional Gaussian noise is added to make the denoising robust, i.e., the sampling process adheres to Langevin dynamics. [HJA20]

Algorithm 2 Sampling [HJA20]

```

 $x_T \sim \mathcal{N}(0, I)$ 
for ( $t \rightarrow T$ ;  $t > 0$ ;  $t \rightarrow t - 1$ )
  if ( $t > 1$ ) :
     $z \sim \mathcal{N}(0, I)$ 
  else:
     $z \rightarrow 0$ 
   $x_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left( x_t - \frac{1 - \alpha_t}{\sqrt{1 - \alpha_t}} \epsilon_{\theta}(x_t, t) \right) + \sigma_t z$ 
return  $x_0$ 

```

For a mathematical rigorous derivation of the formulas, especially for why this specific parameterization, in Equation 2.12 is chosen, refer to the paper "Understanding Diffusion Models: A Unified Perspective", by Calvin Luo [Luo22]. Note that this "Denoising" interpretation is for likelihood-based diffusion models, i.e., diffusion models with the goal to approximate the true underlying probability distribution of the data. However, diffusion models can also be interpreted using score-based or energy-based frameworks, as demonstrated in the work by Yang Song et al. (see [Son+21]).

The data x is a tensor, how this can represent a robotic structure is described in the following chapter.

2.2 Evogym

To enable fundamental research in automated design, Bhatia et al. have created Evogym, a simulation environment with tasks and benchmark algorithms for co-optimizing the design and control of soft robots.[Bha+21]

2.2.1 Robot Representation

The simulation is based on a voxel representation in two dimensions. Thus, a robot is described by a matrix, where each entry holds the type of voxel present in this position, as illustrated in Figure 2.3. For most tasks, the robot is limited to a 5x5 structure. There are five different voxel types, as listed in Table 2.1.[Bha+21]






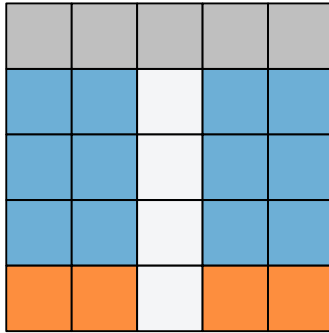
Type	Mathematical Representation	Visual Representation
Empty	0	
Rigid	1	
Soft	2	
Horizontal Actuator	3	
Vertical Actuator	4	

Table 2.1: Types of voxels and their representations

The actuators are soft voxels that can be stretched up to 160% or compressed to 60% of their original length.[Bha+21]



(a) Robot image

$$\begin{bmatrix} 2 & 2 & 2 & 2 & 2 \\ 4 & 4 & 0 & 4 & 4 \\ 4 & 4 & 0 & 4 & 4 \\ 4 & 4 & 0 & 4 & 4 \\ 3 & 3 & 0 & 3 & 3 \end{bmatrix}$$

(b) Voxel matrix representation

Figure 2.3: Example of robot structure in visual and mathematical representation

2.2.2 Task Representation

There are 32 different tasks in Evogym, ranging from walking to climbing and object manipulation, varying in difficulty, classified into easy, medium, and hard tasks. The terrain of a task is made out of rigid, void, and soft voxels. Every task has a different reward function specific to the task. For example, in the walking task, the reward per timestep is the distance covered. [Bha+21]

2.2.3 Structure of Co-Design Algorithms

The problem of finding an optimal robot can be split into two sub-problems. The first part is the control loop, learning the optimal control for a given robot for a given task. The second part is the design, optimizing the mechanical structure of the robot to maximize the achievable reward.[Bha+21]

The control loop is nested in the design loop, and the reward distribution therefore depends on the control loop. However, the focus of Evogym lies in researching the design loop, with all benchmark algorithms using the same control algorithm, Proximal Policy Optimization.[Bha+21]

The design problem is independent of the control loop if the control algorithm finds the optimal control for every structure. Then, the problem reduces to finding an optimum in a discrete design space. Each point in the design space (every possible robot configuration) is mapped to a reward value by running the control algorithm. The design space is restricted to valid robots, where a valid robot contains at least one actuator and all voxels are connected, i.e. there are no multiple disjoint bodies, as shown in Figure A.1.[Bha+21]

Figure 2.4 illustrates a hypothetical example where a robot is made out of two voxels connected horizontally.

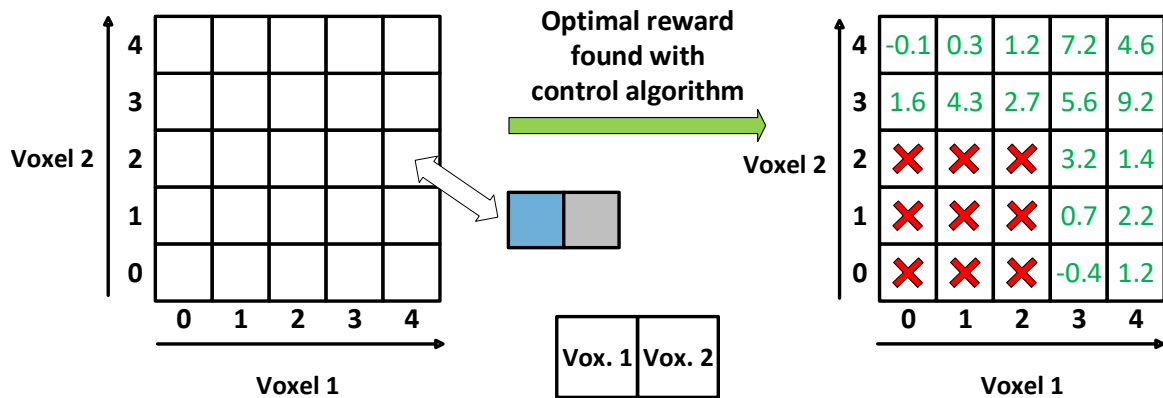


Figure 2.4: Design space for 2 voxel robot

Each axis corresponds to one voxel, and the value of this coordinate gives the voxel type, where the mapping is given in Table 2.1. On the right, a reward distribution for a hypothetical task is given, i.e. every possible configuration is assigned a reward, given by the evaluating of the robot configuration in the control loop. The red crosses indicate invalid robots, because none of the two voxels are an actuator (which would be at coordinates three, for a horizontal actuator, and four for a vertical actuator)

For a robot with a 5x5 structure, this means there are 25 dimensions, spanning $5^{25} \approx 3 \cdot 10^{17}$ configurations (including non-valid robots). The goal is to find the optimum in this design space while evaluating as few robots as possible. The latter constraint arises in practice as each evaluation is computationally expensive and time-intensive.

2.3 Reinforcement Learning

To solve for the control of a given robot structure to maximize the reward for a specific task, the well-established Proximal Policy Optimization method [Sch+17] is used. This reinforcement learning method is both simple and robust, while achieving state-of-the-art results on many benchmarks [Sch+17].

Reinforcement learning is a branch of machine learning that is characterized by the interaction with the environment. The agent observes and interacts with the environment and receives feedback in the form of a scalar reward R for its actions. The goal is for the agent to learn through these actions, given an observation, what actions lead to high rewards in the long term. The state S_t is the agent's internal representation of the environment at time t . [Sut18][Sil15]

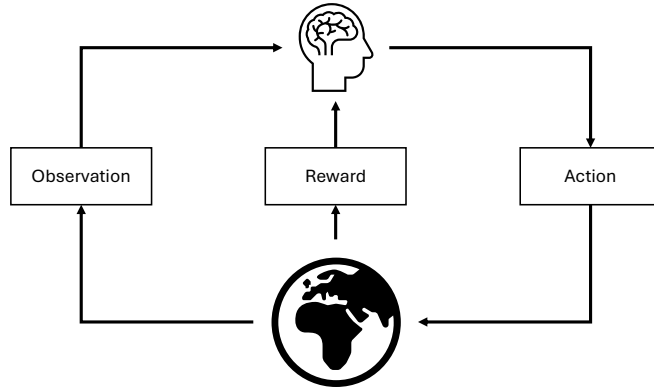


Figure 2.5: The agent interacts with the environment

2.3.1 Markov Decision Process

The environment in reinforcement learning is formally described as a Markov Decision Process (MDP). A state is Markov if it depends only on the current timestep, i.e., the state does not depend on the past. A Markov Decision Process is given by the tuple: $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$. \mathcal{S} denotes a finite set of states, where all states are Markov, and \mathcal{A} denotes a finite set of actions. The state transition probability matrix \mathcal{P} describes the environments dynamics, where the entry $\mathcal{P}_{ss'}$ describes the probability of landing in state s' when taking action a from state s [Sil15]:

$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a] \quad (2.13)$$

\mathcal{R} is the reward function that returns the expected reward of the next timestep when taking action a from state s [Sil15]:

$$\mathcal{R}_s = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a] \quad (2.14)$$

The Goal G_t is the discounted sum of all rewards from timestep t onwards [Sil15]:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \quad (2.15)$$

The discount factor γ is in the range $[0, 1]$ and accounts for the fact that rewards in the future are less certain and thus should be weighted less. If only the immediate reward were important, γ would be set to zero.

The policy $\pi(a|s)$ of an agent describes the probabilities of actions, given the state the agent is in [Sil15]:

$$\pi(a|s) = \mathbb{P}[A_t = a \mid S_t = s] \quad (2.16)$$

The value function is an estimate of how good a given state is, i.e., how much reward is expected in the future, when following policy π , starting from state s [Sil15]:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] \quad (2.17)$$

Similarly, the action-value function describes the expected reward for an action a taken from state s , and then following policy π [Sil15]:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \quad (2.18)$$

A model predicts what will happen in the environment, which state the agent will be in when taking a specific action, and what reward it will get, based on a specific action. [Sil15][Sut18]

A reinforcement learning method can optimize the value function (value-based), the policy (policy-based), or a combination of the two (actor-critic). It can additionally model the environment (model-based) or be model-free. [Sil15][Sut18]

2.3.2 Model-Free Reinforcement Learning

In model-free reinforcement learning, no knowledge of \mathcal{P} or \mathcal{R} is required to solve the problem. Monte-Carlo Policy Evaluation estimates the value function v_π and action-value function q_π , by gathering data under the current policy and updating its estimates:

Algorithm 3 Every-Visit Monte-Carlo Policy Evaluation [Sil15]

Initialize $N(s) = 0$ and $S(s) = 0 \quad \forall s \in \mathcal{S}$

Initialize $N(s, a) = 0$ and $S(s, a) = 0 \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$

Repeat until convergence:

 Generate an episode $\{(s_1, a_1, r_2), (s_2, a_2, r_3), \dots, (s_T, a_T, r_{T+1})\}$ by following policy π

$G \leftarrow 0$

for ($t \rightarrow 1; t < T; t \rightarrow t + 1$)

$G \leftarrow \sum_{k=t+1}^T r_k$

$N(s_t) \leftarrow N(s_t) + 1$

$N(s_t, a_t) \leftarrow N(s_t, a_t) + 1$

$S(s_t) \leftarrow S(s_t) + G$

$S(s_t, a_t) \leftarrow S(s_t, a_t) + G$

$V(s_t) \leftarrow \frac{S(s_t)}{N(s_t)}$

$Q(s_t, a_t) \leftarrow \frac{S(s_t, a_t)}{N(s_t, a_t)}$

Essentially the expectations of v_π and q_π are replaced through sampling and by the law of large numbers [Sil15]:

$$V(s) \rightarrow v_\pi(s) \text{ as } N(s) \rightarrow \infty \quad (2.19)$$

$$Q(s, a) \rightarrow q_\pi(s, a) \text{ as } N(s, a) \rightarrow \infty \quad (2.20)$$

Instead of using $V(s)$, which is essentially a lookup table, where every state s is assigned a value, we use a function V^θ , which is typically comprised of a neural network with parameters θ , that approximates $V(s) \quad \forall s \in \mathcal{S}$. Similarly $Q(s, a)$ is approximated by Q^θ . [Sut18][Sil15]

2.3.3 Policy Gradient Methods

Proximal policy optimization (PPO), the RL method used in this thesis, is model-free and, as the name suggests, a policy-based method. The policy π is parameterized by θ : $\pi_\theta(s|a) = \mathbb{P}[a | s, \theta]$. The value function approximation and action-value approximation under the policy π_θ are denoted as V^{π_θ} and Q^{π_θ} respectively.

PPO also utilizes the Advantage function, making PPO an actor-critic method. The advantage function is defined as the difference between the action-value function and the value function, i.e., how much more (or less) reward is expected when taking action a from state s [Sil15]:

$$A^{\pi_\theta}(s, a) = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s) \quad (2.21)$$

In episodic environments, the quality of a policy can be evaluated by the start-state formulation, given by the expected reward of the episode, i.e., the value function evaluated at the starting state[Sil15][Sut+99]:

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}[v_1] \quad (2.22)$$

The policy gradient method aims to find the θ that maximizes J with gradient ascent. Where the parameters θ get updated by[Sil15]:

$$\theta = \theta + \Delta\theta = \theta + \alpha \nabla_\theta J(\theta) \quad (2.23)$$

with step size or learning rate α . We can derive $\nabla_\theta J(\theta)$ using the policy gradient theorem [Sut+99]:

$$\frac{\partial J}{\partial \theta} = \sum_s d^\pi(s) \sum_a \frac{\partial \pi_\theta(s, a)}{\partial \theta} Q^{\pi_\theta}(s, a) \quad (2.24)$$

The theorem gives the gradient on the expected return for any Markov Decision Process, in the start-state formulation. $d^\pi(s)$ denotes the stationary distribution of states under the policy π , i.e. the probability of being in state s when following the policy, or how frequently state s is visited relative to the total number of visits.

Inserting[Sut18]:

$$\frac{\partial \pi_\theta(s, a)}{\partial \theta} = \pi_\theta(s, a) \frac{\partial \log \pi_\theta(s, a)}{\partial \theta}$$

into Equation 2.24 we get:

$$\frac{\partial J}{\partial \theta} = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s, a) \frac{\partial \log \pi_\theta(s, a)}{\partial \theta} Q^{\pi_\theta}(s, a) \quad (2.25)$$

Using the expectation under the state-action distribution induced by policy π_θ results in:

$$\frac{\partial J}{\partial \theta} = \mathbb{E}_{\pi_\theta} \left[\frac{\partial \log \pi_\theta(s, a)}{\partial \theta} Q^{\pi_\theta}(s, a) \right] \quad (2.26)$$

or equivalently[Sut+99]:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a) Q^{\pi_\theta}(s, a)] \quad (2.27)$$

As the expectation of a baseline multiplied with the score function $\nabla_\theta \log \pi_\theta(s, a)$ is zero, we can subtract it from Equation 2.27 without changing the expectation[Sil15]:

$$\mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a) B(s)] = \sum_{s \in \mathcal{S}} d^{\pi_\theta}(s) \sum_a \nabla_\theta \log \pi_\theta(s, a) B(s) \quad (2.28)$$

$$\begin{aligned} &= \sum_{s \in \mathcal{S}} d^{\pi_\theta}(s) B(s) \nabla_\theta \sum_{a \in \mathcal{A}} \pi_\theta(s, a) \\ &= 0 \end{aligned} \quad (2.29)$$

This can reduce variance. A suitable baseline function is the state value function $V^{\pi_\theta}(s)$ such that Equation 2.27 can be reformulated with the advantage function, given in Equation 2.21[Sch+17]:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a) A^{\pi_\theta}(s, a)] \quad (2.30)$$

We can reformulate the general policy gradient (Equation 2.30) to obtain the default gradient objective of policy optimization methods[Sch+17]:

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t [\log \pi_\theta(a_t | s_t) \hat{A}_t] \quad (2.31)$$

i.e. the empirical average over the log probability of action a_t given the state s_t under the current policy π_θ multiplied with an estimator of the advantage function \hat{A}_t . This reformulation makes explicit that the expectation over the policy's action distribution \mathbb{E}_{π_θ} is estimated empirically based on sampled trajectories $\hat{\mathbb{E}}_t$. Similarly A^{π_θ} is replaced by the estimate \hat{A}_t and $\pi_\theta(s, a)$ becomes $\pi_\theta(s_t, a_t)$ which is equivalent to the notation $\pi_\theta(a_t | s_t)$, which makes the relation between the action and state explicit, meaning that the action a is taken from state s . Thus $J(\theta)$ is estimated by L^{PG} .

