Modular mobile robot design selection with deep reinforcement learning

Julian Whitman, Matthew Travers, and Howie Choset
Department of Mechanical Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
`jwhitman@cmu.edu`

Abstract

The widespread adoption of robots will require a flexible and automated approach to robot design. Exploring the full space of all possible designs when creating a custom robot can prove to be computationally intractable, leading us to consider modular robots, composed of a common set of repeated components that can be reconfigured for each new task. But, conducting a combinatorial optimization process to create a specialized design for each new task and setting is computationally expensive, especially if the task changes frequently. In this work, our goal is to select mobile robot designs that will perform highest in a given environment under a known control policy, with the assumption that the selection process must be conducted for new environments frequently. We use deep reinforcement learning to create a neural network that, given a terrain map as an input, outputs the mobile robot designs deemed most likely to locomote successfully in that environment.

1 Introduction

The robotics community has embraced modular robots for their potential to produce customized solutions for a broad variety of tasks. Our experience with modular robotics has revealed another advantage: they can help accelerate the robot development process by offering the user the opportunity to prototype, on real robots, different designs in quick succession. But, to use modular robots effectively, one must decide which components to use in the robot, where the best design may depend on the task. Our goal is to augment the robot engineering process by identifying promising designs for a given task. The contribution of this work is a design generation system which helps a user efficiently select a mobile robot design to prototype for a given environment.

Our long-term goal is to create a process that creates modular robot designs to complete a task in any specified environment. Prior modular design synthesis methods (1; 2) introduced the notion of incrementally constructing and searching a tree of modular arrangements for manipulators. We extend this idea to mobile robots, where each node added as a child to a current node represents adding a module to the robot, as shown in Figure 3. The construction of this tree can be viewed as a series of states and actions. Each state represents a partially complete design. Each action represents adding a module, forming edges between states on the tree. Under this formulation, we learn a state-action value function (3) which approximates the benefit of adding each module type given the task. We train a deep neural network to approximate this value function (4). Completed designs are simulated, and their resulting performance is used to learn about the capabilities of each design in each environment.

We recognize that the task/environment in reality can never exactly be replicated in simulation. Therefore, we require our algorithm to output multiple designs, and a ranking of their estimated performance, such that a user can physically test or choose between them. The algorithm can explore

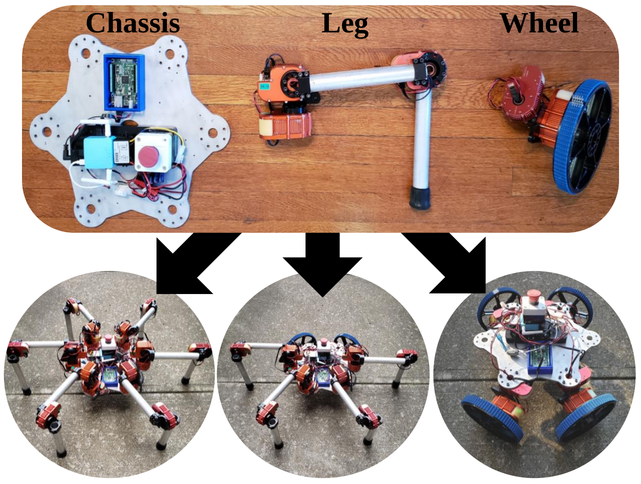


Figure 1: We present a deep reinforcement learning method to create a modular robot design generator. The robots used in this work are comprised of legs and wheels on chassis shared among all designs. Each limb type can be easily interchanged, lending the robot different capabilities on different terrains. In this picture, we show these modules used to create a hexapod, a robot with both legs and wheels, and a car.

