Preprint: Publication to appear at the International Conference on Robotics and Automation (ICRA) 2022.

Modular Robot Design Optimization with Generative Adversarial Networks

Jiaheng Hu, Julian Whitman, Matthew Travers and Howie Choset

***Abstract*— Modular robots are made up of a set of components which can be configured and reconfigured to form customized robots for a wide range of tasks. Fully utilizing the flexibility of modular robots is challenging, as it requires the identification of optimal modular designs for each given task, often with limited computation and time. Previous works in design automation achieve efficient run-times by utilizing machine learning to create a one-to-one mapping from task to design. However, the problem of robot design is often multimodal, where multiple distinct designs can be similarly or equally good for a task. Alternative design solutions may be needed in the field, for instance, if a module in the optimal design fails and no replacement is available. This paper presents a novel method based on generative adversarial networks (GANs) that learns a one-to-many mapping from task to a distribution of designs. We apply our method to construct locomoting modular robots for terrains with varying obstacle heights and infill. We compare our method against the state-of-the-art, and find that our algorithm results in better solution quality, diversity, and alternatives for when the optimal design fails.**

I. Introduction

Modular robots inspire a concept where instead of bringing a family of robots into the field, one can bring a toolbox of modules which are configured, and perhaps re-configured, to provide a customized system design for a particular task. This unique benefit poses many challenges, including the need to determine a particular system design for each given task. This problem, known as design automation, is often solved through population-based optimization such as Evolutionary Algorithms (EAs), where a diverse population of candidate designs is maintained and updated in search of high-performing designs [1], [2], [3]. Unfortunately, this optimization process typically requires repeated simulation and evaluation of many different designs, and can become computational expensive in large design spaces. Though they can produce high-quality designs, these methods cannot be directly applied in scenarios like field deployment, where time or computation is limited.

Recent works [4], [5], [6] utilize machine learning (ML) to generate designs with a low computational cost at run-time. Specifically, these works employ a training phase to learn a one-to-one mapping from task to design for a range of different tasks. This mapping is then used during deployment to create new designs, whether or not the mapping was trained on a given task. While such an approach may lead to non-optimal designs, it does produce them in real-time.

The authors are with the Robotics Institute at Carnegie Mellon University, Pittsburgh, PA 15289, USA {jiahengh, jwhitman, mtravers, choset}@andrew.cmu.edu

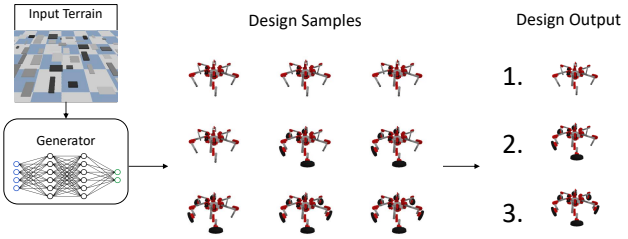

Fig. 1: We present a generative adversarial method to create a modular robot design generator. The generator takes in a task description (in this example, a terrain height map, left) and outputs a distribution of designs, where high-performing designs have high probability mass. We can then sample from the distribution (middle) and provide the user a collection of capable designs (right).

The low run-time cost of existing ML approaches make them seemingly good candidates for use in the field. However, the problem of modular robot design generation is multimodal by nature, where for any given task, there are often multiple distinct designs that are similarly or equally capable. By modelling the task-to-design mapping as a one-to-one mapping, previous methods fail to *fully describe* the relation between task and optimal designs. As a result, these methods tend to neglect a variety of the optimal or near-optimal designs. We posit that in a design automation problem, keeping track of multiple distinct solutions can be valuable, as they provide alternatives in case the "first-choice" solution fails. For instance, if a module fails unexpectedly in the field, and there are not sufficient spare modules to replace it, the user will need to quickly identify and construct a "second-choice" design. Similarly, if the first solution deployed does not work as expected, perhaps due to a simulation to reality gap, then the user will need a different design. We therefore develop an algorithm that marries the solution quality and diversity of EAs with the low run-time cost of ML.

In this work, we introduce RoboGAN, a generative approach to the problem of designing task-specific modular robots. Our method learns a mapping from task to a *distribution* of designs through a modified generative adversarial network (GAN) [7], [8], [9], and can be used to generate diverse and high quality designs in a computationally-efficient manner. We believe that maintaining a distribution of designs allows for a search that does not require additional constructs, like entropy, to force a trade-off between exploration and exploitation.