[Sut+99][Sil15][Sch+17]

2.3.4 Trust Region Policy Optimization

Optimizing on L^{PG} (2.31) results in large policy updates, leading to unstable behaviour. Trust Region Policy Optimization Methods (TRPO) introduce an additional constraint given by the Kullback-Leibler (KL) divergence of the updated and original (old) policy to limit the size of policy updates. TRPO thus solves the constrained optimization problem[Sch+17]:

$$\underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)} \hat{A}_t \right] \quad (2.32)$$

$$\text{subject to} \quad \hat{\mathbb{E}}_t [\text{KL}[\pi_{\theta_{\text{old}}}(\cdot | s_t), \pi_\theta(\cdot | s_t)]] \leq \delta \quad (2.33)$$

with δ determined heuristically, usually set to 0.01. [Sch15]

2.3.5 Proximal Policy Optimization

Proximal Policy Optimization makes use of the same idea of conservative policy updates. Instead of solving a constraint problem, PPO directly incorporates this idea in the clipped surrogate objective.[Sch+17]

With the probability ratio

$$r_t(\theta) = \frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)} \quad (2.34)$$

the unconstrained TRPO objective $L^{CPI}(\theta) = \hat{\mathbb{E}}_t [r_t(\theta) \hat{A}_t]$ would lead to uncontrolled large policy updates.[Sch+17]

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t [\min(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t)] \quad (2.35)$$

By clipping the objective L^{CPI} , L^{CLIP} is equal to L^{CPI} when $r_t(\theta)$ is in the range $[1 - \epsilon, 1 + \epsilon]$,

that is, when π_θ is similar to $\pi_{\theta_{\text{old}}}$. By additionally optimizing on the minimum of the unclipped and clipped L^{CPI} , the clipping only takes effect if it worsens the objective function. This results in more conservative updates when improving the policy, i.e., the policy updates are limited to prevent excessive updates leading to instability while still enabling correcting for mistakes. This means, when the estimated advantage function \hat{A}_t is negative, but the probability of this action increased under the new policy π_θ , this effect is allowed to be reversed, and the same applies for when $\hat{A}_t > 0$ and $r_t(\theta) < 0$. [Sch+17]

The final objective function for the PPO algorithm is [Sch+17]:

$$L^{\text{PPO}}(\theta) = \hat{\mathbb{E}}_t [L^{\text{CLIP}}(\theta) - c_1 L^{\text{VF}}(\theta) + c_2 S[\pi_\theta](s_t)] \quad (2.36)$$

with L^{VF} being the squared error loss $(V_\theta(s_t) - V_t^{\text{targ}})^2$ and S the entropy bonus for the policy π_θ at s_t . This simultaneously optimizes the value function and the policy while encouraging exploration. [Sch+17] In the original Proximal Policy Optimization paper, the authors found that for most simulation environments a clip parameter of $\epsilon = 0.2$ performed best [Sch+17]. For Evogym $\epsilon = 0.1$ was chosen [Bha+21]. All parameters used in Evogym are listed in Table A.1 [Bha+21].

2.3.6 Algorithm Implementation

The concrete algorithm implementation is illustrated in Figure 2.6. First, the N actors gather

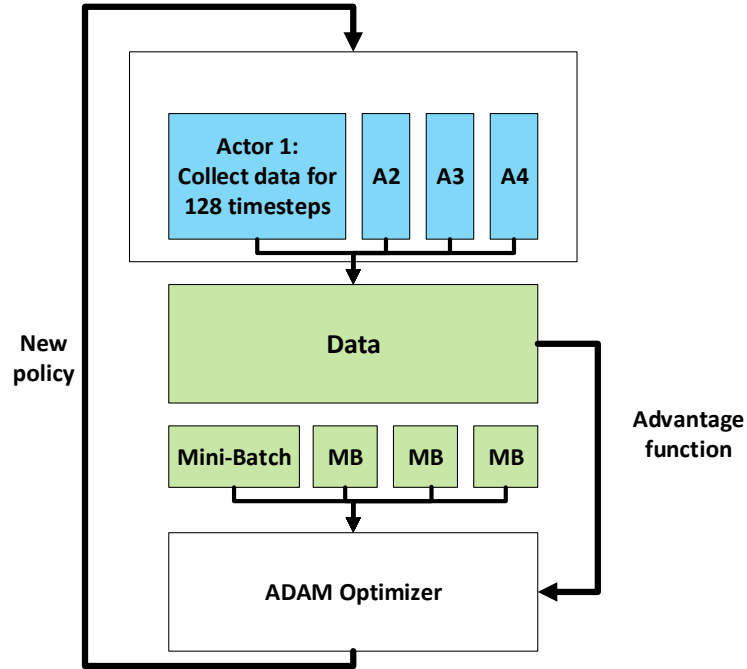


Figure 2.6: Schematic of PPO algorithm

data, which is used to compute the advantage function estimates \hat{A}_1 to \hat{A}_T . Where \hat{A}_t is given by [Sch+17]:

$$\hat{A}_t = -V(s_t) + R_t + \gamma R_{t+1} + \dots + \gamma^{T-t+1} R_{T-1} + \gamma^{T-t} V(s_T) \quad (2.37)$$

Then, K mini-batches of size M are sampled from the combined data and used to optimize L^{PPO} in K epochs. The new policy is then used for the agents to collect new data. This cycle repeats itself until the maximum number of combined timesteps T_{\max} is reached. Evogym uses ADAM as its optimizer[Bha+21][KB17]. The parameters used in Evogym are listed in Table A.1[Bha+21].

[Sch+17]

The algorithm used for the control loop in Evogym is implemented by stable-baselines3 [Raf+21].

Chapter 3

Related Work

Design optimization has already been solved; by nature. There exist many complex, highly efficient, and optimized organisms in nature. This is why many of the current state-of-the-art solutions to the design optimization problem of robotic systems are based on the principles that led to these complex living creatures.

3.1 Genetic Algorithm

One such principle, survival of the fittest, is the theory of Charles Darwin describing biological evolution, and is the basis for the genetic algorithm (GA). The idea is that only the best specimens of the species survive and reproduce, passing on their characteristics to their offspring. In addition, random mutation introduces slight variations that could lead to better or worse fitness, enabling exploration. In Evogym, the genetic algorithm only utilizes mutation to search for an optimal solution; a crossover between two robots is not implemented.[KCK21][Bha+21]

The algorithm starts with a random set of robots that get evaluated on a task. The best-performing robots, the survivors, carry over to the next generation and are allowed to reproduce. The reproduction is a random mutation, i.e. every voxel of the robot has a 10% chance of changing.[Bha+21]

Algorithm 4 Genetic Algorithm

```
 $R_1, \dots, R_n \rightarrow \text{SampleRandomRobots}(n)$ 
 $S \rightarrow \{\}$ 
 $g \rightarrow \text{Number of generations}$ 
for ( $i \rightarrow 0; i < g - 1; i \rightarrow i + 1$ )
  for ( $j \rightarrow 0; j < n - 1; j \rightarrow j + 1$ )
     $C_j \rightarrow \text{OptimizeControl}(R_j)$ 
     $r_j \rightarrow \text{GetReward}(R_j, C_j)$ 
   $S \rightarrow \text{UpdateSurvivors}(R_1, \dots, R_n, r_1, \dots, r_n)$ 
 $R_1, \dots, R_n \rightarrow \text{Reproduce}(S)$ 
```

The genetic algorithm performed the best overall out of the benchmark algorithms[Bha+21]. However, there are still many tasks the algorithm is not able to complete[Bha+21]. Additionally, the produced robot structures are not very intuitive and often look like random noise to the human eye. The performance of the algorithm is also dependent on the initial population, which is randomly sampled, leading to a variance in its results. Since reproduction happens

through mutation, the next generation will be close to the previous one in the design space, which can lead to slow convergence. Some of these problems are solved in CPPN-NEAT.

3.2 CPPN-NEAT

The benchmark algorithm CPPN-NEAT is the most common algorithm for evolving soft robot structures. It combines two methods: Compositional Pattern Producing Networks (CPPNs) and NeuroEvolution of Augmenting Topologies (NEAT).[Bha+21]

3.2.1 NeuroEvolution of Augmenting Topologies (NEAT)

Neuroevolution is a method of reinforcement learning, where instead of learning the weights for a given neural network, the topology of this network is evolving as well. The general outline of the algorithm is similar to that of the genetic algorithm, the crucial difference is that NEAT optimizes the topology of a neural network and not of a robot structure. [SM02]

Algorithm 5 NEAT

```

 $g \rightarrow$  Number of generations
 $G_1, \dots, G_n \rightarrow \text{StartingGenome}(n)$ 
for ( $i \rightarrow 0; i < g - 1; i \rightarrow i + 1$ )
  for ( $j \rightarrow 0; j < n - 1; j \rightarrow j + 1$ )
     $f_j \rightarrow \text{EvaluateFitness}(G_j)$ 
     $G_1, \dots, G_n \rightarrow \text{Reproduce}(G_1, \dots, G_n, f_1, \dots, f_n)$ 

```

In the algorithm, the network architecture is described by a genome, which contains information about all present nodes and the weights of the connections between them, shown in Figure 3.1. The terminology is adopted from genetics, where the genome encodes the phenotype (for example how the DNA encodes observable traits of a human, such as their eye color). [SM02]

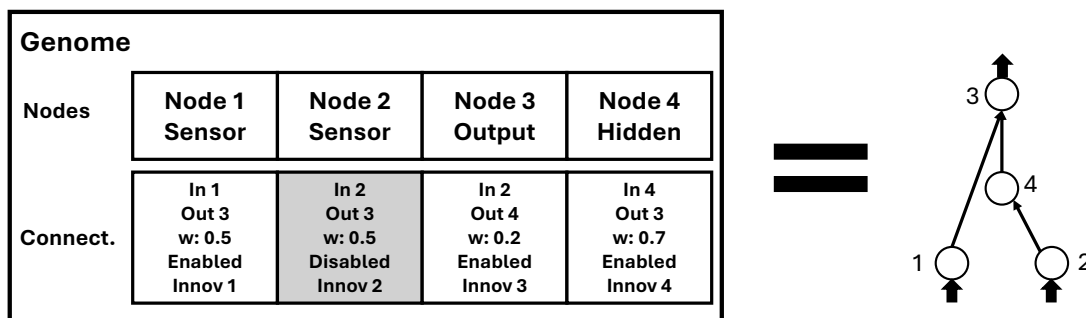


Figure 3.1: Encoding of network architecture in genome

Notice how the initial generation only contains copies of the same genome. In Vanilla NEAT, this genome represents the simplest possible neural network with no hidden layers. This way there is an inherent bias towards simpler structures throughout the search, eliminating the need to constrain or penalize complexity in the network architecture. In contrast to the genetic algorithm, diversity is not achieved by a diverse initial population but rather through the method of reproduction.[SM02]

The core principle for ensuring diversity, i.e., protecting innovation, is speciation. The entire population is divided into different species that are similar in their genetic encoding. Genomes are then only compared within their species, such that initially worse species are given time to develop. The reproduction procedure is as follows: Based on its fitness, each species produces a different number of offspring through a crossover operation, where better-performing species produce more offspring. These offspring undergo random mutation and are then assigned to existing species, or make up a new species if no matching species was found. Finally, the speciated and mutated offspring replace the old genomes and comprise the new generation.[SM02]

Mutation

There are four types of mutation that can occur, three of which alter the topology of the neural network, illustrated in Figure 3.2. A connection between two nodes can be added, a previously disabled connection can become enabled, or a node can be added along an existing connection, in which case the original connection is disabled, and two new connections are added. The fourth type of mutation is simply perturbing the weights of the network.[SM02]

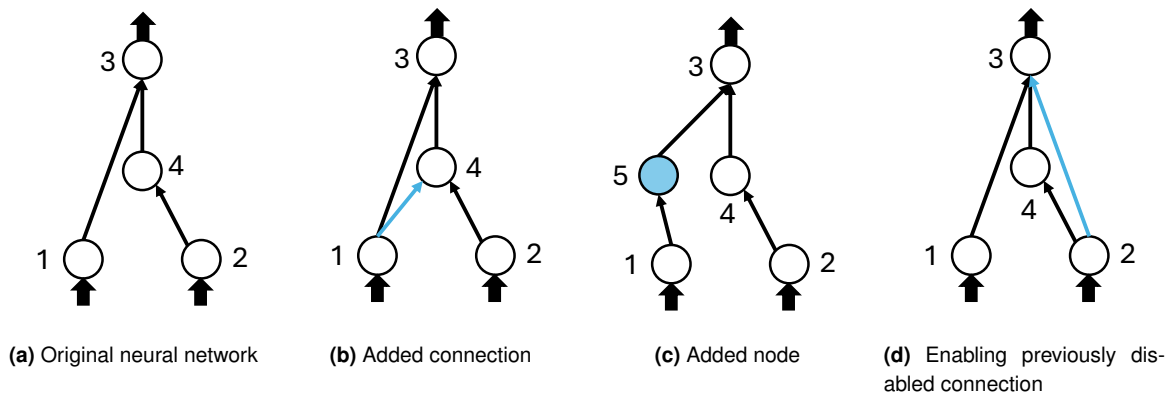


Figure 3.2: Mutation in NEAT

Every added connection is assigned an innovation number that gets encoded together with its weight in the corresponding gene of the genome. This innovation number makes it possible to easily compare and cross two different neural networks.[SM02]

Crossover

Crossover only occurs between two neural networks of the same species. During crossover of two parents, their genomes are aligned using the innovation numbers of their genes. If both parents share the gene (i.e., share the innovation number), the gene is randomly chosen from one of the two (i.e., the weight and status, enabled or disabled). If genes exist that only one parent has (disjoint and excess genes), they are only taken from the better-performing parent. Figure 3.3 shows how the genomes of the neural networks shown in Figure 3.2c (=Parent 1) and Figure 3.2b (= Parent 2) might cross over if Parent 2 is dominant.[SM02]

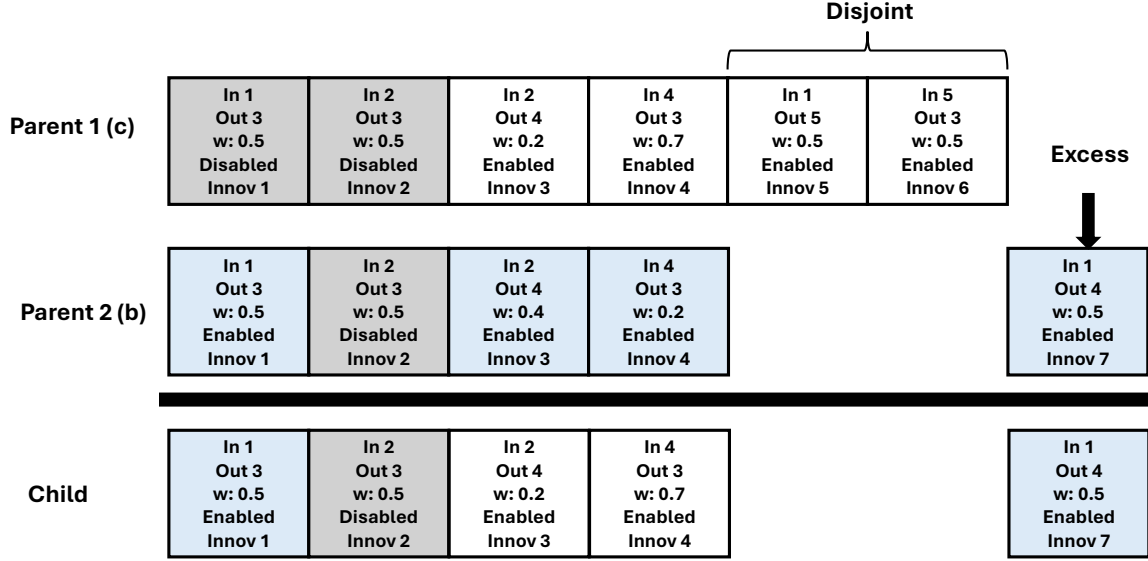


Figure 3.3: Crossover of two genomes

Speciation

Whether two specimens are of the same species is determined by their compatibility, i.e., how many genes their genomes share. To belong to a species the distance δ to the reference genome, given by Equation 3.1 must be smaller than a given threshold δ_t , as illustrated in Figure 3.4. The reference genome is a randomly chosen genome of the previous generation.[SM02]