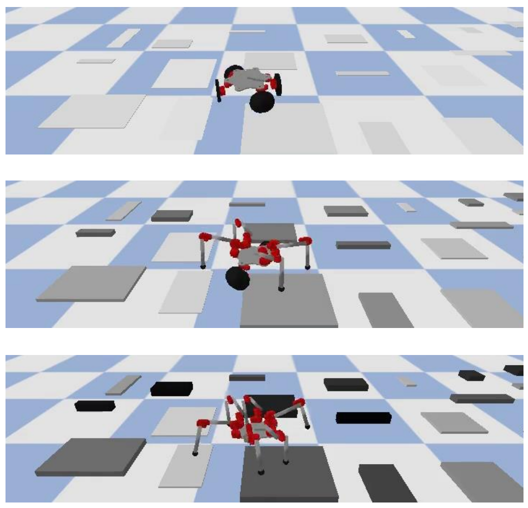
designs inexpensively *in silico*, providing new insights about designs that may have been previously unconsidered by an expert user, or providing evidence to support user intuition. In this work, we consider robots with various combinations of legs and wheels (Fig. 1); our intuition and experience leads us to think that on rough terrain, legs will perform better, and on smooth ground, wheels may perform better. Our algorithm can support or contradict such ideas in a data-driven manner, as well as suggesting less intuitive leg-wheel combinations that may have surprising capabilities.

2 Background

We find parallel design synthesis problems in the chemical engineering literature, where machine learning has been used to design novel molecules, selecting component atoms and bonds from a discrete set (5; 6; 7; 8). In particular, Zhou et al. (8) used a deep reinforcement learning paradigm for molecule discovery by sequentially add atoms to a molecule, which is similar to our robot selection methods.

Unlike the task of molecule generation, a challenge in the robot design problem is that the optimal design depends on both the task and the control policy, such that either a new policy must be created for each design, or a policy must be iteratively developed in concert with the design. Wang et al. (9) used an evolutionary algorithm to optimize robot design while learning a control policy. Schaff et al. (10) learned a policy and a distribution over designs at once, narrowing the distribution at each iteration to converge on an optimal design. Ha et al. (11) used a deep neural network to output both the design parameters and control actions. These methods optimize a single design and control policy for a given environment, and each search conducted is computationally expensive. As a result, if the environment is altered, these algorithms must be restarted, making them costly to use as a design space exploration tool in rapid-prototyping.

Deep function approximators optimized for a single policy and design, as in (9; 10; 11), do not retain information about how various designs performed for use in future searches. In contrast, we use a deep function approximator to encode a mapping between environment and robot enabling us to select a design for a given environment through computationally inexpensive inference rather than comparatively expensive training or evolution. We also output a distribution of multiple designs for a single terrain, presenting the user with more than one option to prototype. In our past work (1), a tree of modular manipulator designs was incrementally constructed and searched using a DQN. This work adapts this method to modular mobile robots.


(a)

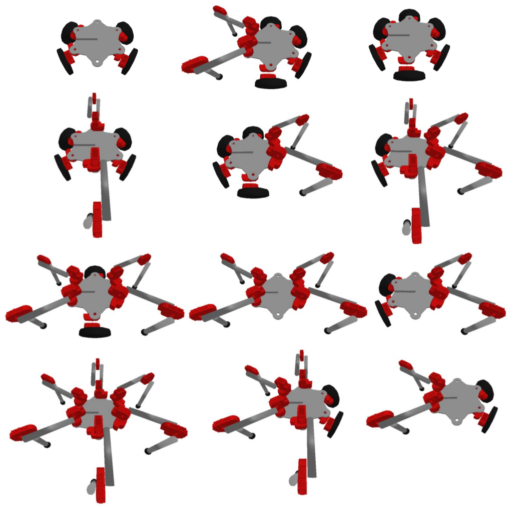

(b)

Figure 2: Our modular designs are evaluated in simulation to gather data on their performance over terrains of varying roughness. Then, our design selection algorithm is used to predict the best design for each environment. (a) For example, smooth terrain (top) may be well suited for wheels. The top image shows a simulated car robot in near-flat terrain. Terrain with low-lying features (middle) may be suited to a combination of legs and wheels. The middle image shows a simulated robot with both legs and wheels. Terrain with taller features (bottom) may be suited to robots with only legs. The bottom image shows a simulated hexapod on terrain with tall features. (b) This figure depicts designs considered valid during training. The left side of each of these designs is the "front". Each of these 12 designs has a different permutation of legs and wheels.