Unlike past works using GANs, where a dataset is present

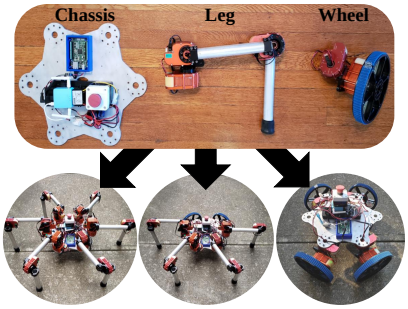

Fig. 2: A set of modules - body, legs, and wheels - can generate many different system configurations of robots.

prior to training [7], [8], [9], [10], [11], we have no data *a priori* from which a mapping from task to robot design can be distilled. Worse yet, collecting such a dataset from scratch for modular robots is enormously computationally burdensome, as obtaining each task-design pair requires solving a combinatorial optimization problem. Instead, our approach actively collects data on-line during training through a novel self-guided data creation process inspired by EAs. This data creation process looks for new promising designs around the current generated designs, promoting the generator to improve the quality and diversity of its output.

We apply our design automation approach to creating mobile robots designs (Fig. 2) specialized for traversing varying terrains. We show that our method is capable of running in real-time during testing, and outperforms competing ML methods both in terms of solution quality and solution diversity, as well as providing alternative solutions when the first design fails.

II. Related work

Evolutionary Algorithms: Population-based stochastic optimization algorithms, such as Evolutionary algorithms (EAs), have been the *de facto* choice for design automation problems due to their ability to efficiently look for solutions in a combinatorial design space [1], [2], [3], [12]. Specifically, [3] used a generative design encoding to evolve locomoting robots. Similarly, [1] used an EA to discover locomoting and swimming robots. EAs are also known for their ability of finding multiple distinct solutions due to their population-based nature, and are thus suitable for multimodal and multi-objective optimizations [13], [14]. However, since EAs require repeated evaluation of the performance of different designs, they can quickly become computationally expensive in the domain of design automation, where evaluation of each design requires running a simulation with a control policy. As a result, EAs are often too time-expensive to be deployed directly in the field, limiting their applicability to scenarios where time or computation is limited.

Learning-Based Design Automation: An alternative to explicitly searching the design space during deployment is to utilize machine learning techniques to map task to designs. These methods utilize a long training phase to learn a mapping from task to design, where the learned mapping can be directly queried during deployment to generate task-specific designs. Specifically, [4], [5] cast modular robot design as a reinforcement learning (RL) problem, and use Q-learning to learn a policy that sequentially adds nodes and edges to the partial graph of designs conditioning on the task. Fit2form [6] learns an end-to-end mapping from task to 3D gripper design. Importantly, these methods have the benefit of real-time execution. However, existing methods typically pose their objective as predicting the single design that maximizes performance for the queried task, possibly due to the difficulty in learning a complex distribution over discrete designs with a neural network. By doing so, they ignore the multimodal nature of the design automation problem.

Generative Models: Instead of trying to learn a one-to-one mapping from task to design, we propose to adopt a generative approach to robot design, where we learn a mapping from task to a distribution of designs. Generative Adversarial Networks [7] have been widely adopted to learn generative models. A GAN is an implicit generative model that attempts to capture the patterns in a dataset, such that the model can be used to generate new samples as if they were drawn from the same underlying distribution as the data. A GAN consists of two components: a generator that learns a mapping from a noise vector to generated data sample, and a discriminator that learns to distinguish generated samples from real samples. These two components are typically implemented as deep neural networks and are optimized simultaneously through gradient descent. Since the two networks have competing objectives, training a GAN can be viewed as two players playing a minimax game [7]. The conditional GAN [9] was later introduced as a variant that output labeled samples by conditioning both the generator and the discriminator on a given label.

Typically, GANs work with continuous domain, such as images, due to the need for backpropagation through the generated data during training. In order to apply GANs to discrete data, Boundary Seeking GAN (BGAN) [8] trains the generator via policy gradient with the likelihood ratio estimated by the discriminator. Since a policy gradient loss doesn't need the data samples to be differentiable with respect to the generator parameters, this method makes it possible to train a GAN for discrete data such as graphs.