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \bar{W} \quad (3.1)$$

Here E is the number of excess, and D the number of disjoint genes. N represents the total number of genes of the larger parent, but can be set to one if the genomes are smaller than 20 genes, according to the paper "Evolving Neural Networks through Augmenting Topologies"[SM02]. \bar{W} is the average weight difference of the matching genes, c_1 , c_2 and c_3 are constants, whose default value is 1, 1 and 0.4 respectively.[SM02]

In order to determine how many offspring a species produces, first the adjusted fitness for every genome is determined, given by Equation 3.2.[SM02]

$$f'_i = \frac{f_i}{\sum_{j=1}^n \text{sh}(\delta(i, j))} \quad (3.2)$$

Where $\sum_{j=1}^n \text{sh}(\delta(i, j))$ simply results in the number of genomes that are within distance δ_t of the genome, illustrated in Figure 3.5.[SM02]

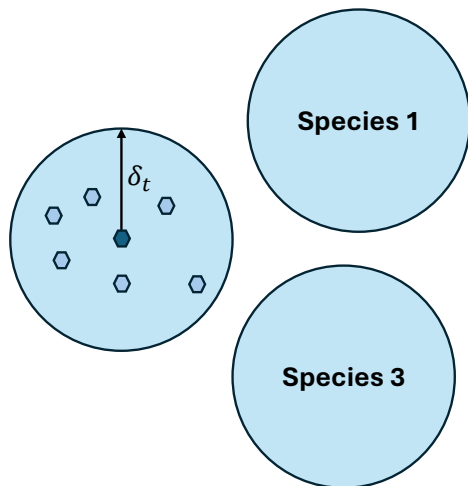


Figure 3.4: Similarity: distance δ

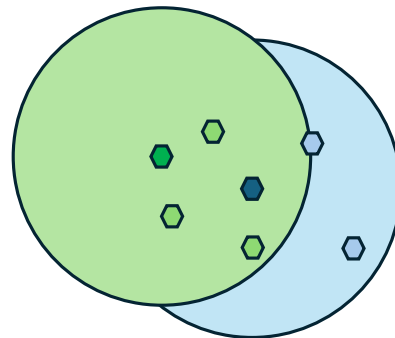


Figure 3.5: Calculating f'_i

The sum of all f'_i of the genomes belonging to a species determines how many offspring the species has. This effectively means that species with regions of high-performing genomes reproduce more, and the number of offspring is independent of the size of the species. Before the species reproduces, the weakest performing genomes (with lowest f_i) are eliminated. Then two parents are chosen at random and reproduce, until the desired number of offspring is reached. Then, for each species, a reference genome is chosen from the parents (the old generation), before the offspring are assigned to species based on their distance to these reference genomes. If none of the reference genomes are within δ_t then a new species is created with that genome as reference. Finally the speciated offspring replace the old generation.[SM02]

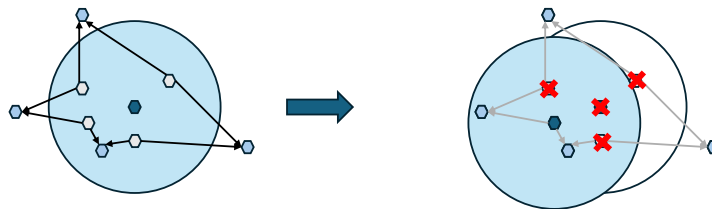


Figure 3.6: Speciation and reproduction

NEAT enables us to optimize the architecture of a neural network. But how does this relate to the design problem? To answer this question we need Compositional Pattern Producing Networks (CPPNs).

3.2.2 Compositional Pattern Producing Networks (CPPNs)

Compositional Pattern Producing Networks presents an alternative model to embryogenesis, the process of how a fertilized egg develops into a complex organism, where the final phenotype is determined by the genetic encoding (the DNA/genome). CPPNs model this generative principle, of how the genome encodes the organism. It argues that this time-intensive process of gradually growing into the organism is purely a physical constraint, and inefficient in simulated evolution. The core idea is that in nature, this process is necessary because

there exists no absolute frame of reference, and thus, each cell divides into different cell types based on what cells are in its vicinity. In artificial embryogenesis, however, we do have this absolute frame of reference. Therefore the entire development of the organism can be skipped. The developmental encoding can instead be modeled as a function, whose inputs are the coordinates of the cell, and the output are activation levels for each type of cell. [Sta07]

This function is a composition of different simple functions, such as sinus and linear functions. In essence, the composition of these functions replaces the interaction of cells. This composition of functions can be represented as a graph, where every node represents a simple function. In this way, the representation becomes very similar to that of a neural network, with the only difference being that in neural networks, the activation functions are typically sigmoid functions, and that the inspiration for neural networks comes from the workings of the human brain, and for CPPNs from the development of embryos. [Sta07]

3.2.3 Combining CPPN and NEAT

Now we can find the optimal network architecture of the CPPN, where a CPPN outputs the robot, by evolving its architecture with NEAT. In Evogym, the CPPN takes the x and y coordinates of a voxel (and its distance to the center) as its inputs and outputs the activation levels for each voxel type (=cell type). This way, by querying the CPPN for every voxel, and taking the voxel with the highest activation, the robot structure is retrieved[Bha+21]. The implementation of CPPN-NEAT in Evogym is based on the PyTorch-NEAT library [Pas+19] and the neat-python library [McI+].

The benchmark algorithm CPPN-NEAT produces output robots that are more structured and do not look like random noise. However, the algorithm performs worse than the genetic algorithm for most tasks, as NEAT has a bias towards simpler structures that are not able to complete more difficult tasks.[Bha+21]

Thus, although the algorithm is more elegant than the genetic algorithm, it has only received limited attention since its publication 17 years ago. The structured outputs are, however, more resembling of natural organisms, with patches of the same cell type that enable coordinated motion[Che+14]. Therefore, there exists a need for a new algorithm that combines the strength of both producing structured results (as CPPN-NEAT) while achieving the comparable (or even better) performance of the genetic algorithm. One promising approach is discussed in the following chapters.

Chapter 4

Designing Robot Structures with Diffusion Models

As explained in Section 2.2.3, finding an optimal robot design is the equivalent of finding the global maximum in the reward distribution over the design space. On the other hand, generative models, especially diffusion models, are a great tool for learning complex data distributions and generating data that fit this distribution, as discussed in Section 2.1. Therefore, using a diffusion model to learn the reward distribution of the design space and generating a structure based on this knowledge is a promising approach and the focus of this thesis.

The goal can be illustrated by mapping the high-dimensional design space into 2D. Figure 4.1 shows an exemplary 2D representation of the reward distribution. Then the objective is to train a diffusion model such that the probability distribution of its outputs approximate the reward distribution, as illustrated in Figure 4.2. This means, sampled structures are very likely to be in a region of high reward.

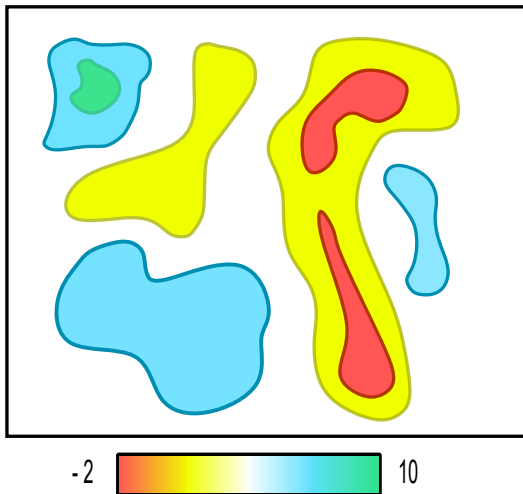


Figure 4.1: Example of simplified reward distribution mapped into 2d

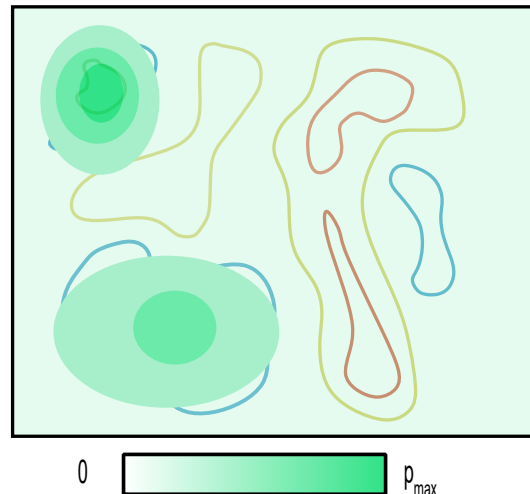


Figure 4.2: Example of probability distribution of samples from fully trained diffusion model mapped into 2d

4.1 Concept: Generational Diffusion Algorithm

When combining diffusion models and design optimization, some contradictions arise. Diffusion models, and deep learning in general, excel at tasks with vast amounts of unlabeled data and finding patterns and representations of this data [LBH15]. In the design optimization

problem, on the other hand, getting data, i.e., evaluating a robot, is computationally very expensive. Additionally, the relation between the diffusion model and the average reward of its outputs is very noisy. For example, the diffusion model DM1 could be better than DM2, but the specific structures sampled from DM1 could still perform worse than those of DM2. Furthermore, the loss function can not be directly dependent on the reward, as the design parameters are discrete and non-differentiable, making gradient descent on the diffusion model's parameters impossible.

A possible solution to this problem is to separate the training of the diffusion model from the evaluation of the robot structures, as described in Algorithm 6.

Algorithm 6 Generational diffusion optimization

```

 $R_1, \dots, R_n \rightarrow \text{SampleRandomRobots}(n)$ 
 $S \rightarrow \{\}$ 
 $g \rightarrow \text{Number of generations}$ 
for ( $i \rightarrow 0; i < g - 1; i \rightarrow i + 1$ )
  for ( $j \rightarrow 0; j < n - 1; j \rightarrow j + 1$ )
     $C_j \rightarrow \text{OptimizeControl}(R_j)$ 
     $r_j \rightarrow \text{GetReward}(R_j, C_j)$ 
   $S \rightarrow \text{UpdateSurvivors}(S, R_1, \dots, R_n, r_1, \dots, r_n)$ 
   $\theta \rightarrow \text{TrainDiffusionModel}(S)$ 
   $R_1, \dots, R_n \rightarrow \text{SampleNewGeneration}(\theta)$ 
  
```

First, the initial population of size n is randomly initialized. Then, its robots are evaluated and the best-performing structures are used to train the diffusion model. This way, the model is not actively trained to generate the best robots, but instead captures the distribution for a given reward range. The benefit of this approach is that the model can be trained on the Mean Square Error (MSE), which can be computed without much computational cost. In the training stage, the survivors of the previous generation (and copies of them) make up the dataset the model is trained on. The model is trained close to overfitting to the data, this is discussed further in Section 4.5. This results in a model whose outputs are similar to the survivors but not identical. The concept is illustrated in Figure 4.3.

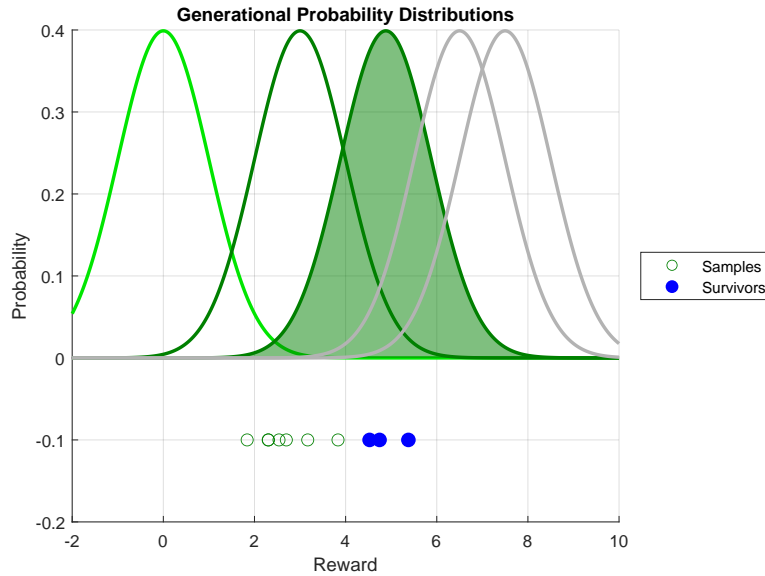


Figure 4.3: Generational diffusion model concept

The Figure shows the transition from the second to the third generation (both have their distribution curves drawn in dark green, the latter has its curve shaded in). For simplicity, we shall refer to the probability distribution of the rewards of the sampled structures of the diffusion model of a generation simply as the reward distribution of this generation. The samples of the second generation have an average reward of approximately three. The best structures of this generation (highlighted in blue), have a mean of approximately five and are now the basis for the third generation. Training the diffusion model on this input data is expected to lead to a reward distribution shaped in a way such that the most likely samples achieve a reward close to that of the input structures (= five). Note that if the model is overfitting to the survivors, its training data, then it will only reproduce the survivors (and their rewards), i.e. the curve is not smooth but has sharp peaks. Then, this process is repeated, gradually increasing the maximum of the generation's reward distribution and the maximum fitness reached.

Essentially, this step of training on the survivors and then sampling from the model is comparable to the mutation step in the genetic algorithm. The crucial difference is that the diffusion model tries to approximate the reward distribution, and the sampled structures therefore have a similar reward as the survivors, but not necessarily a similar mechanical structure and topology. The model also trains on all survivors at once, instead of mutating individual structures. Looking at the exemplary reward distribution of Figure 4.1, assuming the space is mapped such that similar structures are close to each other, every robot of the new generation with the genetic algorithm will be very close to one of the survivors. The robots found through sampling the diffusion model should instead be in similar reward regions.