2.1 Deep Q-learning for Modular Robot Design

We treat the modular robot design problem as a finite-length Markov Decision Process with a discrete action space, in which the robot is constructed by adding one module at a time. We define a *complete* design as one that has attachments to all available ports specified, and a *partial* design as one that does not. At each time step t, the agent selects an action a_t that adds a module to one of the open ports on the chassis of the partial robot (see Fig. 3). The state s_t contains the partial design, so the next state s_{t+1} depends deterministically on only the previous state and the module added. Each action results in a new design state and a scalar reward r_t from the environment. In this context the set of all robot modules defines the action space $\mathcal{A}$, while the set of partial and complete robots defines the state space, $\mathcal{S}$.

We define the return at step t as $R_t = \sum_{t'=t}^{T} r_{t'}$. The state-action value function $Q_\pi : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ is then defined as the expected return given action a_t is taken in state s_t following policy $\pi : \mathcal{S} \mapsto \mathcal{A}$, $Q_\pi(s,a) = \mathbb{E}[R_t | s_t = s, a_t = a]$. Q-learning estimates the optimal state-action value function Q^*, which can be defined in terms of the Bellman equation,

$$Q^*(s_t, a_t) = \max_{\pi} \mathbb{E}\left[r_t + \max_{a' \in \mathcal{A}} Q^*(s_{t+1}, a') \right]. \quad (1)$$

Deep Q-networks (DQN) use a deep neural network as a function approximator $Q(s,a;\theta)$ with network parameters θ to approximate $Q^*(s,a)$ (4). We train this network with experience replay (12) and a target network (13). We condition the value outputs on the task (14), enabling the DQN to apply to a range of tasks (in this case, environments to be traversed).

3 Methods

In order to select a robot design for a given terrain, we learn a *design generator* $G : \mathcal{T} \to \mathcal{D}$ which maps from a terrain grid $\tau \in \mathcal{T} \subset \mathbb{R}^{L \times W}$ (terrain height measurements with length and width

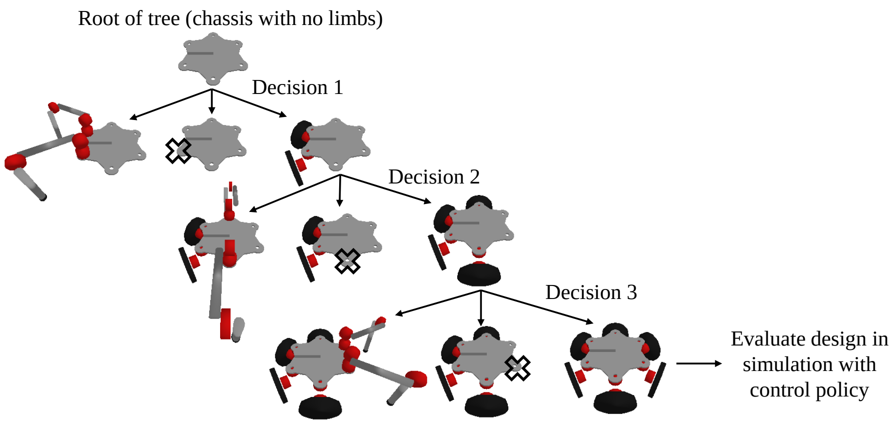

Root of tree (chassis with no limbs)
Decision 1
Decision 2
Decision 3
Evaluate design in simulation with control policy

Figure 3: We search for modular mobile robot designs by viewing the design space as a tree, in which modular limbs are sequentially added from front to back on the chassis. A deep neural network is used to learn a state-action value function for each decision made on this tree, conditioned on the terrain that the robot will operate in. This figure shows an example sequence of decisions on such a tree. At the root lies a chassis with no modules, and each step adds a module, resulting leaf nodes containing robots with various permutations of legs, wheels, and ports deliberately left open.

resolution $L \times W$) to a design $d \in \mathcal{D}$ (the space of possible designs). The design is evaluated using a pre-defined controller. Terrains have randomly distributed density and height of terrain features, where samples from the distribution $\mathbf{T}$ are elements of $\mathcal{T}$. We optimize the parameters of ϕ of the generator neural network to maximize the expected robot performance criteria P over the terrain distribution,

$$\phi^* = \underset{\phi}{\operatorname{argmax}}\, \mathbb{E}_{\tau \sim \mathbf{T}} \big[P(G_\phi(\tau), \tau) \big]. \tag{2}$$

Here, the performance P is the distance travelled by the design over the terrain in a fixed time span, explicitly dependent both on the design $d = G_\phi(\tau)$ and terrain. Designs are drawn from the design generator, which learns to output designs for an input terrain. After training, the design generator is used to identify promising designs for a given terrain.