III. Problem Statement

Design generation requires an appropriate data structure to represent the designs. Modular robots can be represented with graphs, where nodes represent modules and edges represent connections between them. In this work, we represent our design space $\mathbb{X}$ as graphs where the connection between nodes (topology) are fixed but the node type (module type) can be altered.

For each design $\mathcal{X} \in \mathbb{X}$, we assume the existence of a control policy π_X which is used to control the robot. In this work, we obtain our control policy through a separate process before applying our method, described in Sec. IV-D. Let $\mathbb{T}$ denote the task space. Given a task $\mathcal{T} \in \mathbb{T}$, we can obtain a

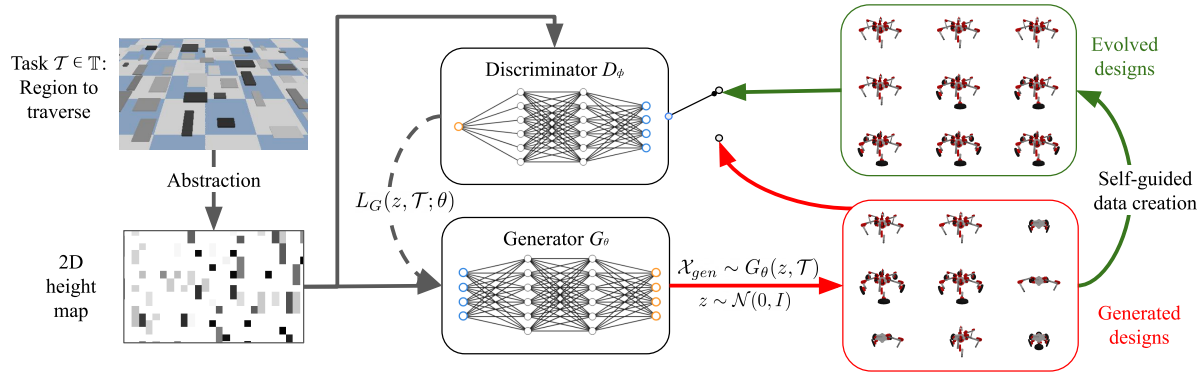

Fig. 3: We consider the task of traversing terrains with varying obstacle heights and infill. At each training iteration, a terrain is generated at random and abstracted into a 2-D height map. The generator implicitly maps this height map into a population of designs, symbolized in the red box. The self-guided data creation step explores around the generated designs by evolving them using a procedure inspired by Evolutionary Algorithms, and creates a population of evolved designs, symbolized in the green box. The discriminator takes as input the terrain and a robot design that is either from the generated designs or the evolved designs, and tries to distinguish from which population the design comes from. The output of the discriminator feeds into the loss function L_G, guiding the generator towards generating high-performing designs.

measurement of $\mathcal{X}$'s performance for task $\mathcal{T}$, $\mathcal{J}(\mathcal{X}, \mathcal{T})$, by controlling $\mathcal{X}$ in simulation with π_X.

Our goal is to learn a function that maps each task $\mathcal{T}$ to a distribution over designs $\mathbb{X}$, where designs with high performance have high probability density. We approach this problem using conditional GANs [9]. The generator $G_\theta(z, \mathcal{T})$ maps a queried task $\mathcal{T}$ and a randomly sampled latent variable $z \sim \mathcal{N}(0, I)$ to a design $\mathcal{X}$. G_θ implicitly defines a conditional distribution over designs, where we sample from this distribution by sampling latent variable z and passing it through the generator along with the task. Our objective is to optimize the parameters θ of the generator G_θ to maximize the expected performance of the generated designs X, that is,

$$\theta^\star = \arg\max_\theta \mathbb{E}_{\mathcal{T}\sim\mathbb{T}, z\sim\mathcal{N}(0,I)} \left[\mathcal{J}(G_\theta(z, \mathcal{T}), \mathcal{T})\right] \quad (1)$$

In the following section, we describe our approach to training the generative model G_θ that maximizes (1).

IV. RoboGAN

Our system is composed of three key components: a generator, a discriminator, and a novel self-guided data creator. An illustration of these components is shown in Fig. 3. The generator maps tasks to a distribution of designs. The discriminator tries to distinguish between designs generated by the generator, and designs produced by the self-guided data creator. Both the generator and the discriminator are implemented as neural networks. At every iteration, the self-guided data creator performs n steps of evolution on the robot designs generated by the generator to obtain training data. These evolution steps iteratively improve the output of the generator, pushing the generator to approximate a distribution of increasingly higher-performing candidate solutions.