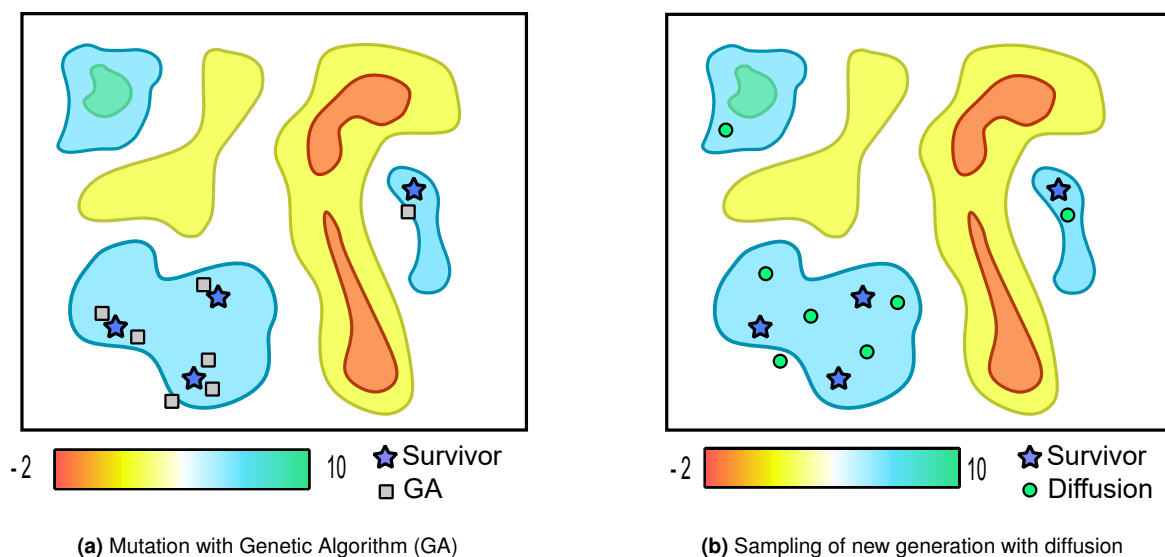


Figure 4.4: Illustration of getting the new generation with diffusion vs genetic algorithm

4.2 Forward Process

As described in Section 2.1, a diffusion model learns by first adding noise to an input, and then learning to predict the noise for the noisy input. Most common diffusion models work with images with many pixels (for example 256 x 256) and a wide range of values (for example 3 x 0 to 255 per pixel for an RGB image). Furthermore, if two pixels have a similar value, they have similar characteristics in reality. For example, in a black and white image,

the pixel values 200 and 202 appear quite similar to the human eye, both as light grey, while the pixel values 10 and 200 are quite different. Thus, the first task is to find a reasonable order for the different voxel types.

There are of course multiple different orderings that are plausible, in the following two examples are discussed. In both options, the ordering is from void to soft, soft but actuated to rigid, as illustrated in Figure 4.5. The noise addition can be cyclic (Option 1) or acyclic (Option 2).

It is noted that for the forward process, this is irrelevant, but in training, the noise addition and the denoising process should be consistent; thus, when using a cyclic forward process, the denoising process is also cyclic. Heuristically, the acyclic noise processes lead to better performance. One possible explanation is that the model can overshoot when it is very confident. For instance, the values in the range $[-100, -1]$ are mapped to a void pixel in the acyclic case, whereas in the cyclic case, these values correspond to different types of voxels.

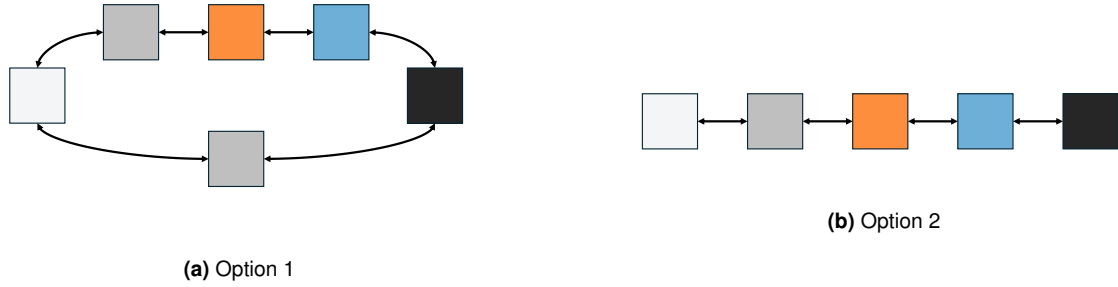


Figure 4.5: Methods for adding noise in discrete design space

A noise addition where every type of voxel is present only once also makes it simpler to ensure that all voxels are equally likely when sampling random noise.

The robot data gets scaled to a range of $[-1, 1]$ while processing in the diffusion model, and the samples get re-scaled, rounded, and mapped back to voxel types before evaluating them. Directly scaling from $[-1, 1]$ to $[0, 4]$ would lead to an under representation of the edge values when sampling random noise. Instead, scaling to $[-0.5, 4.5]$ ensures that all voxels are equally likely when sampling random noise (see Figure 4.6).

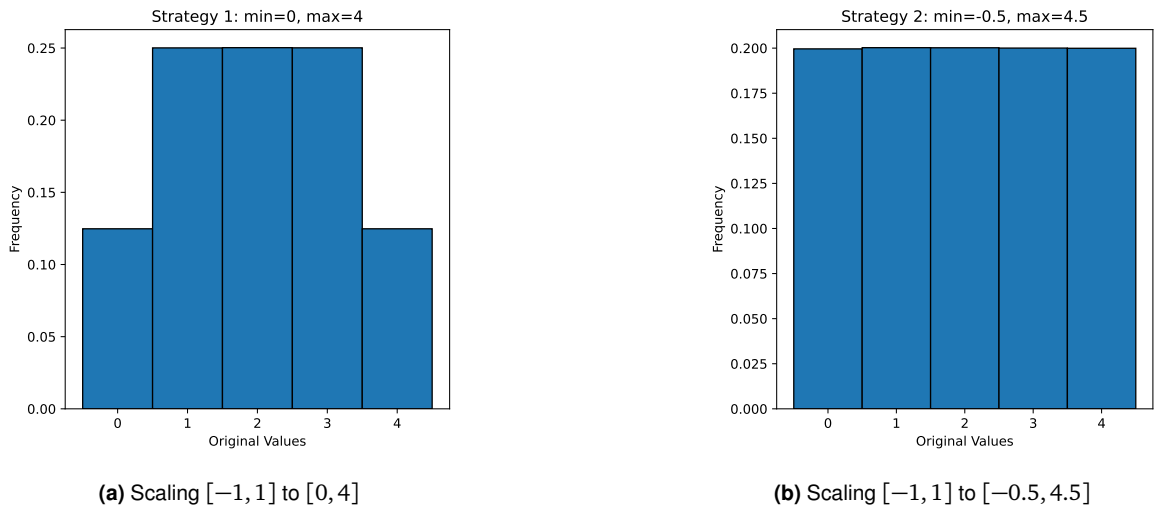


Figure 4.6: Distribution of voxel types from 10^7 random robots (Option 2)

As described in the Algorithm 1 of Section 2.1 [HJA20], during training, a random timestep is sampled from $[1, T]$ and the corresponding noise gets added in one step. Figure 4.7 illustrates the forward process for different timesteps (with $T=500$).

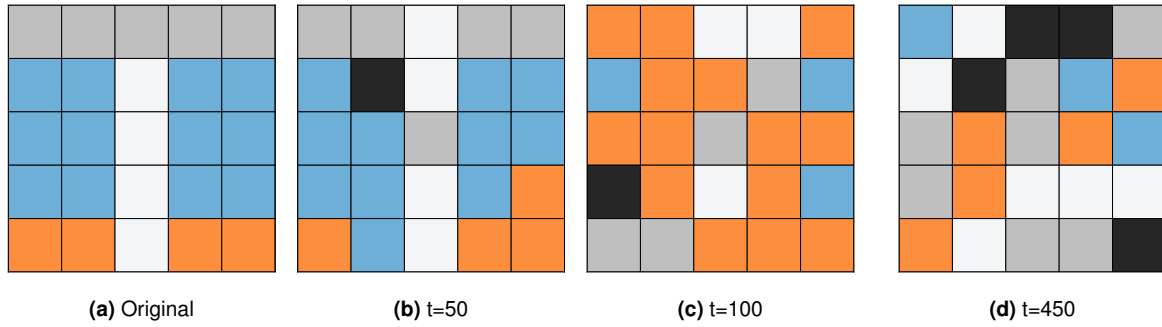


Figure 4.7: Forward process

For illustration purposes these structures were re-scaled and rounded. The model performed best, when it worked with the raw representations, where the robots are only mapped to their discrete voxel values before evaluating them in the control loop.

4.3 Model Architecture

The second part of the training is denoising the noisy inputs. By comparing the predicted noise with the actual noise, the model can improve its predictions over time. The specific architecture of the model for the design optimization is based on the U-Net architecture, as is typical for diffusion models. The code implementation was inspired by and built on the diffusion model of Raha Ahmadi, which was designed to produce realistic images of flowers [Ahm23].

The architecture of the diffusion model is illustrated in Figure 4.8.

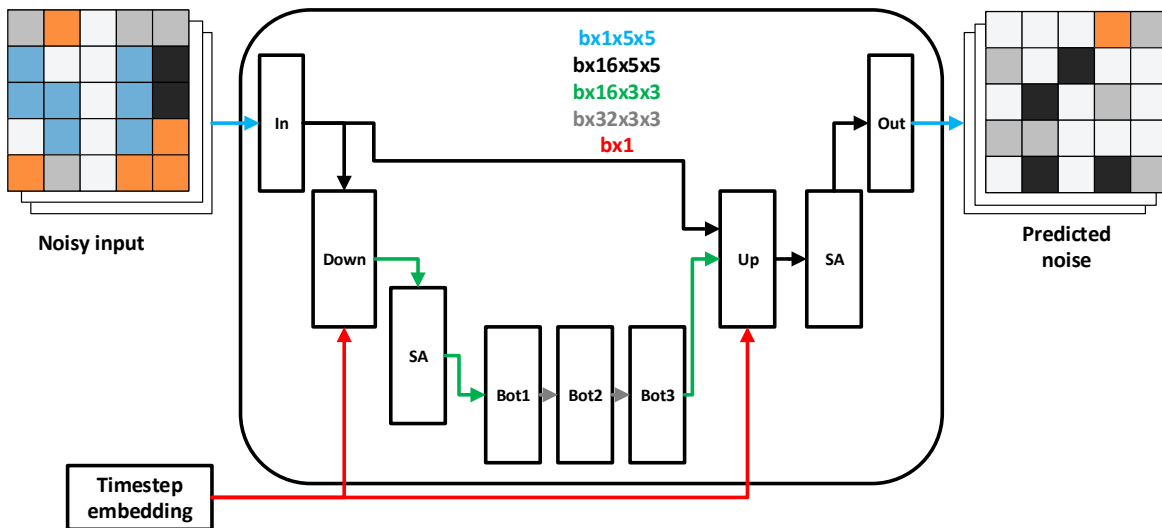


Figure 4.8: Model architecture

The input that gets fed into the model is a batch of normalized, noisy robot structures,

obtained by adding noise to the survivors of the previous generation. Each robot has a size of $1 \times 5 \times 5$, i.e., one channel per voxel. Thus, the size of the input tensor is $b \times 1 \times 5 \times 5$, with batch size b . The first layer (In) transforms the data into 16 channels per pixel, $b \times 16 \times 5 \times 5$ in total. The data is then compressed into a 3×3 representation (Down), processed in the bottleneck layers (Bot1, Bot2, Bot3), and then up-sampled again to the original 5×5 structure (Up). The output layer (Out) finally converts the data from $b \times 16 \times 5 \times 5$ back to $b \times 1 \times 5 \times 5$, such that each output channel can be interpreted as the noise added at a specific voxel. Additionally, Self-Attention layers (SA) are placed after the down and up-sampling layers to capture long-range dependencies.

4.3.1 Down-Sampling

In the Down module, the input ($b \times 16 \times 5 \times 5$) gets compressed into $b \times 16 \times 3 \times 3$ by applying convolutions to the 16 channels. This is implemented in the `nn.Conv2d` module of Pytorch [Pas+19], with a kernel size of 3×3 , stride of 2 and zero padding of 1. Figure 4.9 illustrates (visualizing only one channel), how each kernel (red) goes over the original data (white) with zero padding (light grey) and a stride of two, resulting in the nine evaluations (dark red) that together form the output. This down-sampling process helps extract high-level features while reducing the spatial resolution.

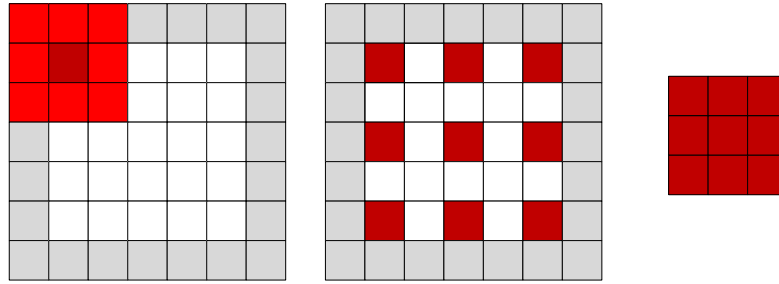


Figure 4.9: Down-sampling visualized

4.3.2 Up-Sampling

The Up module is the counterpart to data compression to bring the data back to its original size. This is done through bi-linear interpolation, done on each channel independently, as illustrated in Figure 4.10, where the position index (1.5, 1.5) refers to an interpolation with the indices (1,1), (1,2), (2,1) and (2,2) of the original $1 \times 3 \times 3$ data. The code is implemented in the `nn.Upsample` module of Pytorch [Pas+19].

4.3.3 Self Attention

The core of the self-attention (SA) layers is comprised of the `nn.MultiheadAttention` module from Pytorch [Pas+19], creating four attention heads. These help to capture longer relationships and embed positional context. For example, in the second SA layer, the channels of the voxel in the top left corner would not be influenced by the voxel in the bottom right, because it is out of reach of the kernel. With the SA layers however, every voxel can influence

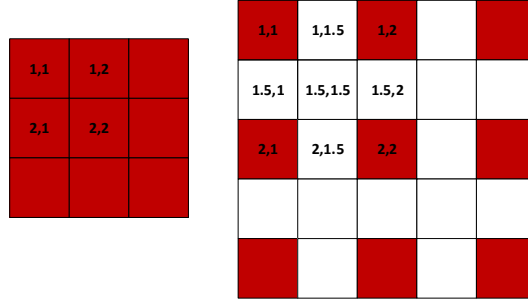


Figure 4.10: Up-sampling visualized

the noise prediction. Through the SA layers, global context modeling is added, such that the absolute position of a voxel is taken into account. [Vas17]

4.3.4 Double Convolution

There are many double convolutional modules in the diffusion model. Some increase, some decrease and some keep the number of channels. In essence, this module applies two convolutions sequentially, which are similar to the down-sampling convolution, except that the stride is set to one, such that the output structure has the same dimensions as the input. The kernel has size $n_{in} \times 3 \times 3$, with n_{in} being the number of input channels. Applying as many kernels as the number of output channels (n_{out}) results in changing the number of channels if $n_{out} \neq n_{in}$. The code is implemented once more in the `nn.Conv2d` module of Pytorch [Pas+19]. Additionally, there are normalization layers after each convolution, and an activation function in between the convolution layers, implemented in Pytorch [Pas+19] in the `nn.GroupNorm` and the `nn.GELU` modules, respectively.

4.4 Hyperparameter Tuning

Due to the low dimensionality of the data, i.e. 5×5 matrices, the architecture of the diffusion model is kept simple, with only 90833 parameters in total.

The most important hyperparameters of the model are the maximum timestep T , the forward process variances β_1 , β_T , and the learning rate l_r . T and β_T must be large enough, such that the original input is nearly completely destructed, such that the model learns the reverse process during training starting from near random noise. This ensures that meaningful results are generated when sampling and denoising random noise. Increasing T beyond this point has no benefit, but takes more computational power when sampling. β_T and T are related, as a higher $\beta_T = \beta_{end}$ results in more noise added per step, thus requiring fewer timesteps in total to achieve the same noise level. To find the optimal relation between these two parameters, the maximum noise ($t=T$) was added to 10^5 zero matrices and the mean and variance of the result were evaluated. The average result should be Gaussian noise, i.e., the mean should be close to zero, and the variance should be close to one. The results of these experiments are given in the Tables A.2, A.3, A.4, A.5, A.6 and their findings are summarized in Table 4.1.

As expected, the higher β_{end} , the fewer steps are needed to achieve Gaussian noise. However, the more noise is added per step, the harder it is for the model to predict the added noise. A small β_{end} , on the other hand, leads to better training but a higher T_{max} .