The performance of a design depends on how that design will be controlled. Our design selection method is agnostic to the particular control policy used, as long as each design uses the same controller consistently. The performance criteria P is queried by simulating the control policy for a fixed time span and measuring the final x-position of the robot, averaged over multiple trials. We use a modular reactive policy network which directs the robot forwards along the x-axis through the terrain, and corrects its course toward the x-axis, regardless of what components are present in the robot. This method is currently under review, and we plan to include further details about the control policy in future versions of this work. A high-level controller observes the robot position and sends a body-frame heading command to the mid-level controller. The mid-level control takes the heading command, robot IMU readings, and joint sensor readings, and sends joint-level commands. We use a Pybullet simulation (15) with robot models corresponding to physical hardware (Fig. 2) made from components produced by Hebi Robotics (16).

3.1 Module selection DQN

Our algorithm assembles a mobile robot one module at a time, as illustrated in Figure 3, using a deep Q-network to choose modules. In this section we define the states, actions, and reward signals in greater detail.

We encode the design d as a list of one-hot vectors, where each index in a single vector indicates a type of module selected, with a user-set maximum number of modules in the arrangement N_{max}. Modules that are not yet chosen in the design are are represented by vectors of zeros to maintain a fixed-size input. We currently restrict our designs to symmetric designs on a chassis with six ports, leading to $N_{max} = 3$ modules to be chosen. The terrain τ contains the height of the terrain over a grid of points, and is passed first into convolutional layers. The output of the convolutional layers are

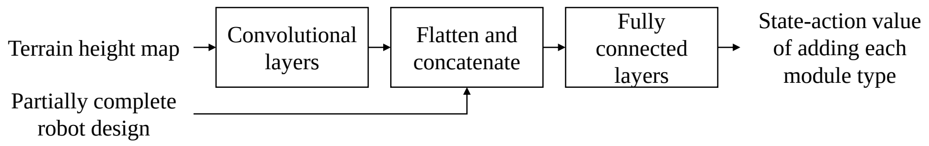


Figure 4: The neural network used takes the terrain grid, passed through as series of convolutional layers (Conv2D), and the design encoding, then passed through a series of fully connected (FC) layers. The output of the network is interpreted as the state-action value of each module type that could be added to the partially complete robot design.

flattened and appended to the current design, and passed through a series of fully connected layers with ReLU activation (Fig. 4).

At each step, an action indicates a module type from the set of N_m module types. We use $N_m = 3$ module types: legs, wheels, or none (allowing unoccupied ports). We append onto the design an additional one-hot vector of length N_{max} which is set to the index of the current open port on the chassis, indicating which port the next module will be added to. The output of the network is interpreted as the state-action values of adding each type of module to the partial design. An episode always ends after N_{max} actions, such that the ports on the robot have either been assigned modules or designated as deliberately unoccupied.

Each action results in a reward $r = 0$ except for the terminal (third) action. Note that if an additional cost were added to (2) to penalize for number or mass of modules, non-terminal rewards could be used to alter the output designs accordingly, as was the case in (1). At the terminal action, the completed robot design is evaluated. We deem some designs undesirable, for instance those with the front or back port unoccupied, and thus treat them as invalid and assign them a terminal reward of -10. Valid designs are sent to the simulator to evaluate their performance within the input terrain. Multiple simulations are run for the same robot and terrain, with slightly perturbed intial states, to obtain an average performance for that terrain. The average distance travelled in meters after 150 steps is then returned as the reward.