A. Generator

The generator $G_\theta(z, \mathcal{T})$ takes as input a numerical encoding of the queried task $\mathcal{T}$, along with a P-dimensional vector $z \in \mathbb{R}^P$ sampled from a standard multivariate Gaussian distribution, $z \sim \mathcal{N}(0, I)$. In our experiments designing mobile robots, each task is a different terrain to traverse, so $\mathcal{T}$ is a 2D elevation map of the given terrain. $\mathcal{T}$ is first passed through two convolution layers, each with a 3×3 kernel of filter size 32. The output of the convolution layer is then flattened, passed through a full connected layer of output dimension P, concatenated with z and passed into a multi-layer perceptron (MLP) network with 3 hidden layers of size 64 to generate the output. Batch normalization [15] and ReLU activation [16] are applied at each hidden layer.

The output of the generator is a graph with m different node types and n maximum amount of nodes allowed, parameterized as a dense $m \times n$ annotation matrix $\mathcal{X}_{\text{dense}}$. Each node type corresponds to a module type, and therefore $\mathcal{X}_{\text{dense}}$ specifies the type of module for each node. Each row of the annotation matrix sums to 1 and can be viewed as a categorical distribution over all possible module selections for the given node. Thus, to generate a discrete design we obtain a one-hot matrix $\mathcal{X}$ through a categorical sampling of $\mathcal{X}_{\text{dense}}$. This sampling process is non-differentiable and prevents gradients from flowing through it. This prevents us from using the more common continuous GAN loss [7]. We adopted Boundary Seeking GAN (BGAN) [8] to extend GAN to graphs, by training the generator via policy gradient based on the likelihood ratio estimated by the discriminator. Since our generator is conditioned on the task, we developed a conditional variant of BGAN, where the generator loss is

$$\begin{aligned} &L_G(z, \mathcal{T}, \{\mathcal{X}^{(1)}, ..., \mathcal{X}^{(m)}\}; \theta) \\ &\quad = -\sum_m \frac{D(\mathcal{X}^{(m)}, \mathcal{T})}{\sum_{m'} D(\mathcal{X}^{(m')}, \mathcal{T})} P_\theta(\mathcal{X}^{(m)} | z, \mathcal{T}) \end{aligned} \quad (2)$$

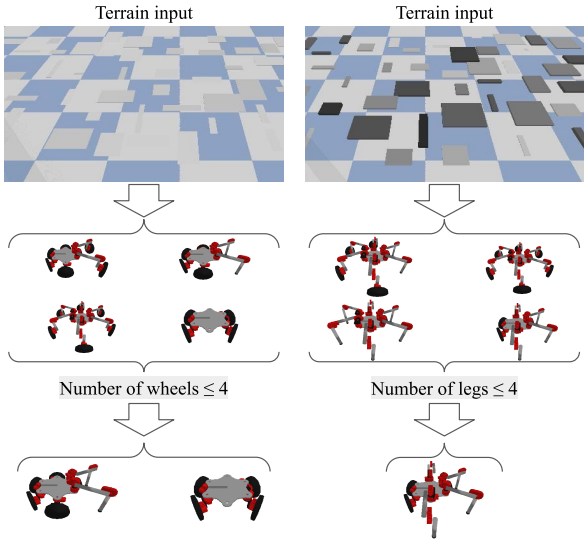

Fig. 4: From top to bottom, we show examples of queried input, top four designs outputted by the trained generator, randomly generated emergency constraints, and the designs filtered by the added constraints for a near-flat terrain (left) and a rough terrain (right). The emergency constraints restricts the maximum number of a certain type of modules, and are added to simulate unexpected module breaks or a shortage during deployment. The diverse design candidates produced by our method allows us to handle these constraints more robustly compared to prior ML approaches.

where $\mathcal{T}$ is the task; z is a latent input; $P_\theta(\mathcal{X}|z,\mathcal{T})$ is the posterior probability for design $\mathcal{X}$ specified by $\mathcal{X}_{\text{dense}} = G_\theta(z,\mathcal{T})$; $\{\mathcal{X}^{(1)},...,\mathcal{X}^{(m)}\}$ are m designs sampled from $P_\theta(\mathcal{X}|z,\mathcal{T})$; and $D(\mathcal{X}^{(m)},\mathcal{T})$ is the output of the discriminator on the (design, task) pair, explained in detail in section IV-B. During training, z is sampled from a standard multivariate Gaussian distribution $q(z)$, and $\mathcal{T}$ is uniformly sampled from the task space.