β_{end}	T_{max}	μ	σ^2
0.01	600	-0.0876	0.9536
0.015	600	-0.0414	0.9881
0.02	500	-0.0323	0.9946
0.03	400	-0.0189	0.9987
0.05	200	-0.0314	0.9931

Table 4.1: Combinations of β_{end} and T_{max} for achieving near Gaussian noise

To determine which β_{end} is small enough, multiple training sessions with varying β_{end} are conducted. The sessions are performed on a dataset containing copies of a single robot 2.3, with an epoch of eight batches with eight structures each. ADAM [KB17] is the optimizer, with a learning rate of 3×10^{-4} . β_{start} is set to 10^{-4} for all training sessions. The results are shown in Figure 4.11.

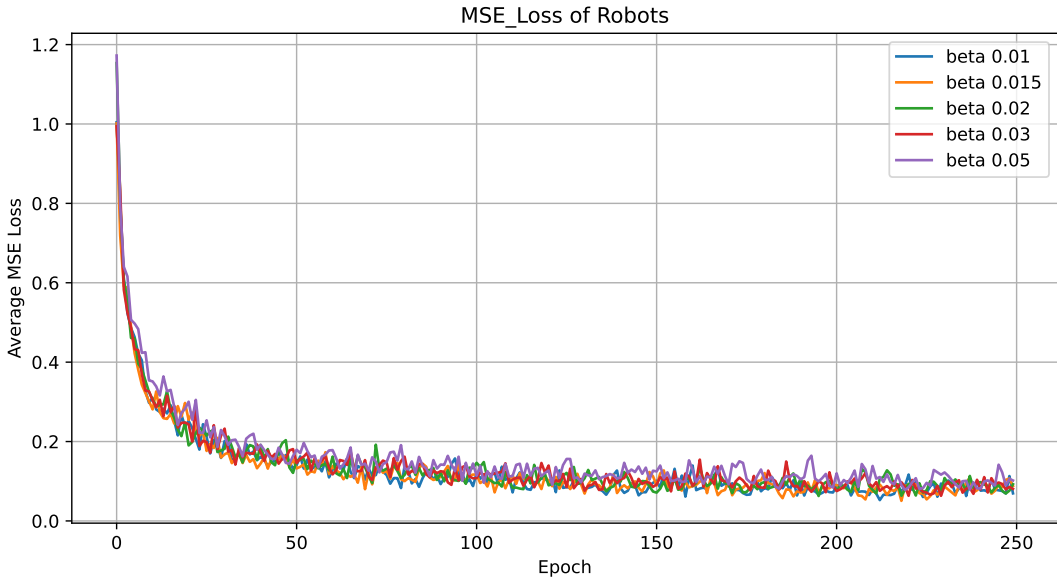


Figure 4.11: MSE loss for different β_{end}

For all β_{end} , the loss converges at approximately 0.1, effectively overfitting to the data. This shows that the model is robust to changes in this parameter. Inspecting the plot closer, in Figure A.2, it can be seen that large β_{end} lead to larger fluctuations and slightly worse convergence. Although this fluctuation is likely negligible, in the following, the safe compromise of $\beta_{end} = 0.02$ is chosen, the same value as used in the experiments of the original paper introducing diffusion models [HJA20].

Finally, the same experiment is run, using $\beta_{end} = 0.02$ but with varying learning rates for the optimizer. Figure 4.12 shows that the different learning rates all converge at approximately the MSE loss of 0.1, demonstrating robustness in this parameter. As expected, the smaller the learning rate, the longer it takes until the MSE loss 0.1 is reached. Inspecting the plot closer, Figure A.3 shows that the learning rate of 0.001 converges the quickest, but does not fluctuate more than the others after convergence, as seen in Figure A.4.

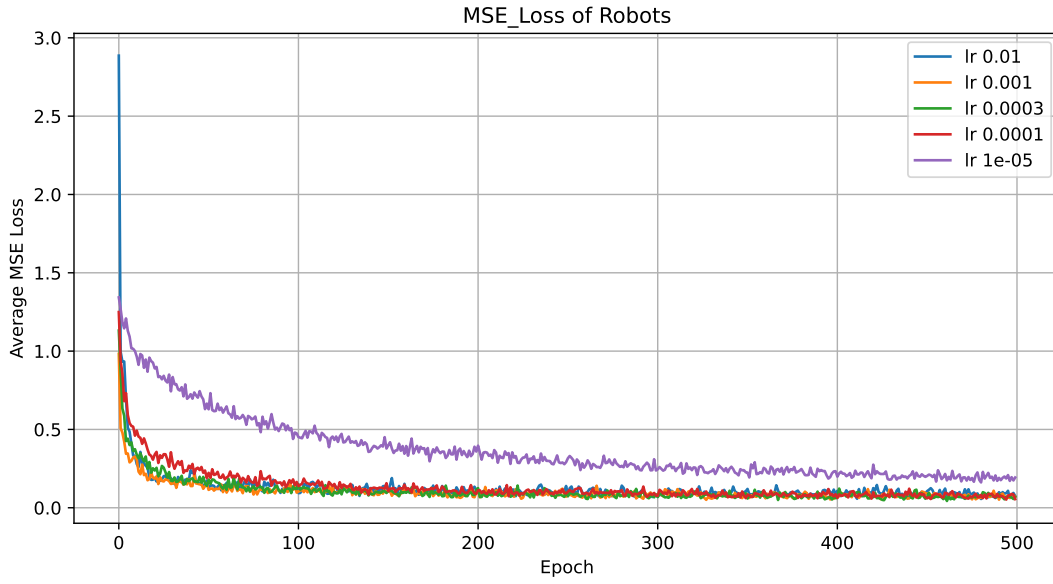


Figure 4.12: MSE loss for different learning rates l_r

4.5 Evaluating Diversity of Generated Robots

Now that we have the hyperparameters that minimize the MSE loss during training, we must evaluate the MSE loss we aim for. Therefore, the MSE loss of 0.1 first needs to be quantified. The input is in the range $[-1, 1]$, where each pixel takes up a fifth of that range, meaning the values $[-1, -0.6]$ map to a void voxel and so on. An MSE of 0.1 means, on average, the voxels are within the range of the correct voxel. Considering the case where the dataset only consists of one specific structure, we can compare the MSE loss of sampled structures with the structure from the dataset before and after training.

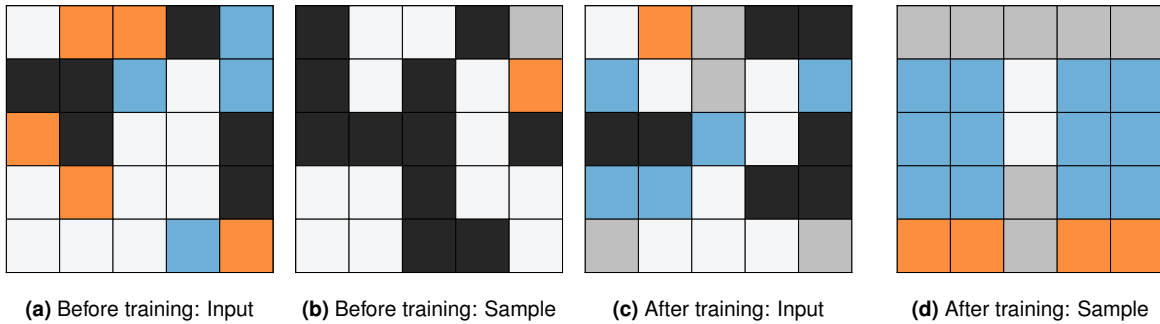


Figure 4.13: Sampling before and after training

Sampling before training leads to an MSE of 494.46. This stems from the fact that the model has not yet learned that the range of values is $[-1, 1]$ and thus the initial sampled tensor has many values outside of this range, many even outside of the range $[-10, 10]$, even though the random noise that was denoised to produce this sample was in the correct range. In Figure 4.13 we can see that the sampled structure after training looks quite similar to the input. The average MSE of samples compared to the structure of the dataset is 1.1.

We can get a better overview of the outputs of the trained model by sampling many samples. When inspecting 100 samples, many of them can be grouped together based on

their characteristics.

The first category of samples are direct replicas (3/100) or replicas with some noise, similar to mutations (10/100), as depicted in Figure 4.14.

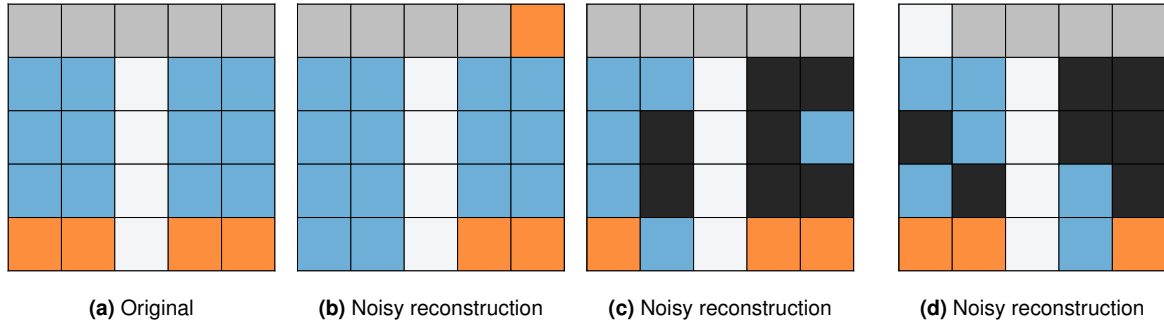


Figure 4.14: Original sample and noisy reconstructions

The next category could be included in the noisy reconstruction but differs in that the robots have an altered shape, as shown in Figure 4.15. They make up 9 out of the 100 samples.

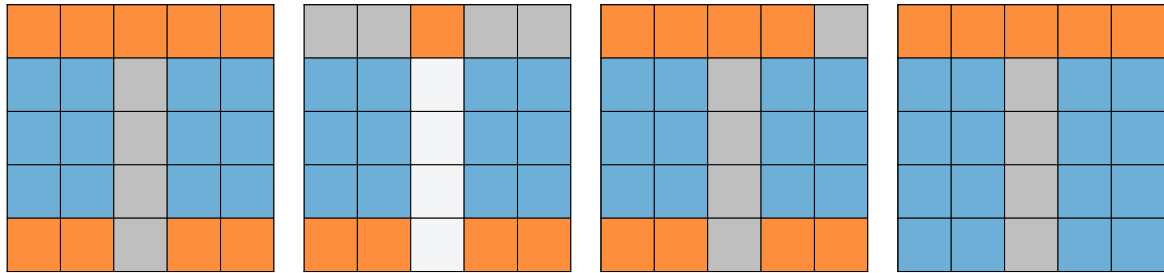


Figure 4.15: Samples with an altered shape

There are also 20 samples with the same structure as the original robot but with different voxel types, as depicted in Figure 4.16, and their noisy reconstructions (34/100), shown in Figure 4.17. This demonstrates that the diffusion model is trying to learn the ground truth of the underlying dataset instead of simply mutating the original data. Despite these robots having a far Euclidian distance in the design space, they have the shape of the original structure.

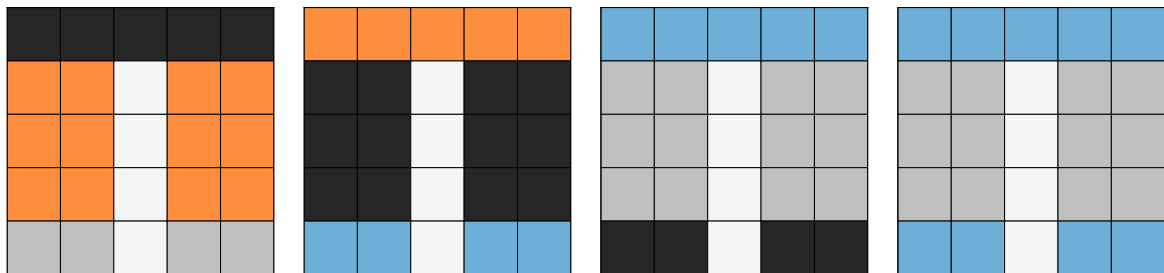


Figure 4.16: Samples with different voxel types

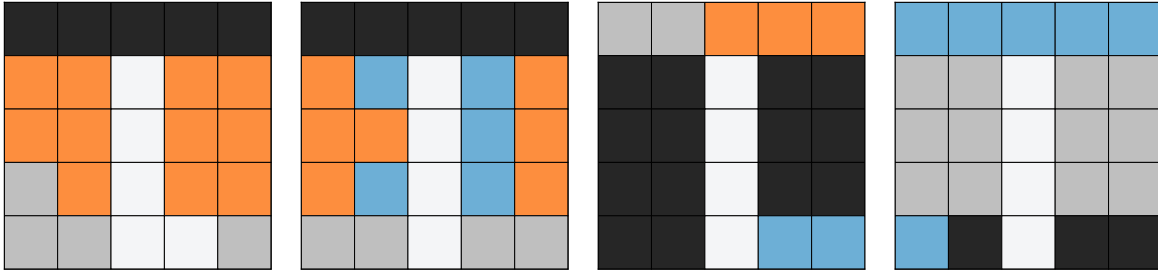


Figure 4.17: Noisy samples with different voxel types

The model also generates samples that have a completely new shape and share no immediate resemblance to the original robot, yet seem structured. Four out of these 15 structures are shown in Figure 4.18.

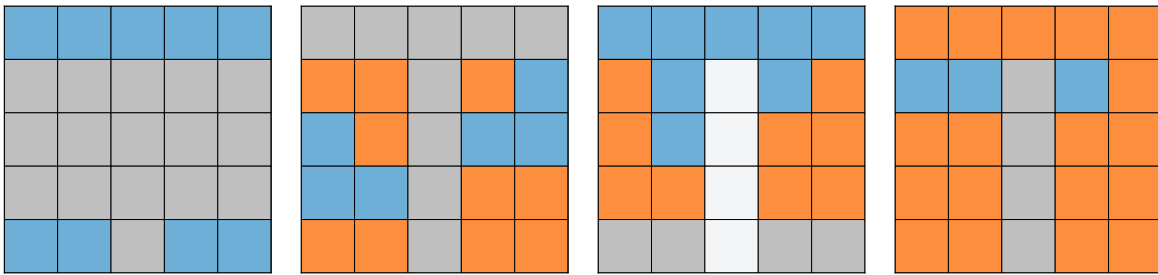


Figure 4.18: Samples with new characteristics

Lastly there are some outputs (9/100), that seem to be very noisy, such that they cannot be assigned to a specific category, examples of which are depicted in Figure 4.19.

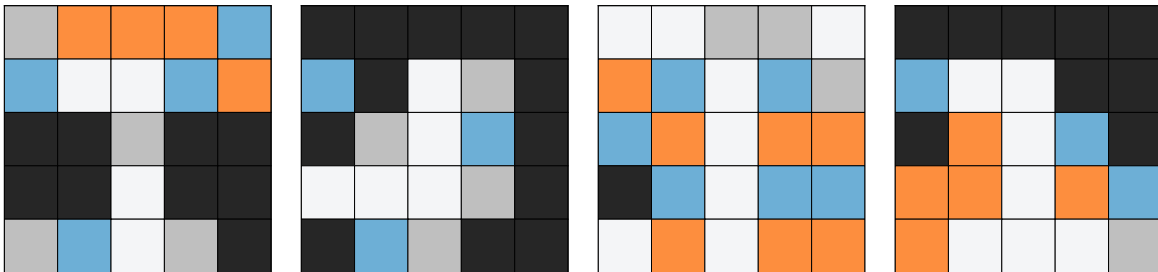


Figure 4.19: Samples that appear to be random noise

All in all, it appears that with the model architecture described in Section 4.3, overfitting is not a concern, as the generated samples are still diverse enough, especially considering that the diversity will increase when trained on more survivors. Essentially, choosing the target MSE controls how explorative the strategy should be. Lower MSE during training means the sampled structures are more similar to the survivors. For the following experiments, the model architecture and hyperparameters of this section are used; for a less exploratory strategy, the model architecture would have to be changed to try and fit the data more accurately, for example by adding more self-attention layers or compressing the data even further to a 2x2 structure before up-sampling again.