In our current implementation, we allow only 12 valid designs, shown in Fig. 2. An alternate formulation of our method would be to learn the total value of each of the designs separately, or even to exhaustively simulate all designs and rank them. However, we expect such approaches would not scale as the number of modules on the design increases– for instance, even with these same components, were we to pick the limbs on the left and right side of the robot independently, there would be 144 (over 10 times as many) possible designs . Our algorithm has an action space scaling with the types of modules, meant to address this combinatorial explosion in state space size. In our ongoing work, we are expanding the robot design space with a wider range of possible module combinations.

3.2 Training process

During training, state-action values are learned from randomized terrains. At each episode, a terrain is created from randomly placed blocks, with upper and lower bounds on maximum block height and minimum distance between blocks. The height at a grid of points on this terrain is measured as input to the DQN. The DQN is called repeatedly, each time with the terrain and current design as input. At first, the design input is empty, and after each call to the DQN, a module is added to the design. The states, actions, and rewards are stored in a replay memory buffer. We use Boltzmann exploration with a temperature hyperparameter that is lowered over the course of training. After each episode, we sample mini-batches from the replay buffer and step the optimizer.

3.3 Sampling tree to get multiple designs

The design selection networks are first trained to recognize patterns in how various combinations of modules contribute to effective locomotion over a given terrain. After training, the design selection network is used to conduct a computationally efficient design search. To use the network for inference, first the height map is measured of the terrain to traverse. Then, that height map is input to the generator network $M = 100$ times in a batch, and design choices are made by interpreting the softmax

Table 1: Preliminary results from design selection network. On each terrain, with low, mid, or high roughness, we compare the output designs d with the five highest performance estimates (P) from the network, or the minimum and maximum from three simulation runs. Designs are specified by the modules chosen: a leg (l), wheels (w) or none (n) on each port. The at least four of the top five designs overlap between the estimated and simulated average performance in the three terrains.

Low Terrain, 5/5 Match				Mid Terrain, 5/5 Match				High Terrain, 4/5 Match			
Estimated		Simulated		Estimated		Simulated		Estimated		Simulated	
d	P	d	P min-max	d	P	d	P min-max	d	P	d	P min-max
lwl	10.9	lww	10.7 - 11.5	lnw	4.6	lwl	7.5 - 9.5	lll	3.9	wll	4.6 - 5.2
wwl	10.3	wwl	11.0 - 11.1	lwl	4.6	lll	4.2 - 4.9	wll	3.1	lll	4.7 - 4.9
lww	10.3	lwl	10.9 - 11.1	lll	4.6	lnw	3.5 - 5.6	lnw	2.7	lnl	3.9 - 4.5
wnl	8.1	wnl	8.7 - 9.7	wll	4.3	wll	1.5 - 5.4	lwl	2.6	lwl	2.0 - 5.0
wll	7.0	wll	6.8 - 7.2	llw	4.2	llw	2.3 - 5.3	llw	2.6	lnw	2.0 - 4.0

of the Q-value outputs as the weights of a categorical distribution. This results in a set of designs $d_1 \dots d_M$, which may contain duplicates. The Q-values at the final step are estimates of the expected reward obtained by each design. We then sort the output designs by terminal Q-value, to obtain a ranking of the top designs for that environment. This means that we can obtain multiple designs to prototype and deploy rather than only a single design, along with estimates of their performance.

4 Results

To evaluate the trained design selection network, we applied it to three test environments with different randomly generated terrain distributions, from low (nearly flat) to high (frequent high terrain features). We sampled designs using the procedure described above, and collected a ranking of the best-performing designs. Then, we simulated all 12 valid designs in each environment as a basis of comparison. Each design was driven multiple times through the same environment to obtain an average performance. We compared the best designs in simulation with the estimated best designs from the DQN.

The results of this experiment are summarized in Table 1. The predicted top five designs from the estimator and simulation overlap with 4 or 5/5 designs in each terrain. Obtaining an exact overlap in rankings between estimator and simulation is difficult, as there is high variability in performance. For instance, we observed on some terrains there are patches on which the robot may become stuck on some trials but may narrowly avoid on others. In our ongoing analysis we are developing additional metrics by which to judge the output of the network.