B. Discriminator

The discriminator $D_\phi(\mathcal{X},\mathcal{T})$ takes in a robot design graph $\mathcal{X}$ and the task $\mathcal{T}$, and outputs a scalar $\in (0,1)$. Similar to the canonical GAN, this scalar can be viewed as a prediction of whether the design is generated by the generator. The discriminator has the same network structure as the generator except for the input and output layer, which are adjusted to correspond to the input and output dimensions. The training data for the discriminator comes from two sources. Half of the designs are synthesised by the generator, and are labeled 0 ("fake"). The other half are obtained from the self-guided data creation step (explain in detail in section IV-C) and are labeled 1 ("real").

The discriminator is trained to minimize the cross entropy loss between predicted and actual design labels:

$$L_D(\mathcal{X}_{pop},\mathcal{X}_{gen},\mathcal{T};\phi) = -[\log D_\phi(\mathcal{X}_{pop},\mathcal{T}) + \log(1 - D_\phi(\mathcal{X}_{gen},\mathcal{T}))] \quad (3)$$

where $\mathcal{X}_{pop}$ is a design obtained from the self-guided data creation step, and $\mathcal{X}_{gen}$ is a design from the generator.

C. Self-Guided Data Creation

Our method is fundamentally different from a canonical GAN since we do not have a dataset collected *a priori*. Instead, we generate training data online through a novel self-guided process, inspired by evolutionary algorithms, which iteratively pushes the generator towards generating designs with higher performance.

During each training iteration, we pass a task $\mathcal{T}$ and a batch of randomly sampled latent vectors z through the generator to obtain a batch of designs. These designs are then treated as if they are the population of an EA, and iterate through n EA steps (e.g. mutations, cross-over, evaluation, and elite selection) to create an evolved population. The evolved population has in expectation a higher mean performance than those generated by the current iteration of the generator, and are passed into the discriminator in place of what would be considered the "real" data in a conventional GAN. By training the GAN with the evolved samples, we effectively guide the generator to model a task-conditioned distribution that is iteratively shifted towards high-performing regions in the solution space.

The generator and discriminator are both conditioned on the task description. Without this task-conditioning, the training procedure would be similar to a standard EA, wherein a population of designs are evolved for a single task. By conditioning the generator and the discriminator on the tasks, and randomly sampling tasks at each iteration, the GAN learns to interpolate between different tasks, which is the key to how our approach is able to generalize to unseen tasks during deployment.

Any variety of population-based optimization algorithm could be used inside the data creation process. However, the specific algorithm chosen will affect the convergence behavior of the generator. We experimented with two versions of EA: a classical genetic algorithm [17] and a deterministic crowding genetic algorithm [18]. The crowding GA explicitly encourages diversity of the population by enforcing a pair-wise replacement between parents and children. In both cases, we used uniform crossover with a crossover probability of 0.5 and a mutation rate of 0.1.

D. Control Policy

In order to evaluate the performance of each design, we need a policy for each design in the design space. In other words, the performance $\mathcal{J}$ is obtained by simulating a design at a task given a policy. We adopted a learned modular policy trained through deep reinforcement learning to control the generated designs. The implementation and training of the controller follows [19], and is completed before RoboGAN starts training. Note that the methods of this paper could be applied regardless of the type of controller used, since the performance evaluation of a design includes both physical and control parameters. We investigate the effect of the controller in Sec. V-D.

	Comparison Metrics		
Methods	Max distance travelled (m)↑	# of distinct solutions ↑	Max dist. travelled (Emergency constraints) (m)↑
Fit2form [6]	4.03 ± 0.91	1.2 ± 0.2	2.55 ± 1.31
Q-learning [5]	4.31 ± 0.26	2.7 ± 0.9	3.10 ± 0.92
MolGAN [10]	3.82 ± 0.55	1.8 ± 0.3	2.93 ± 1.00
Random Sampling	0.72 ± 0.64	–	0.63 ± 0.69
RoboGAN (ours)	5.10 ± 0.42	2.2 ± 0.7	3.77 ± 1.18
RoboGAN-crowding (ours)	$\mathbf{5.73 \pm 0.22}$	$\mathbf{5.8 \pm 1.5}$	$\mathbf{5.36 \pm 0.52}$

TABLE I: Performance comparison between real-time algorithms. ↑ means higher is better. Compare to other algorithms with constant test time, our approach produce designs that are superior both in performance and solution diversity, and as a result handles emergency constraints much better.