Chapter 5

Experiments

5.1 Walker-v0

The first task the generational diffusion algorithm is evaluated on, is the Walker-v0 environment. The walking task is classified as easy and is solved by all benchmark algorithms. The terrain is a flat surface made out of rigid voxels. The length of the course is 99 voxels, and the episode length is 500 timesteps. All benchmark algorithms were able to cross the entire course. [Bha+21]

The results are plotted in Figure 5.1.

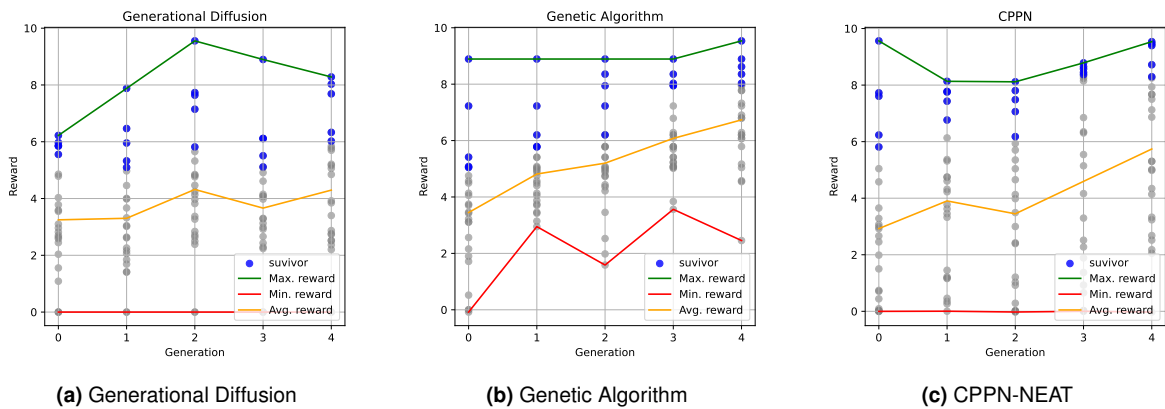


Figure 5.1: Evaluations on Walker-v0

The population sizes for all three algorithms were 25, the survivors for the Generational Diffusion Algorithm (GDA) and Genetic Algorithm (GA) were 5 each. The diffusion model trained for 250 epochs.

The generational diffusion algorithm completes the task in only two generations. After the second generation, the maximum reward decreases again slightly. Furthermore, while there is one outlier in every generation, the lowest rewards of the GDA do increase. Only in the GA do the minimum, maximum and average rewards steadily increase (looking beyond the fourth generation, the CPPN also fluctuates, similar to the GDA, as shown in Figure A.5. Because of the variance of the task, running this experiment multiple times is expected to lead to different outcomes, such that even though the GDA achieved walking faster than the GA, this has only limited significance. In fact, by chance, CPPN-NEAT even achieved walking in the first generation. Thus, while this experiment is great for proving the concept of the algorithm, it gives little information on how it compares with the other benchmark algorithms. However, we can see some characteristic differences in the best-performing robots from each algorithm.

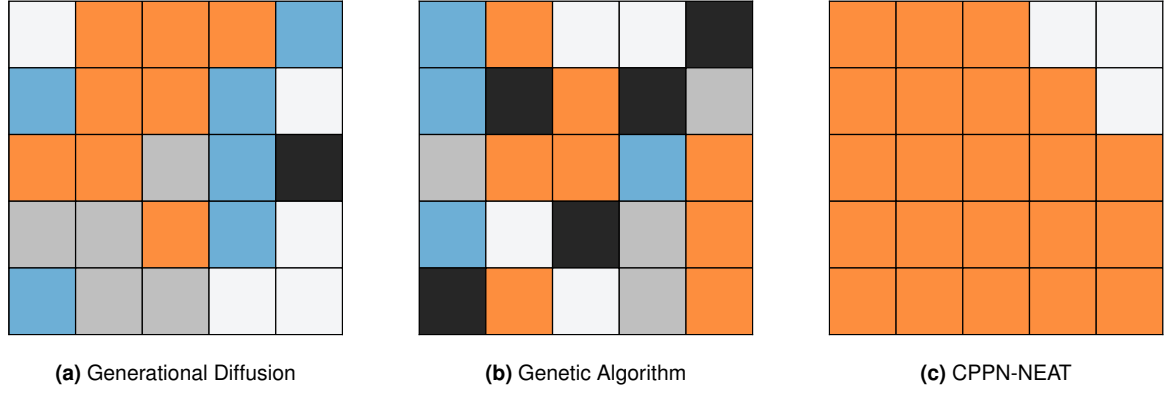


Figure 5.2: First robots to complete task

The best-performing robot of the genetic algorithm looks like random noise, while for CPPN-NEAT the robot is simply a block almost entirely made out of horizontal actuators. The diffusion model seems to find a balance between these two extremes, producing patches of the same voxel type, while still keeping a complex structure.

5.2 Simplified Reward Environment

Assessing how the generational diffusion algorithm compares to the other benchmark algorithms would require running experiments on medium and hard tasks, which would require significant computational resources and time. As discussed previously, the main bottleneck for computational resources is the control loop that maps the robots to their reward. We can simply replace this mapping with a mathematical function, that computes a reward based on the input robot structure. While this makes the problem more abstract, it enables us to run many experiments in a short amount of time, while having complete knowledge of the problem. This means we can design the reward function and track where the samples are instead of only having the reward function evaluated at the samples.

The simplest reward function is the sum of the elements of the robot’s mathematical representation. This means that a robot completely made out of vertical actuators is optimal. While this sounds trivial, this is in fact similar to the underlying reward function of many tasks, where having as many voxels of a specific voxel type is ideal. This is seemingly the case for the Walker-v0 environment, where robots with more horizontal actuators perform better, except that in the Walker-v0 task, if a certain threshold of horizontal actuators is reached, the task is simply complete. We cannot evaluate which of two robots completing the task is better.

Furthermore, completing the task with the simplified reward function is very challenging, as only one configuration achieves the optimal reward (a score of 100) and only 25 robots achieve a reward of 99 and so on, meaning that above a score of 50, better and better rewards are achieved by fewer and fewer robots.

The task is, in fact, so difficult that the genetic algorithm is not able to solve the task in 50 generations, as shown in 5.3. Similarly to the Walker-v0 environment, the CPPN-NEAT algorithm solves the task by chance, as shown in Figure 5.4. However, the algorithm produces very simple patterns, such that the performance on this task is not as insightful, especially considering the task is never completed again after the initial generation.

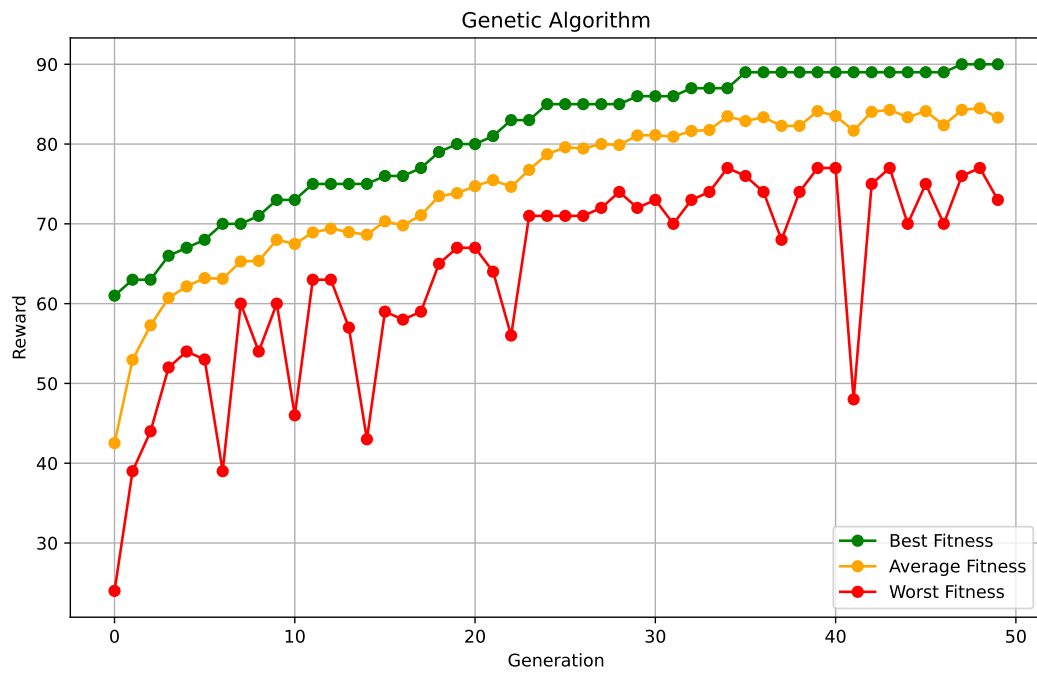


Figure 5.3: Genetic Algorithm on simple environment

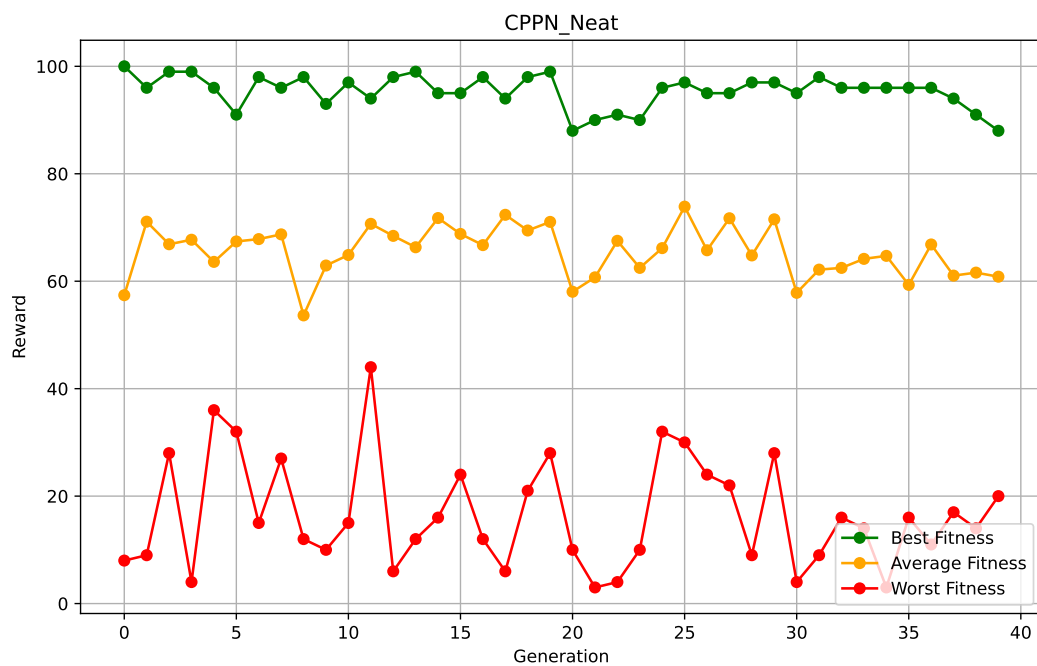


Figure 5.4: CPPN-NEAT on simple environment

The generational diffusion algorithm, run with three different epochs, shown in Figures 5.5, 5.6 and 5.7, manages to complete this task. The fewer epochs are used during the training part of the algorithm, the faster the task is solved. This confirms that training the model less leads to a more explorative algorithm, compared to overfitting, which leads to more conservative and robust updates.

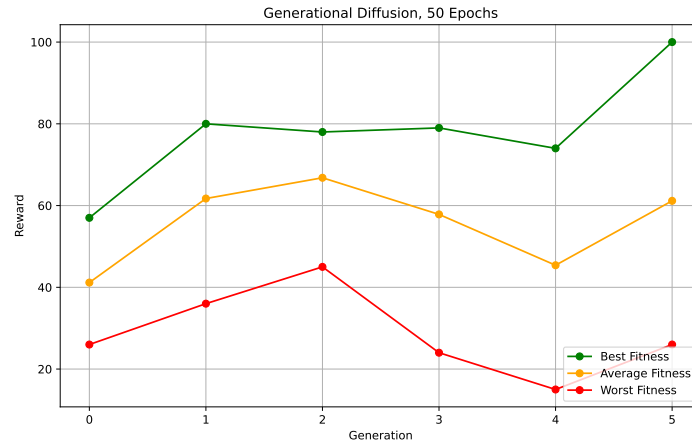


Figure 5.5: GDA on simple environment, epochs = 50

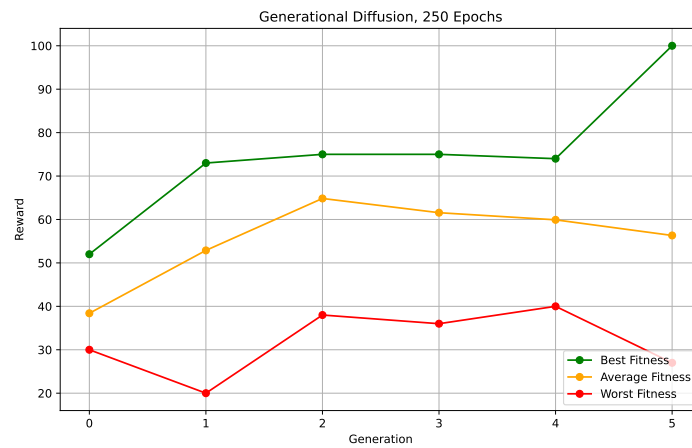


Figure 5.6: GDA on simple environment, epochs = 250

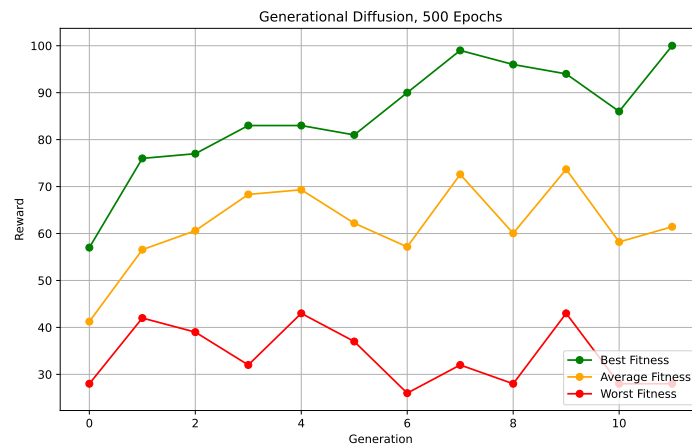


Figure 5.7: GDA on simple environment, epochs = 500

One should note that, while it is certainly plausible that reward mappings based on solving a task with the control loop lead to a similar reward distribution as with the simplified reward function, it can also very well be that the distribution is entirely different. However, the reward calculation in the simplified environment is independent from the algorithm, especially considering that with the noise-adding mapping used, as described in Section 4.2,

the vertical actuator is not mapped to an extreme of the value range. It is therefore not enough for the model to overshoot, as this would lead to robots filled with void or solid voxels, leading to invalid robots.

Looking at the robots of the different generations, it becomes clear that the algorithm learned a specific structure every generation, which slightly changes until the optimal structure is found.