After training, the network can be used in real-time to generate designs conditioned on the terrain. We made an interactive graphical user interface, in which a slider bar changes the height of the randomly generated terrain. The robot design is updated and simulated in real-time as the environment changes, allowing us to quickly investigate how different terrain feature distributions effect the optimal design, without additional training or intensive computation. A video showing this interface can be found at `https://youtu.be/f3PhXnuxk7g`.

5 Ongoing work

Our current work involves applying design selection methods to physical terrain and robots. We can scan a region of interest with a hand-held RGB-D sensor to create a terrain map, and designate the start point and desired direction of motion. This terrain map will be entered to the DQN to select suggested designs, which will be constructed and deployed in reality.

Before training, we specify controllers for all possible designs, then the performance of the robot in a given environment is conditional on the efficacy of that controller. The current control policy used for rough terrain does not include exteroceptive measures of the environment, that is, it cannot preemptively adapt its behavior to upcoming terrain and only adapts to what is sensed through proprioception. Our design selection method can still be applied as more complex control methods are developed, or as environment-dependent longer-horizon planning is added to the controller.

6 Broader Impacts

Our goal is to augment the conventional robot engineering process by considering design and control at once. Our methods provide an avenue for exploring the relative benefits of distinct design components, for instance the choice between legs and wheels, in a data-driven manner. We hope that in the future this will enable non-expert users to customize robots, and that design synthesis algorithms will have a wide reach, given the potential applications for modular systems in manufacturing, defense, or space robotics.

7 Acknowledgements

This work was supported by NASA Space Technology Research Fellowship NNX16AM81H and by a Siemens Futuremakers Fellowship.

References

[1] J. Whitman, R. Bhirangi, M. J. Travers, and H. Choset, "Modular robot design synthesis with deep reinforcement learning.," in *AAAI*, pp. 10418–10425, 2020.

[2] S. Ha, S. Coros, A. Alspach, J. M. Bern, J. Kim, and K. Yamane, "Computational design of robotic devices from high-level motion specifications," *IEEE Transactions on Robotics*, vol. 34, no. 5, pp. 1240–1251, 2018.

[3] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[5] B. Sanchez-Lengeling and A. Aspuru-Guzik, "Inverse molecular design using machine learning: Generative models for matter engineering," *Science*, vol. 361, no. 6400, pp. 360–365, 2018.

[6] N. De Cao and T. Kipf, "Molgan: An implicit generative model for small molecular graphs," *arXiv preprint arXiv:1805.11973*, 2018.

[7] R. Gómez-Bombarelli, J. N. Wei, D. Duvenaud, J. M. Hernández-Lobato, B. Sánchez-Lengeling, D. Sheberla, J. Aguilera-Iparraguirre, T. D. Hirzel, R. P. Adams, and A. Aspuru-Guzik, "Automatic chemical design using a data-driven continuous representation of molecules," *ACS central science*, vol. 4, no. 2, pp. 268–276, 2018.

[8] Z. Zhou, S. Kearnes, L. Li, R. N. Zare, and P. Riley, "Optimization of molecules via deep reinforcement learning," *Scientific reports*, vol. 9, no. 1, pp. 1–10, 2019.

[9] T. Wang, Y. Zhou, S. Fidler, and J. Ba, "Neural graph evolution: Automatic robot design," in *Int. Conf. on Learning Representations*, 2019.

[10] C. Schaff, D. Yunis, A. Chakrabarti, and M. R. Walter, "Jointly learning to construct and control agents using deep reinforcement learning," in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 9798–9805, IEEE, 2019.

[11] D. Ha, "Reinforcement learning for improving agent design," *Artificial Life*, vol. 25, no. 4, pp. 352–365, 2019.

[12] M. Riedmiller, "Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method," in *European Conference on Machine Learning*, pp. 317–328, Springer, 2005.

[13] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Thirtieth AAAI conference on artificial intelligence*, 2016.

[14] T. Schaul, D. Horgan, K. Gregor, and D. Silver, "Universal value function approximators," in *Proceedings of the 1st Annual Conference on Robot Learning*, pp. 1312–1320, 2015.

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

[16] “Hebi Robotics, 2020. [Online]. Available: www.hebirobotics.com.”