V. Experiments

We evaluated our method by designing locomoting robots for different terrains. We generate random terrains by adding blocks of randomly selected height, width, and spacing to flat ground. Examples of such terrain can be seen in the first row of Figure. 4. The design space consists of robots composed from four different types of modules: leg, wheel, leg-wheel, and chassis, where each of these modules corresponds to physical hardware produced by Hebi Robotics [20] (Fig. 2). A "none" module type is also allowed, so that designs may have fewer than the maximum allowable number of modules. Each robot contains one chassis, and each chassis has six ports where modules can be attached. Therefore, there is a total of $4^6 = 4096$ different designs in our design space. We defined the performance of each design on a given terrain as the distance travelled within a fixed amount of time, and obtain the performance by running simulations in Pybullet [21]. We evaluate our method by querying them with randomly sampled terrains and simulating the output designs. Some of the generated designs can be seen in Figure 4. Our supplementary video showcases the locomotion of some of the generated designs in simulation. We conducted all training and testing on a desktop computer with Ubuntu 18.04, Intel i7 eight-core processor at 1.9240GHz, and an NVIDIA GTX 1070 graphics card.

A. Comparison Metrics

We compare our algorithm to a variety of related methods using a set of metrics for the quality and diversity of the output solutions. To measure the quality of the generated designs, we record the performance of the best robot design found by each algorithm, and report it as *max distance travelled*. We then quantify the diversity of the generated solutions following the benchmark proposed in [22] to obtain the *number of distinct solutions*. Specifically, given a set of candidate solutions produced by an algorithm, a candidate solution counts as a distinct solution if it is at least δ distance away from any existing distinct solution, and its performance is within ϵ distance from the population's highest performance. We used $\delta = 1$ and $\epsilon = 0.5$ in our experiment.

To quantitatively examine the benefit of having solution diversity, we introduce the concept of "emergency constraints" into our experiment. Emergency constraints take the form of limitations on the maximum number of each type of modules a design can contain. These constraints model a scenario where a module unexpectedly broke, or a module shortage is discovered during the prototyping process, and these constraints are generated at random during testing. If an algorithm fails to generate any design that satisfies the constraints, a random design that satisfies the constraints is used. We record this quantity as *max distance travelled with emergency constraints*. For evolutionary algorithms, these emergency constraints can be bypassed by rerunning the search process from scratch while taking these constraints into account. Therefore, we only report this quantity for real-time algorithms.

Lastly, when comparing against evolutionary algorithms, we included *average runtime*, which measures the wall time of each algorithm during execution in minutes, to demonstrate the computational efficiency of our algorithm.

B. Comparison with other real-time algorithms

We first compare against learning-based and rule-based approaches. As with our algorithm, these methods can run in near real-time after a longer training procedure. While the related ML algorithms are all trained to generate a single design, their trained model can often be queried stochastically to obtain multiple designs, albeit without any guarantee that designs obtained in this manner are near-optimal. All learning-based algorithms are trained for 12 hours on i.i.d. sampled tasks. Specifically, we compare against:

Fit2form: Our implementation of Fit2form follows [6], where a generator is trained to map task to a single design by maximizing the expected performance of the design. Unlike in the original work, the modular robot we are trying to design in our experiment is not differentiable. Therefore, instead of back-propagating directly with respect to the expected performance, we used a policy gradient loss.

Q-learning: Our implementation of Q-learning follows [5], where a policy sequentially adds nodes and edges to the partial graph of designs. A value function, implemented as neural network, is learned to serve as a search heuristic.

MolGAN: MolGAN [10] was originally introduced to optimize molecule structure given a labelled dataset. This approach involves using a GAN to keep diversity while using reinforcement learning to promote performance. We implement and evaluate a conditional variant of MolGAN for modular robot design. Since we do not have a dataset required for the MolGAN training, we randomly sample valid robot designs from the design space to train the GAN.