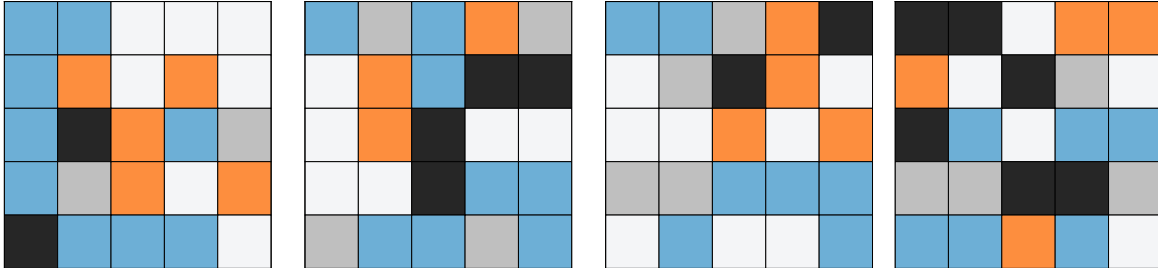


Figure 5.8: Best robots of Generation 0, GDA with 500 Epochs

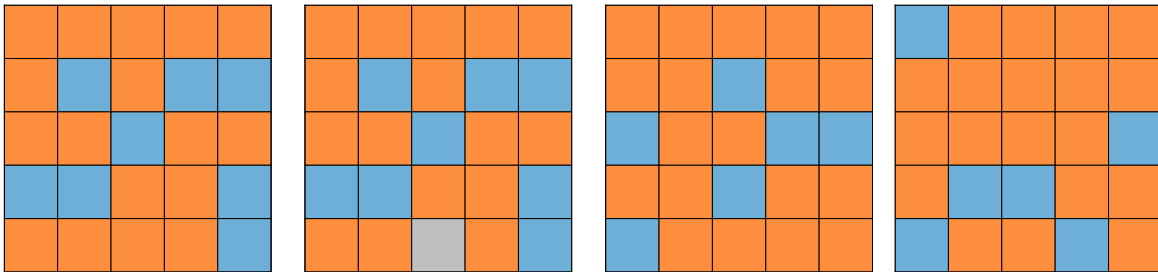


Figure 5.9: Best robots of Generation 4, GDA with 500 Epochs

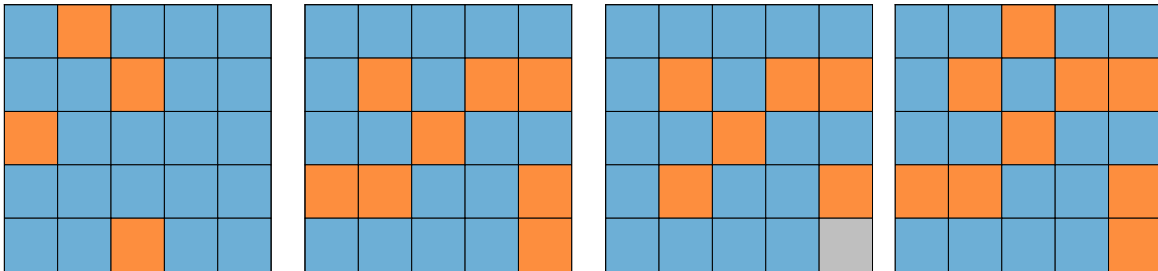


Figure 5.10: Best robots of Generation 8, GDA with 500 Epochs

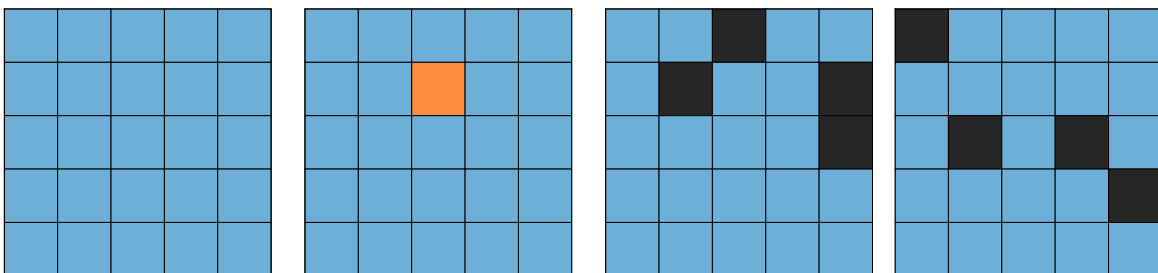


Figure 5.11: Best robots of Generation 11, GDA with 500 Epochs

Chapter 6

Positioning Diffusion Models in Design Optimization

6.1 Comparison to GA and CPPN-NEAT

As discussed in Section 5.2, in order to accurately compare the performance of the generational diffusion approach to other algorithms, more experiments are required. However, we can compare the methodology itself to find similarities and differences to the genetic algorithm and the CPPN-NEAT algorithm.

As described in Section 2.1, the general structure of the algorithms is very similar. Instead of directly mutating the survivors of the previous generation, the survivors are used to train the diffusion models, and the outputs have similar characteristics to the mutated structures of the GA (at least one of the categories of the outputs, the imperfect reconstruction, as seen in Section 5.2).

There are also similarities to the CPPN-NEAT algorithm. Both use a neural network to produce the robot structure. The difference is that the diffusion model denoises random input while CPPN queries over all voxels of the structure. Contrary to NEAT, the architecture of the diffusion model is fixed and does not change over the generations. The parameters of the diffusion model are also not randomly perturbed but learned. One might suggest having an agile network structure, combining diffusion models and NEAT. However, the mutations needed to arrive at even a simple diffusion model architecture would take too many mutations, as NEAT starts with the simplest network topology. Also, tuning the parameters by only perturbing every generation instead of using an optimizer is most likely infeasible. This also shows the limitations of CPPN-NEAT, as some tasks need a very complex neural network to produce a robot capable of solving them.

The generational diffusion method combines elements of both algorithms. This is one possible explanation as to why the outputs seem to strike a balance between the complexity of the robots produced by the GA and the coordinated structure of CPPN-NEAT.

6.2 Weaknesses of Diffusion Models

The method does have some weaknesses, which shall be discussed in the following.

One potential problem is the accuracy at the edges of the robots, arising from the padding used in the convolutional operations. Padding artificially introduces voxels around the structure, which can skew the results, especially at the boundaries. While this effect may be negligible in images with thousands of pixels, its impact becomes significant in this specific use case. For instance, during the down-sampling process, eight out of the nine voxels in a

3x3 convolution are influenced by synthetic data generated from padding, while only the center voxel is derived entirely from actual data. This effect is visualized in Figure 4.9. Similarly, with so few voxels, the information neighboring voxels can infer is limited when determining the type of a voxel.

These two problems are automatically solved when dealing with more complex structures, i.e., hundreds and thousands of cells. Higher dimensional data also allow for more complex network architectures, with multiple compression and decompression layers, and larger kernels, leading to a better understanding of the data distribution.

Another weakness of using a diffusion model for determining cell types is this unnatural ordering of cells. In Section 4.2 an ordering of cells that seems logical was chosen. It is, however, entirely arbitrary. Especially considering that the diffusion model does not work with discrete values, and the final structure is only retrieved by rounding and clipping the data. Thus the diffusion model would perform better if the cell types were continuous. For instance, instead of having the voxel types soft and hard, the transition could be described by a continuous parameter (such as the stiffness). Also, instead of having two discrete directions for actuators (horizontal and vertical) this transition could be characterized by the angle of the direction of actuation. Thus, all voxels could be described by two parameters: the stiffness and the angle of actuation. In this case, all voxels would be actuated, but alternatively there could be a third parameter describing how much influence the actuation has versus the natural deformation. Working with continuous design parameters would thus make the diffusion model a more natural fit, as its operations are also defined on continuous data.

Lastly, the reward is not directly linked to the diffusion model. Therefore, the diffusion model has no incentive to produce better robots. Instead the method relies on randomness and natural selection to increase the rewards of the robots.

6.3 Possible Improvements for Future Work

Incorporating the reward directly in the training would require differentiable design parameters and a differentiable simulation. Then, instead of training a likelihood-based diffusion model, an energy-based diffusion model, introduced by Song et al. [Son+21], could be used. This has the benefit that the entire reward distribution is learned directly, making the intermediate step of training the model on the survivors of the previous generation obsolete. The problem when training directly on the reward, is that the reward has to be computed for every structure during training, compared to the generational method where the mean square error is computed. The computational costs could make this impossible, depending on the evaluation time, but in any case will take longer than computing the MSE.

An alternative solution to incorporating the reward in the diffusion model and learning the entire distribution at once, suitable for a non-differentiable environment, would be to use a heuristic gradient on the model parameters, or, in the simplest case, use a version of the probabilistic hill climber algorithm [Gre96]. However, the problematic computation time would remain.

In the generational diffusion method, the number of epochs used during training is a hyperparameter that indirectly controls the mean square error reached during training. This determines how explorative the algorithm is. One possible improvement could be to make this relationship explicit. Instead of fixing the number of epochs, the model could learn until the target MSE is reached. The target MSE could change depending on how the robots are performing, for example, if the robots' rewards have stagnated over the last couple of generations, the "policy" could be tuned to be more explorative, i.e., aiming for a less restrictive MSE.

Another area of possible improvement is sampling; in the current implementation the diffusion model simply samples robots until enough unique, valid robots are found. Alternatively one could sample many times and then take the samples that were produced the most, as these samples better represent the learned data distribution. Another possibility is to filter what samples are used. For example if the next generation should be made out of 30% noisy reconstructions, 30% of a similar shape but different voxel types and 40% novel shapes (see Section 4.5 for the different categories). This classification of robots could be done automatically by computing characteristic attributes of the structures. For example, when flattening the matrix into a vector, the minimum Euclidean distance to the survivors of the previous generation can indicate if the robot is a noisy reconstruction of one of them.

In general, the diffusion model can be made more efficient. The work of Alex Nichol and Prafulla Dhariwal shows, for example, that a cosine noise schedule instead of a linear one could lead to better performance [ND21].

6.4 Advantages of Diffusion Models

Diffusion models also bring significant advantages, and the most important factors are discussed in this section.

As with any generational approach, it is very easy to introduce prior knowledge into the method, simply by creating the starting generation with this knowledge. Diffusion models make building on previous knowledge even easier, as the model can simply be trained on many different structures, each embedded with possibly unique information. The great advantage is that with the generational diffusion method, this knowledge transfer can be done without the need for the computationally intensive control loop. (The training and sampling times of the diffusion model are negligible compared to the evaluation of robots in the inner loop.) As the diffusion model learns the distribution of its inputs, it enables building on the work of different algorithms and combining the strengths of their samples, illustrated in Figure 6.1.

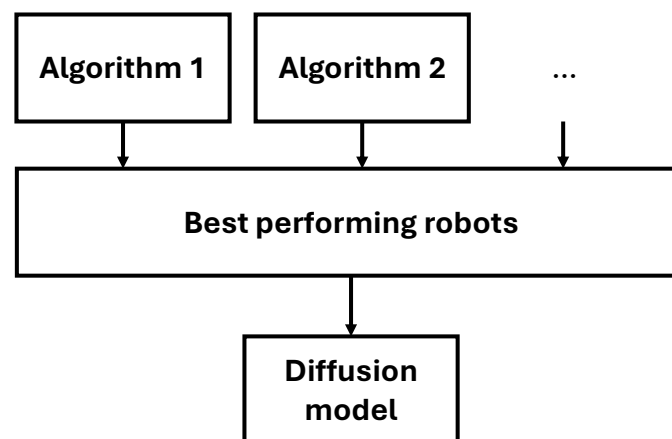


Figure 6.1: Building on prior knowledge with diffusion models

It is also possible to embed the diffusion model in a process together with other algorithms. For example, first finding local maxima with the genetic algorithm, then using a diffusion model to merge these structures into robots with similar fitness, but not necessarily located at local maxima, then using GA to find the local maxima from there, and so on.

Another advantage of diffusion models in design optimization is that it has a meaningful

and natural crossover operation. This crossover is simply the interpolation between the two robots. Interpolation in diffusion models means that a model that is trained on data similar to (or containing) the two robots, adds noise to both ($t < T_{max}$), then interpolates between the two noisy structures and denoises the result [HJA20]. This can be useful, for example, when an algorithm has optimized robots for two separate tasks. Then, the diffusion model can be trained on robots from both tasks, and the interpolation between the best robot of each task leads to a robot that has elements of both, and thus, is likely able to complete both tasks. This idea can be expanded by splitting up complex tasks into small manageable tasks and then merging the characteristics of the robots with a diffusion model, i.e., divide and conquer.

Lastly, diffusion models pose a highly scalable solution for design optimization. Section 6.2 describes how more cells should lead to better results. This is quite the contrary to many other algorithms, for example the genetic algorithms will take much longer to find the local maximum for a robot with more cells, when only individual cells are mutated.

Diffusion models are also highly suitable for parallelization. In the case of an energy-based diffusion model with a reward-based loss function, this is obvious, but even for the presented generational diffusion algorithm, this is the case. Here, a larger batch size leads to a more accurate evaluation of the performance of the diffusion model and makes it possible to increase the number of survivors, which in turn enables a better understanding of the underlying data distribution.

6.5 Conclusion and Outlook

In summary, this thesis describes the function of co-design optimization methods, applied within the Evolution Gym framework, a large-scale benchmark for evolving the structure and control of soft robots. The algorithms use a bi-level optimization structure, where the design optimization occurs in the outer loop and the control optimization in the inner loop.

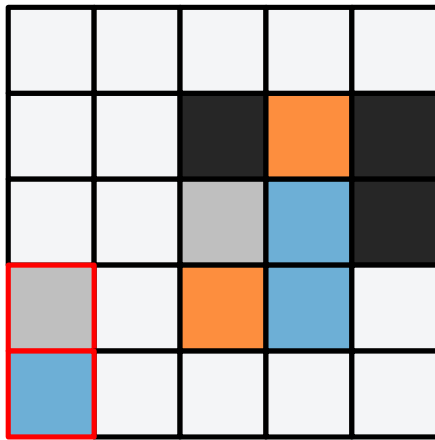
The advantages and shortcomings of the prevalent design algorithms, the genetic algorithm (GA), and Compositional Pattern Producing Networks with NeuroEvolution of Augmenting Topologies (CPPN-NEAT), are elaborated. To overcome their weaknesses, a novel design optimization method using diffusion models is introduced.

The generational diffusion algorithm efficiently combines the strength of diffusion models to learn complex patterns, with the evolutionary principle of natural selection, to find the optimal design of a robot. The results of this method are compared to GA and CPPN-NEAT in terms of their performance and topology of the generated robot structures. Robots are evaluated in the inner loop, where a control optimization process is executed using a reinforcement learning algorithm in a physics simulation to assess their performance. However, these simulations are computationally expensive. As a result, this thesis uses a single environment, Walker-v0, to compare the algorithms. To address the computational challenges posed by the physics simulation, we also developed a conceptual inner loop mechanism, where the performance of the robots is estimated using simple mathematical equations instead of resource-intensive simulations. This approach allowed us to test the proposed diffusion model and benchmark algorithms more efficiently.

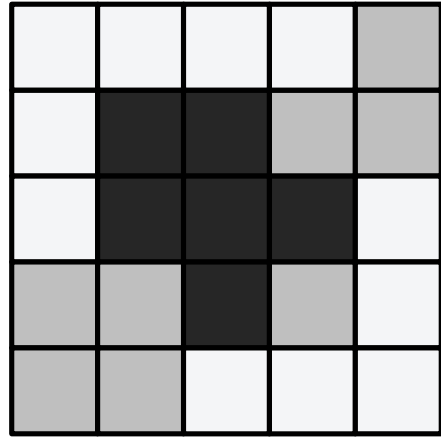
Utilizing diffusion models in more complex design environments has great potential due to the inherent scalability of the method. Thus, future work involves both testing the algorithm on other tasks of Evogym[Bha+21], and applying it to other benchmarks with a higher resolution. Furthermore, the weaknesses of the current method open up possibilities for continued research.

Appendix A

Appendix 1



(a) Robot is not fully connected (disjoint part outlined in red)



(b) No actuator present

Figure A.1: Examples of invalid robots

Parameter	Value
T	128
N	4
K	4
M	4
T_{max}	10^6
ϵ	0.1
c_1	0.5
c_2	0.01

Table A.1: PPO Hyperparameters used in Evogym[Bha+21]

T_{max}	μ	σ^2
100	-0.3099	0.3972
200	-0.2407	0.6373
300	-0.1875	0.7802
400	-0.1446	0.8666
500	-0.1115	0.9198
600	-0.0876	0.9536

Table A.2: Mean and Variance for $\beta_{end} = 0.01$

T_{max}	μ	σ^2
100	-0.2742	0.5316
200	-0.1874	0.7798
300	-0.1279	0.8978
400	-0.0882	0.9510
500	-0.0604	0.9788
600	-0.0414	0.9881

Table A.3: Mean and Variance for $\beta_{end} = 0.015$

T_{max}	μ	σ^2
100	-0.2416	0.6358
200	-0.1457	0.8673
300	-0.0870	0.9523
400	-0.0532	0.9817
500	-0.0323	0.9946
600	-0.0185	0.9987

Table A.4: Mean and Variance for $\beta_{end} = 0.02$

T_{max}	μ	σ^2
100	-0.1865	0.7821
200	-0.0878	0.9524
300	-0.0419	0.9891
400	-0.0189	0.9987
500	-0.0093	0.9980
600	-0.0031	0.9997

Table A.5: Mean and Variance for $\beta_{end} = 0.03$

T_{max}	μ	σ^2
100	-0.1113	0.9212
200	-0.0314	0.9931
300	-0.0094	0.9996
400	-0.0020	0.9991
500	-0.0018	0.9994
600	-0.0004	1.0007

Table A.6: Mean and Variance for $\beta_{end} = 0.05$

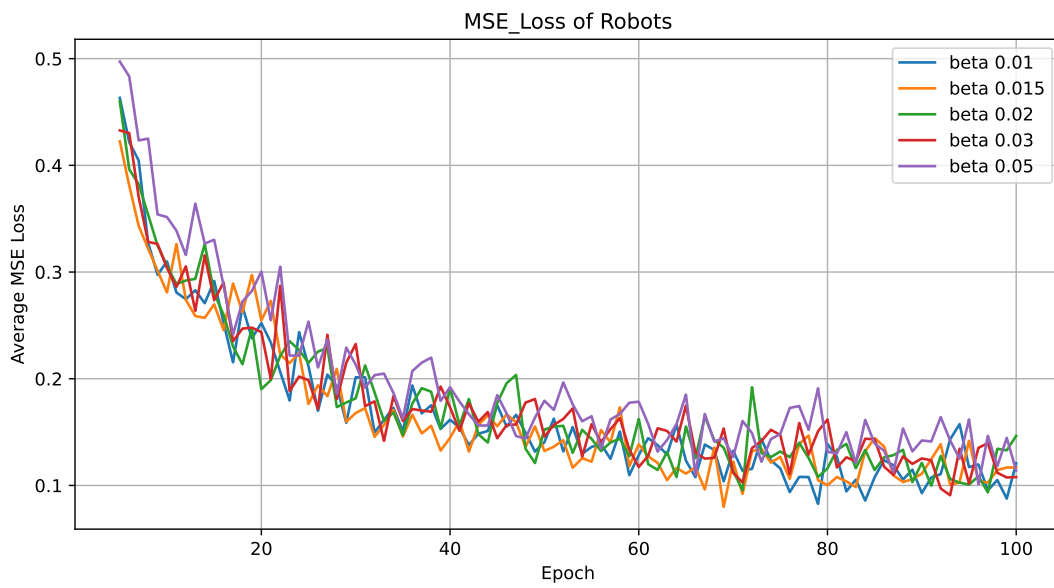


Figure A.2: MSE loss for different β_{end} , from epoch 5 to 100

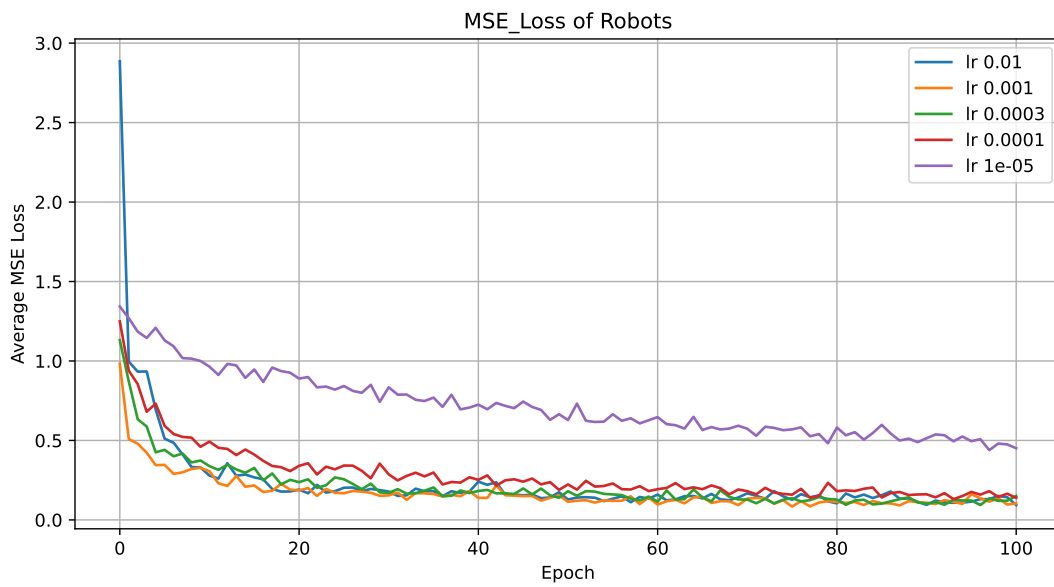


Figure A.3: MSE loss for different learning rates l_r , from epoch 1 to 100

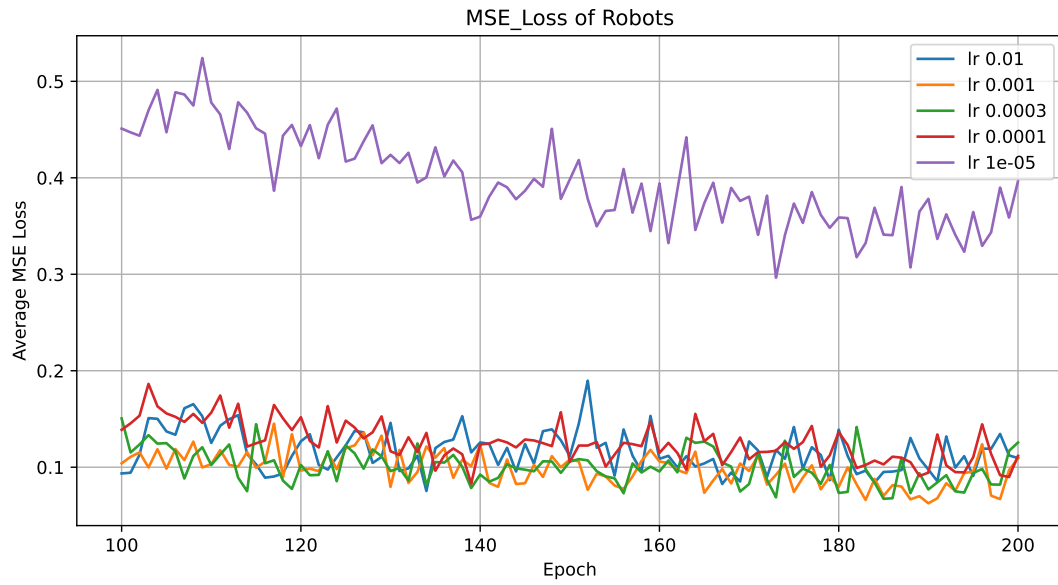


Figure A.4: MSE loss for different learning rates l_r , from epoch 100 to 200

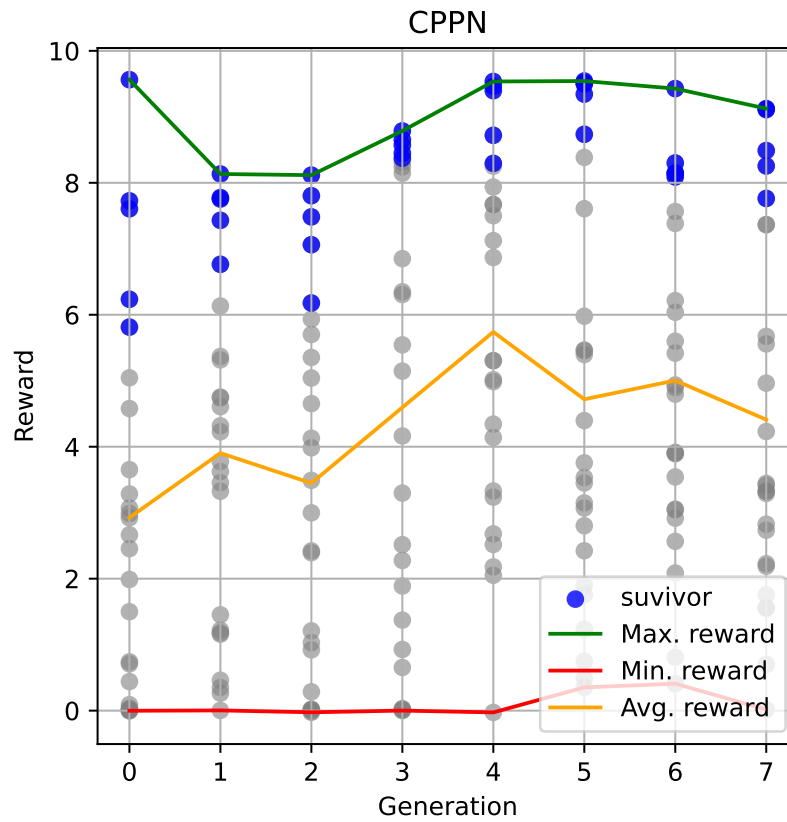


Figure A.5: CPPN evaluations on Walker-v0

Bibliography

- [Ahm23] Ahmadi, R. *An implementation of Diffusion Models in PyTorch*. <https://github.com/rahaahmadi/Diffusion-Model-Scratch>. Accessed: August 07, 2024. 2023.
- [Bha+21] Bhatia, J., Jackson, H., Tian, Y., Xu, J., and Matusik, W. “Evolution gym: A large-scale benchmark for evolving soft robots”. In: *Advances in Neural Information Processing Systems* 34 (2021).
- [Che+14] Cheney, N., MacCurdy, R., Clune, J., and Lipson, H. “Unshackling evolution: evolving soft robots with multiple materials and a powerful generative encoding”. In: *ACM SIGEVOlution* 7.1 (2014), pp. 11–23.
- [Gre96] Greiner, R. “PALO: a probabilistic hill-climbing algorithm”. In: *Artificial Intelligence* 84.1 (1996), pp. 177–208. ISSN: 0004-3702. DOI: [https://doi.org/10.1016/0004-3702\(95\)00040-2](https://doi.org/10.1016/0004-3702(95)00040-2). URL: <https://www.sciencedirect.com/science/article/pii/0004370295000402>.
- [HJA20] Ho, J., Jain, A., and Abbeel, P. “Denoising Diffusion Probabilistic Models”. In: *Advances in Neural Information Processing Systems*. Ed. by Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. Vol. 33. Curran Associates, Inc., 2020, pp. 6840–6851. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/4c5bcfec8584af0d967f1ab10179ca4b-Paper.pdf.
- [KCK21] Katoch, S., Chauhan, S. S., and Kumar, V. “A review on genetic algorithm: past, present, and future”. In: *Multimedia tools and applications* 80 (2021), pp. 8091–8126.
- [Kim+21] Kim, D., Kim, S.-H., Kim, T., Kang, B. B., Lee, M., Park, W., Ku, S., Kim, D., Kwon, J., Lee, H., Bae, J., Park, Y.-L., Cho, K.-J., and Jo, S. “Review of machine learning methods in soft robotics”. In: *PLOS ONE* 16.2 (Feb. 2021), pp. 1–24. DOI: 10.1371/journal.pone.0246102. URL: <https://doi.org/10.1371/journal.pone.0246102>.
- [KB17] Kingma, D. P. and Ba, J. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: <https://arxiv.org/abs/1412.6980>.
- [LBH15] LeCun, Y., Bengio, Y., and Hinton, G. “Deep learning”. In: *Nature* 521.7553 (2015), pp. 436–444. ISSN: 1476-4687. DOI: 10.1038/nature14539. URL: <https://doi.org/10.1038/nature14539>.
- [Lip14] Lipson, H. “Challenges and opportunities for design, simulation, and fabrication of soft robots”. In: *Soft Robotics* 1.1 (2014), pp. 21–27.
- [Luo22] Luo, C. *Understanding Diffusion Models: A Unified Perspective*. 2022. arXiv: 2208.11970 [cs.LG]. URL: <https://arxiv.org/abs/2208.11970>.
- [McI+] McIntyre, A., Kallada, M., Miguel, C. G., Feher de Silva, C., and Netto, M. L. *neat-python*.
- [ND21] Nichol, A. and Dhariwal, P. *Improved Denoising Diffusion Probabilistic Models*. 2021. arXiv: 2102.09672 [cs.LG]. URL: <https://arxiv.org/abs/2102.09672>.

- [Pas+19] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.
- [Raf+21] Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. “Stable-Baselines3: Reliable Reinforcement Learning Implementations”. In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: <http://jmlr.org/papers/v22/20-1364.html>.
- [Sch15] Schulman, J. “Trust Region Policy Optimization”. In: *arXiv preprint arXiv:1502.05477* (2015).
- [Sch+17] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG]. URL: <https://arxiv.org/abs/1707.06347>.
- [Sil15] Silver, D. *Lectures on Reinforcement Learning*. URL: <https://www.davidsilver.uk/teaching/>. 2015.
- [Smi+22] Smith, L., Hainsworth, T., Haimes, J., and MacCurdy, R. “Automated Synthesis of Bending Pneumatic Soft Actuators”. In: *2022 IEEE 5th International Conference on Soft Robotics (RoboSoft)*. 2022, pp. 358–363. DOI: 10.1109/RoboSoft54090.2022.9762105.
- [Son+21] Song, Y., Sohl-Dickstein, J., Kingma, D. P., Kumar, A., Ermon, S., and Poole, B. *Score-Based Generative Modeling through Stochastic Differential Equations*. 2021. arXiv: 2011.13456 [cs.LG]. URL: <https://arxiv.org/abs/2011.13456>.
- [Sta07] Stanley, K. O. “Compositional pattern producing networks: A novel abstraction of development”. In: *Genetic programming and evolvable machines* 8 (2007), pp. 131–162.
- [SM02] Stanley, K. O. and Miikkulainen, R. “Evolving Neural Networks through Augmenting Topologies”. In: *Evolutionary Computation* 10.2 (2002), pp. 99–127. DOI: 10.1162/106365602320169811.
- [Sut18] Sutton, R. S. “Reinforcement learning: An introduction”. In: *A Bradford Book* (2018).
- [Sut+99] Sutton, R. S., McAllester, D., Singh, S., and Mansour, Y. “Policy Gradient Methods for Reinforcement Learning with Function Approximation”. In: *Advances in Neural Information Processing Systems*. Ed. by Solla, S., Leen, T., and Müller, K. Vol. 12. MIT Press, 1999. URL: https://proceedings.neurips.cc/paper_files/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf.
- [Vas17] Vaswani, A. “Attention is all you need”. In: *Advances in Neural Information Processing Systems* (2017).