	Comparison Metrics		
Methods	Max distance travelled (m) ↑	# of distinct solutions ↑	Avg. runtime (min.) ↓
GA-20 [17]	4.21 ± 1.43	1.8 ± 0.8	25.4 ± 0.7
GA-50 [17]	5.63 ± 0.34	2.0 ± 1.4	71.9 ± 1.3
GA-crowding [18]	5.33 ± 0.25	$\mathbf{6.4 \pm 1.3}$	97.6 ± 2.1
RoboGAN (ours)	5.10 ± 0.42	2.2 ± 0.7	$\mathbf{0.003}$
RoboGAN-crowding (ours)	$\mathbf{5.73 \pm 0.22}$	5.8 ± 1.5	$\mathbf{0.003}$

TABLE II: Performance comparison between tested algorithms. ↓ means lower is better, ↑ means higher is better. Compare to evolutionary algorithms, our method can produce designs with on-par quality with significantly faster runtime.

Random Sampling: As a simple baseline, we also examined the result of randomly sampling in the design space. Specifically, a random design is generated by sampling module types for each node of the design graph. For each test terrain, we sample 10 i.i.d. designs, and record the one with the highest performance.

RoboGAN: We included two versions of our algorithm, RoboGAN and RoboGAN-crowding, where the latter version explicitly encourages diversity by utilize deterministic crowding, as explained in detail in Sec. IV-C.

Results are presented in Table I. Each entry is averaged over 10 randomly sampled terrains, each with 20 independent runs. We found that both versions of our algorithm outperform the competing algorithms both in terms of design quality and design diversity. Further, when the emergency constraints are introduced, the crowding version of our algorithm outperforms all other algorithms by a significant margin, demonstrating the advantage gained from outputting multiple designs when the first-choice design fails.

C. Comparison with Evolutionary Algorithms

We also compare our algorithm against evolutionary algorithms. These algorithms are computationally expensive, but can produce high-quality designs. We compare against:

Classical genetic algorithm: Our implementation of a classical genetic algorithm follows [17], with uniform crossover and uniform mutation. We run two versions of the classical genetic algorithm with population size of 20 and 50 respectively, with a max iteration of 50, and report their results as GA-20 and GA-50 in Table I.

Crowding genetic algorithm: Crowding approaches are introduced to EAs as an effective way to generate multi-modal solutions [13]. We therefore use a deterministic crowding genetic algorithm as a benchmark to examine the capability of our algorithm to generate diverse solutions. Our implementation follows [18], using a population size of 50 and a max iteration of 50.

Results are shown in Table II. Compared to EAs, both versions of our algorithm are able to generate comparable designs more efficiently. It is important to note that our execution efficiency comes at the cost of the training time, which EAs do not require. However, the training takes place before the generator is deployed, and need only be trained once before being used for many tasks. Also note that EAs tend to find better solutions given larger population sizes and more of iterations, but their time cost scales proportionally.

D. Effect of the controller applied

We also examined how the performance of our method is affected by the controller applied to the robots. For this experiment, we replaced the neural controller in section IV-D by a hand-crafted controller, where the legs follow a cyclic alternating-tripod gait, and the wheels spin forward. We re-trained RoboGAN with this new controller, and compare the max distance travelled by generated designs with Fit2form and Random Sampling:

Fit2form	Random	**RoboGAN-crowding**
3.81 ± 1.22	0.86 ± 0.81	$\mathbf{4.96 \pm 0.45}$

The hand-crafted controller is rather simple, resulting in a lower average distance travelled for each design than did the neural control. Even so, our algorithm is still able to find designs that are well-suited with this controller, demonstrating that our algorithm is agnostic to the controller used, as long as it is consistent at training and testing times.

VI. Conclusion and Future Work

In this work, we proposed a self-guided generative approach to task-based robot design generation in real-time. By combining ideas from both ML and EA, our method can quickly provide multiple suitable design candidates for a given task, and could be used to assist non-expert modular robot users with design prototyping. One limitation of our method, shared by other ML methods, is the assumption that new tasks will be from the same distribution as those seen during training, such that the outputs from out-of-distribution tasks may be poor. A second limitation is the need for the designs to be made from discrete components, where we do not currently have a method to include continuously varying parameters such as leg lengths, except to discretize them.

While all of our experiments are done in simulation, our generator could be directly deployed in real-world to generate high-quality designs. In the future, we plan to extend our work to larger design space, with more expressive design representations. Designs with different topologies will be introduced by adding an adjacency matrix [10] in addition to the node type matrix. We also plan to extend the scope of our tasks to more complex ones, such as those that involve simultaneous manipulation and navigation, as well as to other domains such as molecule generation for drug discovery. Future work will aim to establish a rigorous basis for substantiating the belief that a distribution drives a balance between exploitation and exploration during the search.

REFERENCES

[1] T. Wang, Y. Zhou, S. Fidler, and J. Ba, "Neural graph evolution: Towards efficient automatic robot design," in *International Conference on Learning Representations*, 2018.

[2] G. S. Hornby, H. Lipson, and J. B. Pollack, "Evolution of generative design systems for modular physical robots," in *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, vol. 4, 2001, pp. 4146–4151 vol.4.

[3] N. Cheney, R. MacCurdy, J. Clune, and H. Lipson, "Unshackling evolution: evolving soft robots with multiple materials and a powerful generative encoding," *ACM SIGEVOlution*, vol. 7, no. 1, pp. 11–23, 2014.

[4] J. Whitman, M. Travers, and H. Choset, "Modular mobile robot design selection with deep reinforcement learning," in *NeurIPS Workshop on ML for engineering modeling, simulation and design*, 2020.

[5] J. Whitman, R. Bhirangi, M. Travers, and H. Choset, "Modular robot design synthesis with deep reinforcement learning," in *Proc. of the AAAI Conf. on Artificial Intelligence*, vol. 34, 2020, pp. 10 418–10 425.

[6] H. Ha, S. Agrawal, and S. Song, "Fit2Form: 3D generative model for robot gripper form design," in *Conference on Robotic Learning (CoRL)*, 2020.

[7] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio, "Generative adversarial nets," in *Neural Information Processing Systems*, 2014.

[8] R. D. Hjelm, A. P. Jacob, T. Che, A. Trischler, K. Cho, and Y. Bengio, "Boundary-seeking generative adversarial networks," in *International Conference on Learning Representations*, 2018.

[9] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv preprint arXiv:1411.1784*, 2014.

[10] N. De Cao and T. Kipf, "MolGAN: An implicit generative model for small molecular graphs," *ICML 2018 workshop on Theoretical Foundations and Applications of Deep Generative Models*, 2018.

[11] G. L. Guimaraes, B. Sanchez-Lengeling, C. Outeiral, P. L. C. Farias, and A. Aspuru-Guzik, "Objective-reinforced generative adversarial networks (organ) for sequence generation models," *arXiv preprint arXiv:1705.10843*, 2017.

[12] O. Chocron and P. Bidaud, "Evolutionary algorithms in kinematic design of robotic systems," in *Proc. of the 1997 IEEE/RSJ Int'l Conf.on Intelligent Robots and Systems*, vol. 2, 1997, pp. 1111–1117.

[13] N. Casas, "Genetic algorithms for multimodal optimization: a review," *arXiv preprint arXiv:1508.05342*, 2015.

[14] D. A. Van Veldhuizen and G. B. Lamont, "Multiobjective evolutionary algorithms: Analyzing the state-of-the-art," *Evolutionary computation*, vol. 8, no. 2, pp. 125–147, 2000.

[15] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. PMLR, 2015, pp. 448–456.

[16] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375*, 2018.

[17] S. Katoch, S. S. Chauhan, and V. Kumar, "A review on genetic algorithm: past, present, and future," *Multimedia Tools and Applications*, pp. 1–36, 2020.

[18] O. J. Mengshoel and D. E. Goldberg, "The crowding approach to niching in genetic algorithms," *Evolutionary computation*, vol. 16, no. 3, pp. 315–354, 2008.

[19] J. Whitman, M. Travers, and H. Choset, "Learning modular robot control policies," 2021.

[20] "Hebi robotics," 2020. [Online]. Available: www.hebirobotics.com

[21] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning," http://pybullet.org, 2016–2019.

[22] X. Li, A. Engelbrecht, and M. G. Epitropakis, "Benchmark functions for cec'2013 special session and competition on niching methods for multimodal function optimization," *RMIT University, Evolutionary Computation and Machine Learning Group, Australia, Tech. Rep*, 2013.